Daniel Pelis
CPE 810
10/23/2020

Lab 2: Histogram

Objective

Implement a histogram routine using atomic operations and shared memory in CUDA. Your code should be able handle arbitrary input vector sizes (vector values should be randomly generated using integers between 0~1024).

Instructions

Make a new folder for your project. Copy and modify the code of histogram in CUDA sample code to include the following key functions:

1. allocate device memory
2. copy host memory to device
3. initialize thread block and kernel grid dimensions
4. invoke CUDA kernel
5. copy results from device to host
6. deallocate device memory
7. implement the routine using atomic operations and shared memory
8. handle thread divergence when dealing with arbitrary input sizes

Your final executable can be run using the following command:

./histogram_atomic -i <BinNum> <VecDim>

Input parameters:

<VecDim> is the dimension of the input vector.

<BinNum> is the number of bins (any integer that can be written as 2^k where k can be any integer from 2 to 8).

Implementation

My implementation of a histogram CUDA program, begins by taking in the user's input for the vec_size and bin_num variables. If these values are not supplied, it will default to a vec_size of 1000 and a bin_num of 64. The program also immediately passes the bin_num to check if can be expressed with 2^k where k can be any integer from 2 to 8. If the bin_num fails, this check it defaults back to 64 bins. The input arguments are also check for a '-d' flag. If the user provides this flag, they have the option to pass in the grid and block dimensions, respectively. The program then begins to allocate memory for the arrays stored in the host memory. The size of the bins is calculated by dividing 1024 (the possible amount of unique value) by the number of bins. The data array is filled with random values between 0 and 1024 by taking the modulus of a random number and 1024. Now pointers for the device arrays are created and memory is allocated on the GPU's global memory. Finally, the data array is

copied to the device memory and the grid and block dimensions are set. Since the data is a 1-D array, the grid and blocks will also be one dimensional. Now that all preparations have been made, the kernel is called and the device data pointer, device histogram pointer, vec_size, bin_size, and bin_num are passed in. Additionally, the memory size of the histogram is passed in with the grid and block dimensions so that it can be used to allocate shared memory by the threads.

The kernel that is called utilizes the second strategy discussed in the textbook, which has the threads deal with elements in the data array sequentially as to enable memory coalescing. This is done by assigned each thread elements separated by an offset equal to the total number of threads. Before this, all the threads of a block help each other to create a private version of the histogram array in their shared memory. Before moving on all threads wait for each other with a __syncthreads() command. Now the kernel declares variables for the current and previous histogram indexes and a variable for the accumulator. These are part of the accumulation method that will be discussed later. The previous index is initialized to -1 to avoid any issues and the accumulator to 0. The kernel now loops through the thread's individual assigned elements in the data array. This is done by starting at the position of the data array equal to the thread's index and moving up by "blockDim.x" positions every iteration. Inside this loop the current value is stored in a variable and the current histogram position is calculated by taking the floor of the current value divided by the size of a bin. There is a boundary check to ensure the value is between 0 and 1024 and the current histogram index is set to the index calculated just before. If the current index is different from the previous index and the accumulator is not zero, the accumulated value will be added to the histogram with an atomic add operation at the previous index and the accumulator will be reset. The previous index will then be set to the value of the current index and the loop will reset. If the previous index and the current index are equal the accumulator will simply be incremented. After the loop has completed the accumulator is checked one more time and if it is not zero, then its value will be added to the previous index value with an atomic add operation. Now that the private histogram for the block has been filled the threads will sync again. Lastly, the threads help add their private histogram to the histogram stored in global memory using atomic additions.

After the kernel completes, the histogram stored in global memory is copied back to the host and the device memory is freed. The performance metrics are calculated using the execution time of the kernel and the calculated number of flops. A function to calculate the histogram for the same data array is then run on the host to be used as a validation reference. The values of both the host histogram and the device calculated histogram are compare and an error prints if there is an inequality found. The values of the device calculated histogram are also summed to ensure that no values were missed by ensuring the sum equals the input vector size. Lastly the host memory is freed to avoid memory leakage.

Questions

Consider an N-dimensional input vector for the histogram computation with M bins:

(1) Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance. Which optimizations gave the most benefit?

