Module 8 Lab

Convolution

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The lab's objective is to implement a tiled image convolution using both shared and constant memory. We will have a constant 5x5 convolution mask, but will have arbitrarily sized image (assume the image dimensions are greater than 5x5 for this Lab).

To use the constant memory for the convolution mask, you can first transfer the mask data to the device. Consider the case where the pointer to the device array for the mask is named M. You can use `const float * __restrict__ M` as one of the parameters during your kernel launch. This informs the compiler that the contents of the mask array are constants and will only be accessed through pointer variable `M`. This will enable the compiler to place the data into constant memory and allow the SM hardware to aggressively cache the mask data at runtime.

Convolution is used in many fields, such as image processing for image filtering. A standard image convolution formula for a 5x5 convolution filter `M` with an Image `I` is:

$$P_{i,j,c} = \sum_{x=-2}^{2} \sum_{y=-2}^{2} I_{i+x,j+y,c} * M_{x,y}$$

where $P_{i,j,c}$ is the output pixel at position `i,j` in channel `c`, $I_{i,j,c}$ is the input pixel at `i,j` in channel `c` (the number of channels will always be 3 for this MP corresponding to the RGB values), and $M_{x,y}$ is the mask at position `x,y`.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed all Module 8 Lecture videos

INPUT DATA

The input is an interleaved image of `height x width x channels`. By interleaved, we mean that the the element `I[y][x]` contains three values representing the RGB channels. This means that to index a particular element's value, you will have to do something like:

```
index = (yIndex*width + xIndex)*channels + channelIndex;
```

For this assignment, the channel index is 0 for R, 1 for G, and 2 for B. So, to access the G value of `I[y][x]`, you should use the linearized expression `I[(yIndex*width+xIndex)*channels + 1]`.

For simplicity, you can assume that `channels` is always set to 3.

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
- implement the tiled 2D convolution kernel with adjustments for channels
- use shared memory to reduce the number of global accesses, handle the boundary conditions in when loading input list elements into the shared memory

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

PSEUDO CODE

A sequential pseudo code would look something like this:

```
maskWidth := 5
maskRadius := maskWidth/2 # this is integer division, so the result is 2
for i from 0 to height do
  for j from 0 to width do
    for k from 0 to channels
      accum := 0
      for y from -maskRadius to maskRadius do
        for x from -maskRadius to maskRadius do
          xOffset := j + x
          yOffset := i + y
          if xOffset >= 0 && xOffset < width &&
             yOffset >= 0 && yOffset < height then
            imagePixel := I[(yOffset * width + xOffset) * channels + k]
```

```
            maskValue := K[(y+maskRadius)*maskWidth+x+maskRadius]
            accum += imagePixel * maskValue
          end
        end
      end
      # pixels are in the range of 0 to 1
      P[(i * width + j)*channels + k] = clamp(accum, 0, 1)
    end
  end
end
```

where `clamp` is defined as

```
def clamp(x, lower, upper)
  return min(max(x, lower), upper)
end
```

