Module 7 Lab

Text Histogram

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to implement an efficient histogram algorithm for an input array of ASCII characters. There are 128 ASCII characters and each character will map into its own bin for a fixed total of 128 bins. The histogram bins will be unsigned 32-bit counters that do not saturate. Use the approach of creating a privitized histogram in shared memory for each thread block, then atomically modifying the global histogram.

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.txt> -o <output.raw> -t integral_vector
```

where <expected.raw> is the expected output, <input.txt> 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) Were there any difficulties you had with completing the optimization correctly.

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) Which optimizations gave the most benifit.

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

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

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

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

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

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

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

(6) Most text files will consist of only letters, numberm and whitespace characters. These 95 characters make up fall between ASCII numbers 32 - 126. What can we say about atomic access contention if more than 95 threads are simultaneously trying to atomically increment a private histogram?

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.**

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");
```

```
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 128
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(const char *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[(unsigned int)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  void histogram(const char *input, unsigned int *bins,
46                 unsigned int num_elements, unsigned int num_bins) {
47
48    // zero out bins
49    CUDA_CHECK(cudaMemset(bins, 0, num_bins * sizeof(unsigned int)));
50    // Launch histogram kernel on the bins
51    {
52      dim3 blockDim(256), gridDim(30);
53      histogram_kernel<<<gridDim, blockDim,
54                         num_bins * sizeof(unsigned int)>>>(
55          input, bins, num_elements, num_bins);
56      CUDA_CHECK(cudaGetLastError());
57      CUDA_CHECK(cudaDeviceSynchronize());
58    }
59  }
60
61  int main(int argc, char *argv[]) {
62    wbArg_t args;
63    int inputLength;
64    char *hostInput;
65    unsigned int *hostBins;
66    char *deviceInput;
67    unsigned int *deviceBins;
68
69    args = wbArg_read(argc, argv);
70
71    wbTime_start(Generic, "Importing data and creating memory on host");
72    hostInput =
73        (char *)wbImport(wbArg_getInputFile(args, 0), &inputLength, "Text");
74    hostBins = (unsigned int *)malloc(NUM_BINS * sizeof(unsigned int));
75    wbTime_stop(Generic, "Importing data and creating memory on host");
76
77    wbLog(TRACE, "The input length is ", inputLength);
78    wbLog(TRACE, "The number of bins is ", NUM_BINS);
79
80    wbTime_start(GPU, "Allocating GPU memory.");
81    //@@ Allocate GPU memory here
82    CUDA_CHECK(cudaMalloc((void **)&deviceInput, inputLength));
```

```
83      CUDA_CHECK(
84          cudaMalloc((void **)&deviceBins, NUM_BINS * sizeof(unsigned int)));
85      CUDA_CHECK(cudaDeviceSynchronize());
86      wbTime_stop(GPU, "Allocating GPU memory.");
87
88      wbTime_start(GPU, "Copying input memory to the GPU.");
89      //@@ Copy memory to the GPU here
90      CUDA_CHECK(cudaMemcpy(deviceInput, hostInput, inputLength,
91                            cudaMemcpyHostToDevice));
92      CUDA_CHECK(cudaDeviceSynchronize());
93      wbTime_stop(GPU, "Copying input memory to the GPU.");
94
95      // Launch kernel
96      // ----------------------------------------------------------
97      wbLog(TRACE, "Launching kernel");
98      wbTime_start(Compute, "Performing CUDA computation");
99      // @@ Insert code here
100     histogram(deviceInput, deviceBins, inputLength, NUM_BINS);
101     wbTime_stop(Compute, "Performing CUDA computation");
102
103     wbTime_start(Copy, "Copying output memory to the CPU");
104     //@@ Copy the GPU memory back to the CPU here
105     CUDA_CHECK(cudaMemcpy(hostBins, deviceBins,
106                           NUM_BINS * sizeof(unsigned int),
107                           cudaMemcpyDeviceToHost));
108     CUDA_CHECK(cudaDeviceSynchronize());
109     wbTime_stop(Copy, "Copying output memory to the CPU");
110
111     wbTime_start(GPU, "Freeing GPU Memory");
112     //@@ Free the GPU memory here
113     CUDA_CHECK(cudaFree(deviceInput));
114     CUDA_CHECK(cudaFree(deviceBins));
115     wbTime_stop(GPU, "Freeing GPU Memory");
116
117     // Verify correctness
118     // -----------------------------------------------------
119     wbSolution(args, hostBins, NUM_BINS);
120
121     free(hostBins);
122     free(hostInput);
123     return 0;
124 }
```