A4 – Histograms, Convolution and Stencils

Reference: Wen Mei Hwu, PMPP, 2e. Module 11.
Deadline: 9AM, August 23, 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.

Q1. 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?

Q2. 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 privatized histogram in shared memory for each thread block, then atomically modifying the global histogram. Run 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.
2. Which optimizations gave the most benifit.
3. For the histogram kernel, how many global memory reads are being performed by your kernel? explain.
4. For the histogram kernel, how many global memory writes are being performed by your kernel? explain.
5. For the histogram kernel, how many atomic operations are being performed by your kernel? explain.
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?

Q3. **Thrust Histogram Sort.** Implement a histogramming algorithm for an input array of integers. This approach composes several distinct algorithmic steps to compute a histogram, which makes Thrust a valuable tools for its implementation.

Problem Setup
Consider the dataset
input = [2 1 0 0 2 2 1 1 1 1 4]
A resulting histogram would be
histogram = [2 5 3 0 1]
reflecting 2 zeros, 5 ones, 3 twos, 0 threes, and one 4 in the input dataset.
Note that the number of bins is equal to max(input) + 1

Histogram Sort Approach
First, sort the input data using thrust::sort. Continuing with the original example:
sorted = [0 0 1 1 1 1 1 2 2 2 4]
Determine the number of bins by inspecting the last element of the list and adding 1:
num_bins = sorted.back() + 1
To compute the histogram, we can compute the culumative histogram and then work backwards. To do this in Thrust, use thrust::upper_bound. upper_bound takes an input data range (the sorted input) and a set of search values, and for each search value will report the largest index in the input range that the search value could be inserted into without changing the sorted order of the inputs. For example,
[2 8 11 11 12] = thrust::upper_bound([0 0 1 1 1 1 1 2 2 2 4], *// input*
[0 1 2 3 4]) *// search*
By carefully crafting the search data, thrust::upper_bound will produce a cumulative histogram. The search data must be a range [0,num_bins).
Once the cumulative histogram is produced, use thrust::adjacent_different to compute the histogram.
[2 5 3 0 1] = thrust::adjacent_difference([2 8 11 11 12])
Check the thrust documentation for details of how to use upper_bound and adjacent_difference. Instead of constructing the search array in device memory, you may be able to use thrust::counting_iterator.
`./ThrustHistogramSort_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. This is a simplified version of the material presented in the Thrust repository here. Questions.

1. Are there places in your solution where there is an implicit memory copy between the host and device (a copy that is not from thrust::copy)?
2. What is the asympotic runtime of this approach to histogram ("bigO" time)?
3. What is the asympotic runtime of the privatized atomic approach to histogram?
4. Why might the developers of Thrust not provide a straightforward interface?

Q4. **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?

Q5. Perform **shared-memory tiling** by implementing a 7-point stencil. Instructions:

- Edit the template code to implement a 7-point stencil.
- Edit the template code to launch the kernel you implemented. The function should launch 2D CUDA grid and blocks.

Algorithm

You will be implementing a 7-point stencil without having to deal with boundary conditions. The result is clamped so the range is between the values of 0 and 255.

```
for i from 1 to height-1:   # notice the ranges exclude the boundary
  for j from 1 to width-1:  # this is done for simplification
    for k from 1 to depth-1:# the output is set to 0 along the boundary
      res = in(i, j, k + 1) + in(i, j, k - 1) + in(i, j + 1, k) +
          in(i, j - 1, k) + in(i + 1, j, k) + in(i - 1, j, k) -
          6 * in(i, j, k)
      out(i, j, k) = Clamp(res, 0, 255)
    end
  end
end
```

with Clamp defined as

```
def Clamp(val, start, end):
  return Max(Min(val, end), start)
end
```

and in(i, j, k) and out(i, j, k) are helper functions defined as

```
#define value(arry, i, j, k) arry[(( i )*width + (j)) * depth + (k)]
#define in(i, j, k)   value(input_array, i, j, k)
#define out(i, j, k)  value(output_array, i, j, k)
```

Questions

1. How many global memory reads does your program make?
2. How many shared memory reads does your program make?
3. This stencil would need a 3x3 convolution kernel, where the center entry is -6 and the adjacent entries are 1. Corner entries would be 0.
4. Does your stencil make more, equal, or fewer memory accesses than the equivalent 3x3 convolution code would?