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:

```
./Convolution_Template -e <expected.ppm> \
    -i <input0.ppm>,<input1.raw> -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.

The images are stored in PPM (`P6`) format, this means that you can (if you want) create your own input images. The easiest way to create image is via external tools such as `bmptoppm`. The masks are stored in a CSV format. Since the input is small, it is best to edit it by hand.

QUESTIONS

(1) Name 3 applications of convolution.

ANSWER: **Image processing, signal+processing, stencil operations.**

(2) How many floating operations are being performed in your convolution kernel? explain.

ANSWER: **Each output location requires KERNEL_WIDTH * KERNEL_WIDTH, and there are width*height*channels outputs**

(3) How many global memory reads are being performed by your kernel? explain.

ANSWER: **Each thread block reads** $(\text{TILE\_WIDTH} + \text{MASK\_WIDTH} - 1)^2$**. There are** $\text{width}/\text{TILE\_WIDTH} * \text{height}/\text{TILE\_WIDTH}$ **thread blocks, for a total of** $(\text{TILE\_WIDTH} + \text{MASK\_WIDTH} - 1)^2 * \text{height} * \text{width}/\text{TILE\_WIDTH}^2$**. This includes halo elements, which are not read. Subtract** $((\text{height} + \text{MASK\_WIDTH}/2)(\text{width} + \text{MASK\_WIDTH}/2) - \text{height} * \text{width})$

(4) How many global memory writes are being performed by your kernel? explain.

ANSWER: **One write per output location, or width*height*channels.**

(5) What is the minimum, maximum, and average number of `real` operations that a thread will perform? `Real` operations are those that directly contribute to the final output value.

ANSWER: **All threads do the same amount of convolution work, which is KERNEL_WIDTH * KERNEL_WIDTH**

(6) What is the measured floating-point computation rate for the CPU and GPU kernels in this application? How do they each scale with the size of the input?

ANSWER: **This will depend on the target machine. The CPU implementation's performance will drop with input size as the problem grows too large for the cache. The GPU implementation's FLOP rate should remain relatively constant across input sizes.**

(7) How much time is spent as an overhead cost for using the GPU for computation? Consider all code executed within your host function with the exception of the kernel itself, as overhead. How does the overhead scale with the size of the input?

ANSWER: **This answer may depend on the target machine. The amount of data transfer and work in convolution scale at the same rate, so for large inputs they should grow at the same rate.?**

(8) What do you think happens as you increase the mask size (say to 1024) while you set the block dimensions to 16x16? What do you end up spending most of your time doing? Does that put other constraints on the way you'd write your algorithm (think of the shared/constant memory size)?

ANSWER: **If the convolution kernel is too large, the required shared memory will be too large to have enough parallelism within a thread block. For large kernels most of the time will be spent reading overlapping kernel halos from global into shared memory, which defeats the purpose of using this algorithm.**

(9) Do you have to have a separate output memory buffer? Put it in another way, why can't you perform the convolution in place?

ANSWER: **By doing convolution in place, you may change a value that another thread needs as input, thereby causing an incorrect answer to be produced.**

(10) What is the identity mask?

ANSWER: **All zeros, except a 1 in the center.**

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 wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      return -1;                                                          \
    }                                                                     \
  } while (0)

#define Mask_width 5
#define Mask_radius Mask_width / 2
#define TILE_WIDTH 16
#define w (TILE_WIDTH + Mask_width - 1)
#define clamp(x) (min(max((x), 0.0), 1.0))

//@@ INSERT CODE HERE

int main(int argc, char *argv[]) {
  wbArg_t arg;
  int maskRows;
  int maskColumns;
  int imageChannels;
  int imageWidth;
  int imageHeight;
  char *inputImageFile;
  char *inputMaskFile;
  wbImage_t inputImage;
  wbImage_t outputImage;
  float *hostInputImageData;
  float *hostOutputImageData;
  float *hostMaskData;
  float *deviceInputImageData;
  float *deviceOutputImageData;
  float *deviceMaskData;

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

  inputImageFile = wbArg_getInputFile(arg, 0);
  inputMaskFile  = wbArg_getInputFile(arg, 1);

  inputImage   = wbImport(inputImageFile);
  hostMaskData = (float *)wbImport(inputMaskFile, &maskRows, &maskColumns);

