A3 – CUDA

Reference: Hwu, Programming Massively Parallel Processors, 2e/3e.
Deadline: January, 31, 2018.

These assignment questions are courtesy the GPU Accelerated Computing kit by UIUC and NVIDIA. Dataset generators and the template CUDA code may have errors. The image processing programs in this assignment use image read/write code from libwb (https://github.com/abduld/libwb). Do understand what's happening under the hood to extract the max out of this assignment.

GPU Server: Student logins have been created on the K40 GPU Server. SSH to the server. In case of any questions, contact Pramod (p.yelmewad@gmail.com). 35 student accounts have been created – one per team.
ssh <loginname>@10.100.12.123
Login: student01 to student35. Password is the same as the login.

Q1. Write the device query code, compile and run it on your system. Query enough information to know all the details of your device. Example queries: GPU card's name, GPU computation capabilities, Maximum number of block dimensions, Maximum number of grid dimensions, Maximum size of GPU memory, Amount of constant and share memory, Warp size, etc. Answer the following questions in your report.

1. What is the architecture and compute capability of your GPU?
2. What are the maximum block dimensions for your GPU?
3. Suppose you are launching a one dimensional grid and block. If the hardware's maximum grid dimension is 65535 and the maximum block dimension is 512, what is the maximum number threads can be launched on the GPU?
4. Under what conditions might a programmer choose not want to launch the maximum number of threads?
5. What can limit a program from launching the maximum number of threads on a GPU?
6. What is shared memory? How much shared memory is on your GPU?
7. What is global memory? How much global memory is on your GPU?
8. What is constant memory? How much constant memory is on your GPU?
9. What does warp size signify on a GPU? What is your GPU's warp size?
10. Is double precision supported on your GPU?

Q2. Write a CUDA program to calculate the sum of the elements in an array. The array contains single precision floating point numbers. Generate your input array. For this question, do the following.

1. Allocate device memory
2. Copy host memory to device
3. Initialize thread block and kernel grid dimensions
4. Invoke CUDA kernel
5. Copy results from device to host
6. Free device memory
7. Write the CUDA kernel that computes the sum

Q3. Perform Matrix Addition of two large integer matrices in CUDA. Answer the following questions.

1. How many floating operations are being performed in the matrix addition kernel?
2. How many global memory reads are being performed by your kernel?
3. How many global memory writes are being performed by your kernel?

Q4. Implement an efficient image blurring algorithm for an input image. An 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 (Gaussian Blur). Edit the code in the template to perform the following:

1. allocate device memory
2. copy host memory to device
3. initialize thread block and kernel grid dimensions
4. invoke CUDA kernel
5. copy results from device to host
6. deallocate device memory

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.

A pseudo-code version of the algorithm is shown below:

```
BLUR_SIZE = 1 // gives a 3x3 BLUR window
foreach pixel in the image; do
  get pixel values of all valid pixels under the BLUR_SIZE x BLUR_SIZE window;
  sum the pixel values of all valid pixels under the window;
  new pixel value = (sum of pixel values) ÷ (no. of valid pixels) // average of the window pixel values
done
```

Answer the following questions.

1. How many floating operations are being performed in your color conversion kernel?
2. How many global memory reads are being performed by your kernel?
3. How many global memory writes are being performed by your kernel?
4. Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

Q5. Convert an RGB image into a gray scale image. The input is an RGB triple of float values. You have to convert the triplet to a single float grayscale intensity value. A pseudo-code version of the algorithm is shown below:

```
for ii from 0 to height do
   for jj from 0 to width do
      idx = ii * width + jj
      # here channels is 3
      r = input[3*idx]
      g = input[3*idx + 1]
      b = input[3*idx + 2]
      grayImage[idx] = (0.21*r + 0.71*g + 0.07*b)  // converts 3 r g b values to a single grayscale value.
   end
end
```

Image Format

The input image is in PPM P6 format while the output grayscale image is to be stored in PPM P5 format. You can create your own input images by exporting your favorite image into a PPM image. On Unix, **bmptoppm** converts BMP images to PPM images (you could use **gimp** or similar tools too).

Run command:
`./ImageColorToGrayscale_Template -e <expected.pbm> -i <input.ppm> -o <output.pbm> -t image`
where `<expected.pbm>` is the expected output, `<input.ppm>` is the input dataset, and `<output.pbm>` is an optional path to store the results. Questions.

1. How many floating operations are being performed in your color conversion kernel?
2. Which format would be more efficient for color conversion: a 2D matrix where each entry is an RGB value or a 3D matrix where each slice in the Z axis representes a color. I.e. is it better to have color interleaved in this application? can you name an application where the oposite is true?
3. How many global memory reads are being performed by your kernel?
4. How many global memory writes are being performed by your kernel?
5. Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

Q6. Implement a tiled dense matrix multiplication routine using shared memory. Use the template code. Run command:
`./TiledMatrixMultiplication_Template -e <expected.raw>  -i <input0.raw>,<input1.raw> -o <output.raw> -t matrix`
where `<expected.raw>` is the expected output, `<input0.raw>,<input1.raw>` is the input dataset, and `<output.raw>` is an optional path to store the results.

1. How many floating operations are being performed in your matrix multiply kernel? explain.
2. How many global memory reads are being performed by your kernel? explain.
3. How many global memory writes are being performed by your kernel? explain.
4. Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.
5. Compare the implementation difficulty of this kernel compared to the previous MP. What difficulties did you have with this implementation?
6. Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.
7. Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

Q7. 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 question.

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. Run 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.
2. Were there any difficulties you had with completing the optimization correctly.
3. Which optimizations gave the most benifit?
4. For the histogram kernel, how many global memory reads are being performed by your kernel? explain.
5. For the histogram kernel, how many global memory writes are being performed by your kernel? explain.
6. For the histogram kernel, how many atomic operations are being performed by your kernel? explain.
7. For the histogram kernel, what contentions would you expect if every element in the array has the same value?
8. For the histogram kernel, what contentions would you expect if every element in the input array has a random value?

Q8. **Convolution**. 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 Q).

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.

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 template to perform the following:

1. allocate device memory
2. copy host memory to device
3. initialize thread block and kernel grid dimensions
4. invoke CUDA kernel
5. copy results from device to host
6. deallocate device memory
7. implement the tiled 2D convolution kernel with adjustments for channels
8. use shared memory to reduce the number of global accesses, handle the boundary conditions in when loading input list elements into the shared memory

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
```

Run 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.
2. How many floating operations are being performed in your convolution kernel? explain.
3. How many global memory reads are being performed by your kernel? explain.
4. How many global memory writes are being performed by your kernel? explain.
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.
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?
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?
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)?
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?
10. What is the identity mask?

****