Module 3 Lab

CUDA Image Blur

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to implement an efficient image blurring algorithm for an input image. Like the image convolution Lab, the image is represented as `RGB float` values. You will operate directly on the RGB float values and use a 3x3 Box Filter to blur the original image to produce the blurred image.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed all Module 3 lecture videos

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:

```
./ImageBlur_Template -e <expected.ppm> -i <input.ppm> \
  -o <output.ppm> -t image
```

where `<expected.ppm>` is the expected output, `<input.ppm>` is the input dataset, and `<output.ppm>` 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) How many floating operations are being performed in your color conversion kernel? EXPLAIN.

ANSWER: **Implemented correctly, one needs to perform one floating point addition and one division for each pixel on average**

(2) How many global memory reads are being performed by your kernel? EXPLAIN.

ANSWER: **There is one global memory read per pixel. Since the image is monochromatic this corresponds to one floating point read.**

(3) How many global memory writes are being performed by your kernel? EXPLAIN.

ANSWER: **There is one global memory write per pixel. Since the image is monochromatic this corresponds to one floating point read.**

(4) Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **One can represent the image as unsigned characters. This means that each pixel occupies one byte compared to 4 required for single precision floating point.**

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.

```
1  #include <wb.h>
2
3  #define wbCheck(stmt)                                     \
4    do {                                                    \
```

```
    cudaError_t err = stmt;                                                 \
    if (err != cudaSuccess) {                                               \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                           \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));        \
      return -1;                                                            \
    }                                                                       \
  } while (0)

#define BLUR_SIZE 5

//@@ INSERT CODE HERE

int main(int argc, char *argv[]) {
  wbArg_t args;
  int imageWidth;
  int imageHeight;
  char *inputImageFile;
  wbImage_t inputImage;
  wbImage_t outputImage;
  float *hostInputImageData;
  float *hostOutputImageData;
  float *deviceInputImageData;
  float *deviceOutputImageData;

  args = wbArg_read(argc, argv); /* parse the input arguments */

  inputImageFile = wbArg_getInputFile(args, 0);

  inputImage = wbImport(inputImageFile);

  // The input image is in grayscale, so the number of channels
  // is 1
  imageWidth  = wbImage_getWidth(inputImage);
  imageHeight = wbImage_getHeight(inputImage);

  // Since the image is monochromatic, it only contains only one channel
  outputImage = wbImage_new(imageWidth, imageHeight, 1);

  hostInputImageData  = wbImage_getData(inputImage);
  hostOutputImageData = wbImage_getData(outputImage);

  wbTime_start(GPU, "Doing GPU Computation (memory + compute)");

  wbTime_start(GPU, "Doing GPU memory allocation");
  cudaMalloc((void **)&deviceInputImageData,
             imageWidth * imageHeight * sizeof(float));
  cudaMalloc((void **)&deviceOutputImageData,
             imageWidth * imageHeight * sizeof(float));
  wbTime_stop(GPU, "Doing GPU memory allocation");

  wbTime_start(Copy, "Copying data to the GPU");
  cudaMemcpy(deviceInputImageData, hostInputImageData,
             imageWidth * imageHeight * sizeof(float),
```

```
58              cudaMemcpyHostToDevice);
59    wbTime_stop(Copy, "Copying data to the GPU");
60
61    ////////////////////////////////////////////////////
62    wbTime_start(Compute, "Doing the computation on the GPU");
63
64    wbTime_stop(Compute, "Doing the computation on the GPU");
65
66    ////////////////////////////////////////////////////
67    wbTime_start(Copy, "Copying data from the GPU");
68    cudaMemcpy(hostOutputImageData, deviceOutputImageData,
69               imageWidth * imageHeight * sizeof(float),
70               cudaMemcpyDeviceToHost);
71    wbTime_stop(Copy, "Copying data from the GPU");
72
73    wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");
74
75    wbSolution(args, outputImage);
76
77    cudaFree(deviceInputImageData);
78    cudaFree(deviceOutputImageData);
79
80    wbImage_delete(outputImage);
81    wbImage_delete(inputImage);
82
83    return 0;
84  }
```