Like the textbook I began by writing a kernel that utilized a block partitioning method, which split the input vector into chunks for each thread to work on. For example, if the input vector size were 50 and there were 5 threads, thread 0 would work on element 1 to 10, thread 2 on element 11 to 20, and so on. One major issue with this method is that the threads try access completely different areas of the global memory at the same time, leading to increased global memory latency. To combat this, I switched to an interleaved partitioning strategy that would enable memory coalescing. Now that the threads are accessing sequential memory locations multiple values can be loaded in one memory access. In this method the elements processed by a particular thread are separated by an offset equal to the total number of threads. The next optimization delt with the case where multiple atomic operations need to be done on the same element. When an atomic operation is executed the memory location is locked down,

so no other operation can be performed on it at the same time. This means that if multiple atomic operations must be done on the same element, they must wait for each other to finish. This issue is intensified by the high latency of global memory calls that each atomic operation must make. The solution to this issue was to apply a privatization technique to the kernel. This technique created private version of the histogram for every thread block in shared memory. Now most of the atomic operations will be done to the shared memory which has much lower latency than global memory. The only atomic operations done to global memory now, are the operations done to copy each thread block's private histogram to the main histogram stored in global memory. The last optimization I made was to decrease the amount of atomic operations, which are inherently long latency operations. This was done by employing an aggregation technique. This technique keeps count of how many times an element of a certain value appears in a row. Once a new value is encountered, the count is added to the histogram and a new count is kept for the new value. Now the kernel no longer needs to perform an atomic operation for every element. If a multiple elements in a row have the same value, their atomic operations are combines. This further cuts down on the kernel's latency by cutting out unnecessary long latency operations.

(2) How many global memory reads (write) per input element are being performed by your kernel? Explain.

Since my kernel divides the input vector so that each element is handled by only one thread, the amount of global memory reads would be equal to the size of the input vector. So, in the case of a N-dimensional input vector, there would be ***N*** global memory reads.

The amount of global memory writes has been decreased by the implementation of the privatization technique. Originally, the amount of global memory writes would have also been equal to the size of the input vector, since for every element its corresponding histogram bin was incremented using a global memory write. This would have led the amount of global memory writes for the case of a N-dimensional input vector, to be *N*. The privatization technique makes it so that each thread only writes to the global memory at the end of the kernel when it helps add its block's private histogram to the global memory histogram. The threads are organized so that one global memory write will be performed for each bin in the block's private histogram. This process is executed by each thread block, so the number of writes can be calculated by multiplying the number of bins by the grid dimension. Therefore, the number of global memory writes with privatization can be expressed as, $\boldsymbol{M * GridDim.x}$ assuming the histogram has M bins.

(3) How many atomic operations are being performed? Explain.

The number of atomic operations performed by my kernel changed as I made optimizations. Originally, without privatization or aggregation, the number of atomic operations was *N*, assuming an N-dimensional input vector. This number comes from the fact that each thread will perform an atomic operation to increment the appropriate bin of the histogram for each of its elements. Additionally, each element is only handled by one thread, meaning no threads overlap elements.

When a privatization technique was added, it increased the amount of atomic operations. However, since most of these operations were done to shared memory, they were completed much faster. With privatization, each thread still performs an atomic operation for every element is handles, however this operation is now on its block's private copy of the histogram in shared memory. Therefore, we still have the *N* atomic operations from before. However, now the threads must help add their block's private histograms to the main one stored in shared memory. This adds an additional $M * GridDim.x$, assuming *M* is the number of bins. This means overall there are $N + M * GridDim.x$ atomic operations, but N of these operations are on shared memory and $M * GridDim.x$ are on global memory. Although the total

number of atomic operations has increased, the number of operations on global memory should decrease, depending on the bin size and grid size.

Although aggregation does not guarantee a decrease in atomic operations, it creates the opportunity for the kernel to perform less atomic operations depending on the elements. If all of the values of the elements were the same there would be $(BlockDim.x + bin_{size}) * GridDim.x$ atomic operations. Since all the values are the same, the threads will simply keep a count of how many elements they get and add this count to their block's private histograms in one operation. This creates a *thread_count* number of atomic operations. Each thread also still has to help add their block's private histogram to the one in global memory, resulting in $bin_{size} * GridDim.x$ atomic operations. Finally, we get $GridDim.x * BlockDim.x + bin_{size} * GridDim.x = (BlockDim.x + bin_{size}) * GridDim.x$. If all the values in the input vector were completely random then the number of atomic operations would not be affected by aggregation, since each thread would have to increment the histogram for every value. This results in $\boldsymbol{N + M * GridDim.x}$ atomic operations.

