

# Module 7 Lab

## Histogram

*GPU Teaching Kit – Accelerated Computing*

### OBJECTIVE

The purpose of this lab is to implement an efficient histogramming algorithm for an input array of integers within a given range. Each integer will map into a single bin, so the values will range from 0 to (NUM\_BINS - 1). The histogram bins will use unsigned 32-bit counters that must be saturated at 127 (i.e. no roll back to 0 allowed). The input length can be assumed to be less than  $2^{32}$ . NUM\_BINS is fixed at 4096 for this lab.

This can be split into two kernels: one that does a histogram without saturation, and a final kernel that cleans up the bins if they are too large. These two stages can also be combined into a single kernel.

### PREREQUISITES

Before starting this lab, make sure that:

- You have completed all of the Module 7 lecture videos and materials.

### INSTRUCTIONS

Edit the code in the code tab to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

## LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the [Bitbucket repository](#). A description on how to use the [CMake](#) tool in along with how to build the labs for local development found in the [README](#) document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./Histogram_Template -e <expected.raw> \
-i <input.raw> -o <output.raw> -t integral_vector
```

where <expected.raw> is the expected output, <input.raw> is the input dataset, and <output.raw> is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

## QUESTIONS

- (1) Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance.

ANSWER: **The main optimization in the sample solution is for each threadblock to have a privatized histogram in shared memory. Atomic operations on shared memory are much faster than global memory.**

- (2) Were there any difficulties you had with completing the optimization correctly.

ANSWER: **Using shared memory requires thread synchronization.**

- (3) Which optimizations gave the most benefit.

ANSWER: **Using shared memory privatization provides significant performance improvement.**

- (4) For the histogram kernel, how many global memory reads are being performed by your kernel? explain.

ANSWER: **One read per input element.**

- (5) For the histogram kernel, how many global memory writes are being performed by your kernel? explain.

ANSWER: **Each threadblock does an atomic add into every global memory bin.**

- (6) For the histogram kernel, how many atomic operations are being performed by your kernel? explain.

ANSWER: **One atomic operation per input element into shared memory, then NUM\_BINS atomic operation per thread block to accumulate the results into the global bins.**

- (7) For the histogram kernel, what contentions would you expect if every element in the array has the same value?

ANSWER: **If every pixel in the image is the same value, every thread in a thread block will contend when they increment in shared memory.**

- (8) For the histogram kernel, what contentions would you expect if every element in the input array has a random value?

ANSWER: **We would expect little contention since there are 512 threads doing atomic operations over a random 4096 bins every iteration.**

## CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code in the sections demarcated with //@. Students expected the other code unchanged. The tutorial page describes the functionality of the wb\* methods.

```

1  #include <wb.h>
2
3  #define NUM_BINS 4096
4
5  #define CUDA_CHECK(ans)
6      { gpuAssert((ans), __FILE__, __LINE__); }
7  inline void gpuAssert(cudaError_t code, const char *file, int line,
8                      bool abort = true) {
9      if (code != cudaSuccess) {
10         fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
11             file, line);
12         if (abort)
13             exit(code);
14     }
15 }
16
17 int main(int argc, char *argv[]) {
18     wbArg_t args;
19     int inputLength;
20     unsigned int *hostInput;
21     unsigned int *hostBins;
22     unsigned int *deviceInput;
23     unsigned int *deviceBins;
24
25     args = wbArg_read(argc, argv);
26
27     wbTime_start(Generic, "Importing data and creating memory on host");
28     hostInput = (unsigned int *)wbImport(wbArg_getInputFile(args, 0),
29                                         &inputLength, "Integer");
30     hostBins = (unsigned int *)malloc(NUM_BINS * sizeof(unsigned int));
31     wbTime_stop(Generic, "Importing data and creating memory on host");
32
33     wbLog	TRACE, "The input length is ", inputLength);

```

```

34     wbLog(TRACE, "The number of bins is ", NUM_BINS);
35
36     wbTime_start(GPU, "Allocating GPU memory.");
37     ///@@ Allocate GPU memory here
38     CUDA_CHECK(cudaDeviceSynchronize());
39     wbTime_stop(GPU, "Allocating GPU memory.");
40
41     wbTime_start(GPU, "Copying input memory to the GPU.");
42     ///@@ Copy memory to the GPU here
43     CUDA_CHECK(cudaDeviceSynchronize());
44     wbTime_stop(GPU, "Copying input memory to the GPU.");
45
46     // Launch kernel
47     // -----
48     wbLog(TRACE, "Launching kernel");
49     wbTime_start(Compute, "Performing CUDA computation");
50     ///@@ Perform kernel computation here
51     wbTime_stop(Compute, "Performing CUDA computation");
52
53     wbTime_start(Copy, "Copying output memory to the CPU");
54     ///@@ Copy the GPU memory back to the CPU here
55     CUDA_CHECK(cudaDeviceSynchronize());
56     wbTime_stop(Copy, "Copying output memory to the CPU");
57
58     wbTime_start(GPU, "Freeing GPU Memory");
59     ///@@ Free the GPU memory here
60     wbTime_stop(GPU, "Freeing GPU Memory");
61
62     // Verify correctness
63     // -----
64     wbSolution(args, hostBins, NUM_BINS);
65
66     free(hostBins);
67     free(hostInput);
68     return 0;
69 }

```

## CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```

1  #include <wb.h>
2
3  #define NUM_BINS 4096
4
5  #define CUDA_CHECK(ans)
6      { gpuAssert((ans), __FILE__, __LINE__); }
7  inline void gpuAssert(cudaError_t code, const char *file, int line,
8                      bool abort = true) {
9      if (code != cudaSuccess) {