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 wbCheck(stmt)                                                     \
4     do {                                                                    \
5       cudaError_t err = stmt;                                               \
6       if (err != cudaSuccess) {                                             \
7         wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
8         wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
9         return -1;                                                          \
10      }                                                                     \
11    } while (0)
12
13  #define BLUR_SIZE 5
14
15  //@@ INSERT CODE HERE
16  #define TILE_WIDTH 16
17  __global__ void blurKernel(float *out, float *in, int width, int height) {
18    int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
19    int row = blockIdx.y * blockDim.y + threadIdx.y;
20
21    if (col < width && row < height) {
22      int pixVal = 0;
23      int pixels = 0;
24
25      // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
26      for (int blurrow = -BLUR_SIZE; blurrow < BLUR_SIZE + 1; ++blurrow) {
27        for (int blurcol = -BLUR_SIZE; blurcol < BLUR_SIZE + 1; ++blurcol) {
28
29          int currow = row + blurrow;
30          int curcol = col + blurcol;
31          // Verify we have a valid image pixel
32          if (currow > -1 && currow < height && curcol > -1 &&
33              curcol < width) {
34            pixVal += in[currow * width + curcol];
35            pixels++; // Keep track of number of pixels in the avg
36          }
37        }
38      }
39
40      // Write our new pixel value out
41      out[row * width + col] = (unsigned char)(pixVal / pixels);
42    }
43  }
44
45  int main(int argc, char *argv[]) {
46    wbArg_t args;
47    int imageWidth;
48    int imageHeight;
49    char *inputImageFile;
50    wbImage_t inputImage;
51    wbImage_t outputImage;
52    float *hostInputImageData;
53    float *hostOutputImageData;
54    float *deviceInputImageData;
55    float *deviceOutputImageData;
56
57    args = wbArg_read(argc, argv); /* parse the input arguments */
58
59    inputImageFile = wbArg_getInputFile(args, 0);
60
61    inputImage = wbImport(inputImageFile);
62
63    // The input image is in grayscale, so the number of channels
64    // is 1
65    imageWidth  = wbImage_getWidth(inputImage);
66    imageHeight = wbImage_getHeight(inputImage);
67
68    // Since the image is monochromatic, it only contains only one channel
69    outputImage = wbImage_new(imageWidth, imageHeight, 1);
70
71    hostInputImageData  = wbImage_getData(inputImage);
```

```
72    hostOutputImageData = wbImage_getData(outputImage);
73
74    wbTime_start(GPU, "Doing GPU Computation (memory + compute)");
75
76    wbTime_start(GPU, "Doing GPU memory allocation");
77    cudaMalloc((void **)&deviceInputImageData,
78               imageWidth * imageHeight * sizeof(float));
79    cudaMalloc((void **)&deviceOutputImageData,
80               imageWidth * imageHeight * sizeof(float));
81    wbTime_stop(GPU, "Doing GPU memory allocation");
82
83    wbTime_start(Copy, "Copying data to the GPU");
84    cudaMemcpy(deviceInputImageData, hostInputImageData,
85               imageWidth * imageHeight * sizeof(float),
86               cudaMemcpyHostToDevice);
87    wbTime_stop(Copy, "Copying data to the GPU");
88
89    ///////////////////////////////////////////////////////
90    wbTime_start(Compute, "Doing the computation on the GPU");
91    //@@ INSERT CODE HERE
92    dim3 dimGrid(ceil((float)imageWidth / TILE_WIDTH),
93                 ceil((float)imageHeight / TILE_WIDTH));
94    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
95    blurKernel<<<dimGrid, dimBlock>>>(deviceOutputImageData,
96                                      deviceInputImageData, imageWidth,
97                                      imageHeight);
98    wbTime_stop(Compute, "Doing the computation on the GPU");
99
100   ///////////////////////////////////////////////////////
101   wbTime_start(Copy, "Copying data from the GPU");
102   cudaMemcpy(hostOutputImageData, deviceOutputImageData,
103              imageWidth * imageHeight * sizeof(float),
104              cudaMemcpyDeviceToHost);
105   wbTime_stop(Copy, "Copying data from the GPU");
106
107   wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");
108
109   wbSolution(args, outputImage);
110
111   cudaFree(deviceInputImageData);
112   cudaFree(deviceOutputImageData);
113
114   wbImage_delete(outputImage);
115   wbImage_delete(inputImage);
116
117   return 0;
118 }
```