(4) How many contentions would occur if every element in the array has the same value? What if every element has a random value? Explain.

The amount of contentions that will occur will vary depending on the optimizations I include (privatization, aggregation) and the values of the elements in the array (all same, all random). First, assuming all the values in the array are the same and no optimizations have been made there would simply be the *N* contentions, where N is size of the input vector. This occurs because all the threads are trying to add to the same histogram bin, meaning at any time all threads executing will clash with each other. Now adding privatization, the number of contentions when accounting for the elements in the data array should stay at N. Even though privatization causes threads to work on histograms private to their blocks, threads will still be causing contentions in these private histograms. This means there will still be *N* contentions at this step, they will just be resolved faster due to the shared memory's lower latency. The number of contentions that occur in the last step of the kernel, depends mainly on the number blocks running at the same time. For this example, we can assume that all the blocks of the grid are running at the same time. Therefore, the number of contentions at this step would be equal to M * GridDim.x, where M is the number of bins. Therefore, with privatization we can express the number of contentions as $N + M * GrimDim.x$. If an extra condition was added so that the blocks only copied non-zero histogram values to the global memory, then the contentions could be brought down to $N + GrimDim.x$ for this case. Now adding in an aggregation technique could greatly reduce the amount of contentions when dealing with the data array. Since all the values are the same, each thread will accumulate all the elements its responsible for. Therefore, the N contentions will become $GridDim.x * BlockDim.x$ contentions since only the threads in a block will contend when adding their accumulated value. This brings the total contentions to $\boldsymbol{GridDim.x * BlockDim.x + M * GrimDim.x}$. In this case a lower thread count will cause less contentions.

Assuming that all the values in the histogram were random would change the expected number of contention in the kernel. Also, assuming that the thread count is low enough where no running threads deal with values in the same bin at the same time, there will be zero contentions while using just the interleaving technique. Adding privatization will increase the amount of contentions, but there should still be an increase in performance due to the lower latency of shared memory. Again, assuming that the numbers are random enough that no threads will contend while adding to their block's private histogram. However, there will be contentions at the end of the kernel when the threads help add their block's private histogram to the global memory. Here we will see $\boldsymbol{M * Grid.Dim.x}$ contentions since each block needs to add each of its histogram values to global memory. Finally adding the aggregation

optimization should make no change to the number of contentions, since we assumed there were already no contentions in the data array portion of the kernel with privatization.

(5) How would the performance (GFLOPS) change when sweeping from 4 to 256 (k=2 to 8)? Compare your predicted results with the realistic measurements when using different thread block sizes.

For this kernel the amount of operations executed will be $N + M * GridDim.x$, since we get one atomic addition for every element in the input vector (N) and one atomic addition for every bin of every block's private histogram (M * GridDim.x). Theoretically as we increase M from 4 to 256, we should see a general increase in the GFLOPS performance since more operations are being executed. Even though this means more contentions when trying to add the private histograms to the global histogram, there should be a large decrease in the number of contentions when dealing with the input vector values. Since there are more bins to divide the values between there should be less instances where multiple threads are adding to the same value.

After performing the tests in the "Experimentation" section below. I was able to observe the actual kernel results generally followed the trend that theoretical calculations described. As I increased the bin size from 4 to 256 in my tests, I saw an increase in GFLOPs performance. There were a few runs that contain discrepancies in the results, which showed a lower performance for higher bin counts. This can most likely be attributed to the fact that the vectors are filled with random values. Since the input values are random it would be very difficult to derive a theoretical equation that accounts for the increase in performance due to aggregation. Some runs could contain more instances of threads encountering the same value multiple times in a row, than others. This could cause those threads to perform better than other even if they have lower thread counts comparatively. A way to combat this an get better results would be to take multiple readings of a single run and average the performance values. This method could possibly remove any discrepancies caused by the randomness of the input values.

(6) Propose a scheme for handling extremely large data set that cannot be fully stored in a single GPU's device memory. (Hint: how to design an implementation for efficiently leveraging multiple GPUs for parallel histogram computation?)

