Lab Assignment 4
Due Nov 6, 2023 at 11:59pm

1 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 an arbitrarily sized image (assume the image dimensions are greater than 5x5 for this Lab).

2 Instructions

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`.

Please use the "output tiling" algorithm discussed in the class for this assignment. You need to keep the thread block size the same as the tile size in your output matrix. All threads will equally participate in computing output matrix elements, but a subset of threads will load more elements into the shared memory than the others.

The code template in `template.cu` provides a starting point and handles the import and export as well as the checking of the solution. Students are expected to insert their code demarcated with `//@@`. Students are expected to leave the other code unchanged.
Edit the skeleton code to perform the following:

- Allocate device memory
- Copy host memory to device
- Initialize grid and thread blocks
- Launch the kernel
- Copy results from device to host
- Free device memory
- Write the CUDA kernel

Compile the template with the provided `Makefile`. The executable generated as a result of compilation can be run using the following code:

```
    ./ConvolutionTemplate -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.

`README.md` has details on how to build `libgputk`, `template.cpp` and the dataset generator.

3 Input Data

The input is an interleaved image of *height* × *width* × *channels*. By interleaved, we mean that 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.

4 What to Turn in

Submit a report that includes the following:

1. How many floating operations are being performed by your convolution kernel?
2. How many global memory reads are being performed by your convolution kernel?
3. How many global memory writes are being performed by your convolution kernel?
4. 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?
5. 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?
6. Your version of `template.cu`.
7. The result as a table/graph of kernel execution times for different input data, with the system information where you performed your evaluation. Run your implementation with the input generated by the provided dataset generator. For time measurement, use `gpuTKTime_start` and `gpuTKTime_stop` functions (You can find details in `libgputk/README.md`).