```

```

10     fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
11               file, line);
12     if (abort)
13         exit(code);
14 }
15 }
16
17 __global__ void histogram_kernel(unsigned int *input, unsigned int *bins,
18                                 unsigned int num_elements,
19                                 unsigned int num_bins) {
20
21     unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
22
23     // Privatized bins
24     extern __shared__ unsigned int bins_s[];
25     for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
26          binIdx += blockDim.x) {
27         bins_s[binIdx] = 0;
28     }
29     __syncthreads();
30
31     // Histogram
32     for (unsigned int i = tid; i < num_elements;
33          i += blockDim.x * gridDim.x) {
34         atomicAdd(&(bins_s[input[i]]), 1);
35     }
36     __syncthreads();
37
38     // Commit to global memory
39     for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
40          binIdx += blockDim.x) {
41         atomicAdd(&(bins[binIdx]), bins_s[binIdx]);
42     }
43 }
44
45 __global__ void convert_kernel(unsigned int *bins, unsigned int num_bins) {
46
47     unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
48
49     if (tid < num_bins) {
50         bins[tid] = min(bins[tid], 127);
51     }
52 }
53
54 void histogram(unsigned int *input, unsigned int *bins,
55               unsigned int num_elements, unsigned int num_bins) {
56
57     // zero out bins
58     CUDA_CHECK(cudaMemset(bins, 0, num_bins * sizeof(unsigned int)));
59     // Launch histogram kernel on the bins
60     {
61         dim3 blockDim(512), gridDim(30);
62         histogram_kernel<<<gridDim, blockDim,

```

```

63         num_bins * sizeof(unsigned int)>>>(
64         input, bins, num_elements, num_bins);
65     CUDA_CHECK(cudaGetLastError());
66     CUDA_CHECK(cudaDeviceSynchronize());
67 }
68
69 // Make sure bin values are not too large
70 {
71     dim3 blockDim(512);
72     dim3 gridDim((num_bins + blockDim.x - 1) / blockDim.x);
73     convert_kernel<<<gridDim, blockDim>>>(bins, num_bins);
74     CUDA_CHECK(cudaGetLastError());
75     CUDA_CHECK(cudaDeviceSynchronize());
76 }
77 }
78
79 int main(int argc, char *argv[]) {
80     wbArg_t args;
81     int inputLength;
82     unsigned int *hostInput;
83     unsigned int *hostBins;
84     unsigned int *deviceInput;
85     unsigned int *deviceBins;
86
87     args = wbArg_read(argc, argv);
88
89     wbTime_start(Generic, "Importing data and creating memory on host");
90     hostInput = (unsigned int *)wbImport(wbArg_getInputFile(args, 0),
91                                         &inputLength, "Integer");
92     hostBins = (unsigned int *)malloc(NUM_BINS * sizeof(unsigned int));
93     wbTime_stop(Generic, "Importing data and creating memory on host");
94
95     wbLog	TRACE, "The input length is ", inputLength);
96     wbLog	TRACE, "The number of bins is ", NUM_BINS);
97
98     wbTime_start(GPU, "Allocating GPU memory.");
99     ///@@ Allocate GPU memory here
100    CUDA_CHECK(cudaMalloc((void **)&deviceInput,
101                          inputLength * sizeof(unsigned int)));
102    CUDA_CHECK(
103        cudaMalloc((void **)&deviceBins, NUM_BINS * sizeof(unsigned int)));
104    CUDA_CHECK(cudaDeviceSynchronize());
105    wbTime_stop(GPU, "Allocating GPU memory.");
106
107    wbTime_start(GPU, "Copying input memory to the GPU.");
108    ///@@ Copy memory to the GPU here
109    CUDA_CHECK(cudaMemcpy(deviceInput, hostInput,
110                          inputLength * sizeof(unsigned int),
111                          cudaMemcpyHostToDevice));
112    CUDA_CHECK(cudaDeviceSynchronize());
113    wbTime_stop(GPU, "Copying input memory to the GPU.");
114
115    // Launch kernel

```

```

116 // -----
117 wbLog	TRACE, "Launching kernel");
118 wbTime_start(Compute, "Performing CUDA computation");
119
120 histogram(deviceInput, deviceBins, inputLength, NUM_BINS);
121 wbTime_stop(Compute, "Performing CUDA computation");
122
123 wbTime_start(Copy, "Copying output memory to the CPU");
124 //@@ Copy the GPU memory back to the CPU here
125 CUDA_CHECK(cudaMemcpy(hostBins, deviceBins,
126                       NUM_BINS * sizeof(unsigned int),
127                       cudaMemcpyDeviceToHost));
128 CUDA_CHECK(cudaDeviceSynchronize());
129 wbTime_stop(Copy, "Copying output memory to the CPU");
130
131 wbTime_start(GPU, "Freeing GPU Memory");
132 //@@ Free the GPU memory here
133 CUDA_CHECK(cudaFree(deviceInput));
134 CUDA_CHECK(cudaFree(deviceBins));
135 wbTime_stop(GPU, "Freeing GPU Memory");
136
137 // Verify correctness
138 // -----
139 wbSolution(args, hostBins, NUM_BINS);
140
141 free(hostBins);
142 free(hostInput);
143 return 0;
144 }

```