Assuming the input array passed in is too large to fit on a single device's memory, we could implement a technique like privatization. The size of the device's global memory can be determined by querying the devices and accessing each device's "totalGlobalMem" attribute. We can then divide the input array between the devices. The algorithm to determine how many elements of the input array would be something resembling the following, $\frac{(global\_mem\_size - bin\_num * 4)}{4}$. The 4 comes from the byte size of an integer in memory. Once we determine the chunks of the input vector that each device will handle, we can store each chunk in its own vector on the host and pass that into the kernel as the input vector. We can choose which device to execute the kernel on by using the "cudaSetDevice()" function. When the kernel is finished the host code can copy the histogram created by that device to a master histogram held in the host's memory. After all the devices finish, the histogram stored in the host's memory would contain all the information for the entire input vector.

Experimentation

To choose appropriate grid and block dimensions for the rest of my experimentation, I looked at the performance difference by varying the dimensions of the grid and blocks. I used an example with 8 bins and 10,000

elements in the input vector and then tested different combinations grid and block dimensions. The results can be viewed below in Table 1.

GFLOPS	BlockDim.x = 32	BlockDim.x = 64	BlockDim.x = 128
GridDim.x = 10	0.09	0.23	0.34
GridDim.x = 100	0.7	0.96	1.28
GridDim.x = 1000	1.49	1.19	0.74

Table 1: Dimension Tests (Bin_num = 8; Vec_size = 10,000)

Based on the results of the experiments above, I will be using the two highlighted grid and block dimension combinations for the following experimentations. 1000 by 32 was chosen since it had the highest performance and 1000 by 64 was chosen since it will most likely outperform the 32-blockDim combination in tests with a very larger input vector size. Next, I moved on to testing different combinations of bin numbers and vector sizes. The bin numbers will begin at k =2 and be increment up to k = 8 for 2^k (4, 8, 16, 32, 64, 128, 256). The input vector size will begin at 1,000 and be incremented by a factor of ten to 1,000,000. Results for these tests can be viewed in Tables 2 and 3.

GridDim.x = 1000, BlockDim.x = 32

	Vec_size = 1,000	Vec_size = 10,000	Vec_size = 100,000	Vec_size = 1,000,000
Bin_num = 4	0.48	1.12	2.81	3.35
Bin_num = 8	0.84	1.17	2.39	3.49
Bin_num = 16	1.47	2.11	3.43	3.36
Bin_num = 32	2.20	2.40	2.65	3.73
Bin_num = 64	2.69	2.52	3.03	3.18
Bin_num = 128	4.33	3.14	4.64	3.27
Bin_num = 256	4.29	5.73	4.61	3.74

Table 2: GFLOP results for 32-thread blocks

GridDim.x = 1000, BlockDim.x = 64

	Vec_size = 1,000	Vec_size = 10,000	Vec_size = 100,000	Vec_size = 1,000,000
Bin_num = 4	0.17	0.76	2.21	4.41
Bin_num = 8	0.64	1.00	3.39	4.42
Bin_num = 16	1.18	1.62	3.43	3.96
Bin_num = 32	2.27	2.50	3.95	4.78
Bin_num = 64	3.06	3.60	4.77	4.66
Bin_num = 128	3.15	3.73	4.85	4.58
Bin_num = 256	5.23	5.64	5.35	5.59

Table 3: GFLOP results for 64-thread blocks

After observing the resulting of the two series of test, we can conclude that in general there was an increase in performance as the number of bins was increased. This is mostly likely due to less contentions during the execution of the kernel. Since there are more bins a value could possibly fit in, it is less likely that multiple threads will try to access the same memory location at the same time. This translates into less threads waiting for a memory location to become available before they can perform their atomic addition.

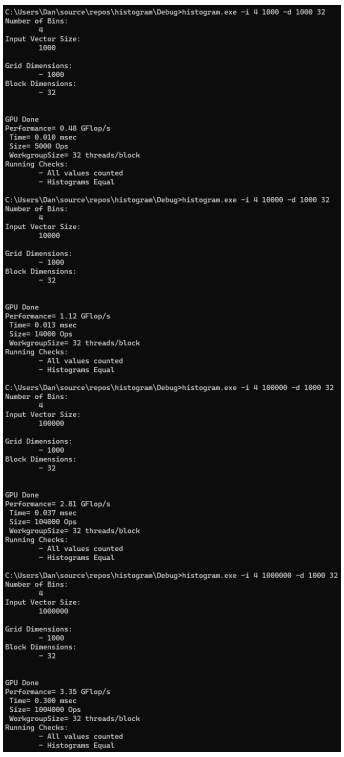
Additionally, it seems the hypothesis that the 64-thread block would perform better on the larger input sets was true. Except for some outlying cases, the 32-thread blocks outperformed the 64-thread blocks for the 1,000 and 10,000 vec_size runs, while the opposite is true for the 100,000 and 1,000,000 vec_size runs. The higher thread count runs perform better on the larger data sets because they are taking better advantage of the device's available resources. The higher thread count runs also do worse on the smaller dataset runs because they are running many more threads than there are elements in the input vector.