```

```
46    assert(maskRows == 5);    /* mask height is fixed to 5 in this mp */
47    assert(maskColumns == 5); /* mask width is fixed to 5 in this mp */
48
49    imageWidth    = wbImage_getWidth(inputImage);
50    imageHeight   = wbImage_getHeight(inputImage);
51    imageChannels = wbImage_getChannels(inputImage);
52
53    outputImage = wbImage_new(imageWidth, imageHeight, imageChannels);
54
55    hostInputImageData  = wbImage_getData(inputImage);
56    hostOutputImageData = wbImage_getData(outputImage);
57
58    wbTime_start(GPU, "Doing GPU Computation (memory + compute)");
59
60    wbTime_start(GPU, "Doing GPU memory allocation");
61    //@@ INSERT CODE HERE
62    wbTime_stop(GPU, "Doing GPU memory allocation");
63
64    wbTime_start(Copy, "Copying data to the GPU");
65    //@@ INSERT CODE HERE
66    wbTime_stop(Copy, "Copying data to the GPU");
67
68    wbTime_start(Compute, "Doing the computation on the GPU");
69    //@@ INSERT CODE HERE
70    convolution<<<dimGrid, dimBlock>>>(deviceInputImageData, deviceMaskData,
71                                       deviceOutputImageData, imageChannels,
72                                       imageWidth, imageHeight);
73    wbTime_stop(Compute, "Doing the computation on the GPU");
74
75    wbTime_start(Copy, "Copying data from the GPU");
76    //@@ INSERT CODE HERE
77    cudaMemcpy(hostOutputImageData, deviceOutputImageData,
78               imageWidth * imageHeight * imageChannels * sizeof(float),
79               cudaMemcpyDeviceToHost);
80    wbTime_stop(Copy, "Copying data from the GPU");
81
82    wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");
83
84    wbSolution(arg, outputImage);
85
86    //@@ Insert code here
87
88    free(hostMaskData);
89    wbImage_delete(outputImage);
90    wbImage_delete(inputImage);
91
92    return 0;
93  }
```

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 wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      return -1;                                                          \
    }                                                                     \
  } while (0)

#define Mask_width 5
#define Mask_radius Mask_width / 2
#define TILE_WIDTH 16
#define w (TILE_WIDTH + Mask_width - 1)
#define clamp(x) (min(max((x), 0.0), 1.0))

//@@ INSERT CODE HERE
__global__ void convolution(float *I, const float *__restrict__ M,
                            float *P, int channels, int width,
                            int height) {
  __shared__ float N_ds[w][w];
  int k;
  for (k = 0; k < channels; k++) {
    // First batch loading
    int dest = threadIdx.y * TILE_WIDTH + threadIdx.x, destY = dest / w,
        destX = dest % w,
        srcY  = blockIdx.y * TILE_WIDTH + destY - Mask_radius,
        srcX  = blockIdx.x * TILE_WIDTH + destX - Mask_radius,
        src   = (srcY * width + srcX) * channels + k;
    if (srcY >= 0 && srcY < height && srcX >= 0 && srcX < width) {
      N_ds[destY][destX] = I[src];
    } else {
      N_ds[destY][destX] = 0;
    }

    // Second batch loading
    dest =
        threadIdx.y * TILE_WIDTH + threadIdx.x + TILE_WIDTH * TILE_WIDTH;
    destY = dest / w, destX = dest % w;
    srcY = blockIdx.y * TILE_WIDTH + destY - Mask_radius;
    srcX = blockIdx.x * TILE_WIDTH + destX - Mask_radius;
    src  = (srcY * width + srcX) * channels + k;
    if (destY < w) {
      if (srcY >= 0 && srcY < height && srcX >= 0 && srcX < width) {
        N_ds[destY][destX] = I[src];
      } else {
```

