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 privitized 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 syncrhonization.**

(3) Which optimizations gave the most benifit.

ANSWER: **Using shared memory privitization 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 is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>

#define NUM_BINS 4096

#define CUDA_CHECK(ans)                                                   \
  { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line,
                      bool abort = true) {
  if (code != cudaSuccess) {
    fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
            file, line);
    if (abort)
      exit(code);
  }
}

int main(int argc, char *argv[]) {
  wbArg_t args;
  int inputLength;
  unsigned int *hostInput;
  unsigned int *hostBins;
  unsigned int *deviceInput;
  unsigned int *deviceBins;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput = (unsigned int *)wbImport(wbArg_getInputFile(args, 0),
                                       &inputLength, "Integer");
  hostBins = (unsigned int *)malloc(NUM_BINS * sizeof(unsigned int));
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);
```

```
  wbLog(TRACE, "The number of bins is ", NUM_BINS);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
  CUDA_CHECK(cudaDeviceSynchronize());
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  CUDA_CHECK(cudaDeviceSynchronize());
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  // Launch kernel
  // ----------------------------------------------------------
  wbLog(TRACE, "Launching kernel");
  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Perform kernel computation here
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here
  CUDA_CHECK(cudaDeviceSynchronize());
  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here
  wbTime_stop(GPU, "Freeing GPU Memory");

  // Verify correctness
  // -----------------------------------------------------
  wbSolution(args, hostBins, NUM_BINS);

  free(hostBins);
  free(hostInput);
  return 0;
}
```

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.

```
#include <wb.h>

#define NUM_BINS 4096

#define CUDA_CHECK(ans)                                                   \
  { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line,
                      bool abort = true) {
  if (code != cudaSuccess) {
```

```
    fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
            file, line);
    if (abort)
      exit(code);
  }
}

__global__ void histogram_kernel(unsigned int *input, unsigned int *bins,
                                 unsigned int num_elements,
                                 unsigned int num_bins) {

  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Privatized bins
  extern __shared__ unsigned int bins_s[];
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
       binIdx += blockDim.x) {
    bins_s[binIdx] = 0;
  }
  __syncthreads();

  // Histogram
  for (unsigned int i = tid; i < num_elements;
       i += blockDim.x * gridDim.x) {
    atomicAdd(&(bins_s[input[i]]), 1);
  }
  __syncthreads();

  // Commit to global memory
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
       binIdx += blockDim.x) {
    atomicAdd(&(bins[binIdx]), bins_s[binIdx]);
  }
}

__global__ void convert_kernel(unsigned int *bins, unsigned int num_bins) {

  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (tid < num_bins) {
    bins[tid] = min(bins[tid], 127);
  }
}

void histogram(unsigned int *input, unsigned int *bins,
               unsigned int num_elements, unsigned int num_bins) {

  // zero out bins
  CUDA_CHECK(cudaMemset(bins, 0, num_bins * sizeof(unsigned int)));
  // Launch histogram kernel on the bins
  {
    dim3 blockDim(512), gridDim(30);
    histogram_kernel<<<gridDim, blockDim,
```

```
                        num_bins * sizeof(unsigned int)>>>(
        input, bins, num_elements, num_bins);
    CUDA_CHECK(cudaGetLastError());
    CUDA_CHECK(cudaDeviceSynchronize());
  }

  // Make sure bin values are not too large
  {
    dim3 blockDim(512);
    dim3 gridDim((num_bins + blockDim.x - 1) / blockDim.x);
    convert_kernel<<<gridDim, blockDim>>>(bins, num_bins);
    CUDA_CHECK(cudaGetLastError());
    CUDA_CHECK(cudaDeviceSynchronize());
  }
}

int main(int argc, char *argv[]) {
  wbArg_t args;
  int inputLength;
  unsigned int *hostInput;
  unsigned int *hostBins;
  unsigned int *deviceInput;
  unsigned int *deviceBins;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput = (unsigned int *)wbImport(wbArg_getInputFile(args, 0),
                                       &inputLength, "Integer");
  hostBins = (unsigned int *)malloc(NUM_BINS * sizeof(unsigned int));
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);
  wbLog(TRACE, "The number of bins is ", NUM_BINS);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
  CUDA_CHECK(cudaMalloc((void **)&deviceInput,
                        inputLength * sizeof(unsigned int)));
  CUDA_CHECK(
      cudaMalloc((void **)&deviceBins, NUM_BINS * sizeof(unsigned int)));
  CUDA_CHECK(cudaDeviceSynchronize());
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  CUDA_CHECK(cudaMemcpy(deviceInput, hostInput,
                        inputLength * sizeof(unsigned int),
                        cudaMemcpyHostToDevice));
  CUDA_CHECK(cudaDeviceSynchronize());
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  // Launch kernel
```

```
    // -----------------------------------------------------------
    wbLog(TRACE, "Launching kernel");
    wbTime_start(Compute, "Performing CUDA computation");

    histogram(deviceInput, deviceBins, inputLength, NUM_BINS);
    wbTime_stop(Compute, "Performing CUDA computation");

    wbTime_start(Copy, "Copying output memory to the CPU");
    //@@ Copy the GPU memory back to the CPU here
    CUDA_CHECK(cudaMemcpy(hostBins, deviceBins,
                          NUM_BINS * sizeof(unsigned int),
                          cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaDeviceSynchronize());
    wbTime_stop(Copy, "Copying output memory to the CPU");

    wbTime_start(GPU, "Freeing GPU Memory");
    //@@ Free the GPU memory here
    CUDA_CHECK(cudaFree(deviceInput));
    CUDA_CHECK(cudaFree(deviceBins));
    wbTime_stop(GPU, "Freeing GPU Memory");

    // Verify correctness
    // -----------------------------------------------------
    wbSolution(args, hostBins, NUM_BINS);

    free(hostBins);
    free(hostInput);
    return 0;
  }
```