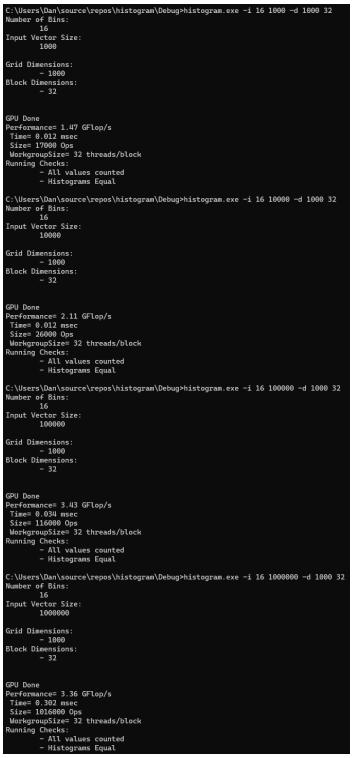
Finally, both series of tests show a general increase in GFLOPs as the vector size increase with the same number of bins. This most likely occurs because the kernel is performing more operations, however the number of operations needed to copy the histogram over stays constant. Since the operations performed using the input

vector elements are done to the private histograms in shared memory, they can be performed relatively quick. The high latency operations in the kernel are the additions done to copy the private histograms to global memory. Since the number of slow operations stays the same and the number of fast operations increases, the kernel can now perform many more FLOPs with a much smaller increase in execution time.