```
48          N_ds[destY][destX] = 0;
49        }
50      }
51      __syncthreads();
52
53      float accum = 0;
54      int y, x;
55      for (y = 0; y < Mask_width; y++) {
56        for (x = 0; x < Mask_width; x++) {
57          accum +=
58              N_ds[threadIdx.y + y][threadIdx.x + x] * M[y * Mask_width + x];
59        }
60      }
61      y = blockIdx.y * TILE_WIDTH + threadIdx.y;
62      x = blockIdx.x * TILE_WIDTH + threadIdx.x;
63      if (y < height && x < width)
64        P[(y * width + x) * channels + k] = clamp(accum);
65      __syncthreads();
66    }
67  }
68
69  int main(int argc, char *argv[]) {
70    wbArg_t arg;
71    int maskRows;
72    int maskColumns;
73    int imageChannels;
74    int imageWidth;
75    int imageHeight;
76    char *inputImageFile;
77    char *inputMaskFile;
78    wbImage_t inputImage;
79    wbImage_t outputImage;
80    float *hostInputImageData;
81    float *hostOutputImageData;
82    float *hostMaskData;
83    float *deviceInputImageData;
84    float *deviceOutputImageData;
85    float *deviceMaskData;
86
87    arg = wbArg_read(argc, argv); /* parse the input arguments */
88
89    inputImageFile = wbArg_getInputFile(arg, 0);
90    inputMaskFile  = wbArg_getInputFile(arg, 1);
91
92    inputImage   = wbImport(inputImageFile);
93    hostMaskData = (float *)wbImport(inputMaskFile, &maskRows, &maskColumns);
94
95    assert(maskRows == 5);    /* mask height is fixed to 5 in this mp */
96    assert(maskColumns == 5); /* mask width is fixed to 5 in this mp */
97
98    imageWidth    = wbImage_getWidth(inputImage);
99    imageHeight   = wbImage_getHeight(inputImage);
100   imageChannels = wbImage_getChannels(inputImage);
```

```
101
102     outputImage = wbImage_new(imageWidth, imageHeight, imageChannels);
103
104     hostInputImageData  = wbImage_getData(inputImage);
105     hostOutputImageData = wbImage_getData(outputImage);
106
107     wbTime_start(GPU, "Doing GPU Computation (memory + compute)");
108
109     wbTime_start(GPU, "Doing GPU memory allocation");
110     cudaMalloc((void **)&deviceInputImageData,
111                imageWidth * imageHeight * imageChannels * sizeof(float));
112     cudaMalloc((void **)&deviceOutputImageData,
113                imageWidth * imageHeight * imageChannels * sizeof(float));
114     cudaMalloc((void **)&deviceMaskData,
115                maskRows * maskColumns * sizeof(float));
116     wbTime_stop(GPU, "Doing GPU memory allocation");
117
118     wbTime_start(Copy, "Copying data to the GPU");
119     cudaMemcpy(deviceInputImageData, hostInputImageData,
120                imageWidth * imageHeight * imageChannels * sizeof(float),
121                cudaMemcpyHostToDevice);
122     cudaMemcpy(deviceMaskData, hostMaskData,
123                maskRows * maskColumns * sizeof(float),
124                cudaMemcpyHostToDevice);
125     wbTime_stop(Copy, "Copying data to the GPU");
126
127     wbTime_start(Compute, "Doing the computation on the GPU");
128     //@@ INSERT CODE HERE
129     dim3 dimGrid(ceil((float)imageWidth / TILE_WIDTH),
130                  ceil((float)imageHeight / TILE_WIDTH));
131     dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
132     convolution<<<dimGrid, dimBlock>>>(deviceInputImageData, deviceMaskData,
133                                        deviceOutputImageData, imageChannels,
134                                        imageWidth, imageHeight);
135     wbTime_stop(Compute, "Doing the computation on the GPU");
136
137     wbTime_start(Copy, "Copying data from the GPU");
138     cudaMemcpy(hostOutputImageData, deviceOutputImageData,
139                imageWidth * imageHeight * imageChannels * sizeof(float),
140                cudaMemcpyDeviceToHost);
141     wbTime_stop(Copy, "Copying data from the GPU");
142
143     wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");
144
145     wbSolution(arg, outputImage);
146
147     cudaFree(deviceInputImageData);
148     cudaFree(deviceOutputImageData);
149     cudaFree(deviceMaskData);
150
151     free(hostMaskData);
152     wbImage_delete(outputImage);
153     wbImage_delete(inputImage);
```

```
154
155     return 0;
156 }
```