Experimental Images

GridDim.x = 1000, BlockDim.x = 32:





```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 1000 -d 1000 32
Number of Bins:
        32
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 2.20 GFlop/s
 Time= 0.015 msec
 Size= 33000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 10000 -d 1000 32
Number of Bins:
        32
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 2.40 GFlop/s
 Time= 0.017 msec
 Size= 42000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 100000 -d 1000 32
Number of Bins:
        32
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 2.65 GFlop/s
 Time= 0.050 msec
 Size= 132000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 1000000 -d 1000 32
Number of Bins:
        32
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.73 GFlop/s
 Time= 0.277 msec
 Size= 1032000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 1000 -d 1000 32
Number of Bins:
        64
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 2.69 GFlop/s
 Time= 0.024 msec
 Size= 65000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 10000 -d 1000 32
Number of Bins:
        64
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 2.52 GFlop/s
 Time= 0.029 msec
 Size= 74000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 100000 -d 1000 32
Number of Bins:
        64
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.03 GFlop/s
 Time= 0.054 msec
 Size= 164000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 1000000 -d 1000 32
Number of Bins:
        64
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.18 GFlop/s
 Time= 0.334 msec
 Size= 1060000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 1000 -d 1000 32
Number of Bins:
        128
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 4.33 GFlop/s
 Time= 0.030 msec
 Size= 129000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 10000 -d 1000 32
Number of Bins:
        128
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.14 GFlop/s
 Time= 0.044 msec
 Size= 138000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 100000 -d 1000 32
Number of Bins:
        128
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 4.64 GFlop/s
 Time= 0.049 msec
 Size= 228000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 1000000 -d 1000 32
Number of Bins:
        128
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.27 GFlop/s
 Time= 0.345 msec
 Size= 1128000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 1000 -d 1000 32
Number of Bins:
        256
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 4.29 GFlop/s
 Time= 0.060 msec
 Size= 257000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 10000 -d 1000 32
Number of Bins:
        256
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 5.73 GFlop/s
 Time= 0.046 msec
 Size= 266000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 100000 -d 1000 32
Number of Bins:
        256
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 4.61 GFlop/s
 Time= 0.077 msec
 Size= 356000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 1000000 -d 1000 32
Number of Bins:
        256
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 32


GPU Done
Performance= 3.74 GFlop/s
 Time= 0.336 msec
 Size= 1256000 Ops
 WorkgroupSize= 32 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

GridDim.x = 1000, BlockDim.x = 64:

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 4 1000 -d 1000 64
Number of Bins:
        4
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 0.17 GFlop/s
 Time= 0.030 msec
 Size= 5000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 4 10000 -d 1000 64
Number of Bins:
        4
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 0.76 GFlop/s
 Time= 0.018 msec
 Size= 14000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 4 100000 -d 1000 64
Number of Bins:
        4
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 2.21 GFlop/s
 Time= 0.047 msec
 Size= 104000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 4 1000000 -d 1000 64
Number of Bins:
        4
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.41 GFlop/s
 Time= 0.228 msec
 Size= 1004000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 8 1000 -d 1000 64
Number of Bins:
        8
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 0.64 GFlop/s
 Time= 0.014 msec
 Size= 9000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 8 10000 -d 1000 64
Number of Bins:
        8
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 1.00 GFlop/s
 Time= 0.018 msec
 Size= 18000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 8 100000 -d 1000 64
Number of Bins:
        8
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.39 GFlop/s
 Time= 0.032 msec
 Size= 108000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 8 1000000 -d 1000 64
Number of Bins:
        8
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.42 GFlop/s
 Time= 0.228 msec
 Size= 1008000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 16 1000 -d 1000 64
Number of Bins:
        16
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 1.18 GFlop/s
 Time= 0.014 msec
 Size= 17000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 16 10000 -d 1000 64
Number of Bins:
        16
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 1.62 GFlop/s
 Time= 0.016 msec
 Size= 26000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 16 100000 -d 1000 64
Number of Bins:
        16
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.43 GFlop/s
 Time= 0.034 msec
 Size= 116000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 16 1000000 -d 1000 64
Number of Bins:
        16
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.96 GFlop/s
 Time= 0.257 msec
 Size= 1016000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 1000 -d 1000 64
Number of Bins:
        32
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 2.27 GFlop/s
 Time= 0.015 msec
 Size= 33000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 10000 -d 1000 64
Number of Bins:
        32
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 2.50 GFlop/s
 Time= 0.017 msec
 Size= 42000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 100000 -d 1000 64
Number of Bins:
        32
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.95 GFlop/s
 Time= 0.033 msec
 Size= 132000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 32 1000000 -d 1000 64
Number of Bins:
        32
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.78 GFlop/s
 Time= 0.216 msec
 Size= 1032000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 1000 -d 1000 64
Number of Bins:
        64
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.06 GFlop/s
 Time= 0.021 msec
 Size= 65000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 10000 -d 1000 64
Number of Bins:
        64
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.60 GFlop/s
 Time= 0.021 msec
 Size= 74000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 100000 -d 1000 64
Number of Bins:
        64
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.77 GFlop/s
 Time= 0.034 msec
 Size= 164000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 64 1000000 -d 1000 64
Number of Bins:
        64
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.66 GFlop/s
 Time= 0.228 msec
 Size= 1064000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 1000 -d 1000 64
Number of Bins:
        128
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.15 GFlop/s
 Time= 0.041 msec
 Size= 129000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 10000 -d 1000 64
Number of Bins:
        128
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 3.73 GFlop/s
 Time= 0.037 msec
 Size= 138000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 100000 -d 1000 64
Number of Bins:
        128
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.85 GFlop/s
 Time= 0.047 msec
 Size= 228000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 128 1000000 -d 1000 64
Number of Bins:
        128
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 4.58 GFlop/s
 Time= 0.246 msec
 Size= 1128000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```

```
C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 1000 -d 1000 64
Number of Bins:
        256
Input Vector Size:
        1000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 5.23 GFlop/s
 Time= 0.049 msec
 Size= 257000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 10000 -d 1000 64
Number of Bins:
        256
Input Vector Size:
        10000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 5.64 GFlop/s
 Time= 0.047 msec
 Size= 266000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 100000 -d 1000 64
Number of Bins:
        256
Input Vector Size:
        100000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 5.35 GFlop/s
 Time= 0.066 msec
 Size= 356000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal

C:\Users\Dan\source\repos\histogram\Debug>histogram.exe -i 256 1000000 -d 1000 64
Number of Bins:
        256
Input Vector Size:
        1000000

Grid Dimensions:
        - 1000
Block Dimensions:
        - 64


GPU Done
Performance= 5.59 GFlop/s
 Time= 0.225 msec
 Size= 1256000 Ops
 WorkgroupSize= 64 threads/block
Running Checks:
        - All values counted
        - Histograms Equal
```