Programming Assignment 4 - Optimized Convolutions

1. Task Description

In Programming Assignment 1, we implemented a 2D convolution kernel that operates on one input image and one filter. In this assignment, we extend the convolution kernel, and the accompanying host code, to operate on multiple images and filters. In other words, we relax the constraints on **N** and **K**. We also fix **R** for simplicity. Following are the new constraints,

Input image height: **H, 64 <= H <= 256**
Input image width: **W, 64 <= W <= 256**
Filter height = Filter width: **R, R = 3**
Horizontal stride = Vertical stride: **U, U = 1**
Number of input images: **N, 1 <= N <= 16**
Number of filters: **K, 1 <= K <= 512**
Both the input and the filter will have **float** values in them. Do not use **double** data types in your code.

You will do this task in two steps,

1. **Extend your convolution kernel from the first programming assignment**. Make the necessary changes to your original kernel so that it can convolve an arbitrary number of input images with an arbitrary number of filters. This means you will launch your kernel only once, and it will convolve all the input images with the filters. This will be the first version of your code (conv2dV1.cu)
2. **Optimize your convolution kernel.** Optimize your new kernel using Shared Memory. Each thread block(TB) will load a tile of the input image to shared memory, convolve it with the filters, and store it to the output image, and repeat this **N** times. See the next section for details on shared memory optimization. This will be the second version of your code (conv2dV2.cu)

You need to convolve every input image with every filter. In total, for a given input and filter set, you will perform **N*K** convolutions, and thus generate **N*K** output images.

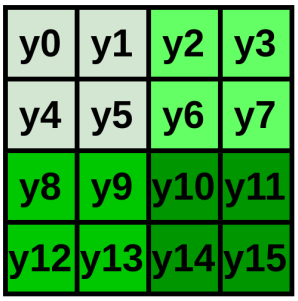
Shared Memory Optimization

CUDA offers a software managed cache under the name “Shared Memory”. This structure is often used to optimize memory-bound applications. The idea is to bring a smaller chunk of the input to the shared memory, process it, store it back, and repeat until the entire input is processed. Physically, the data in shared memory is stored in the L1 cache of the SMs, therefore the space is limited, and it is in the range of kilobytes. Following is an example declaration of a shared memory array in the CUDA kernel,

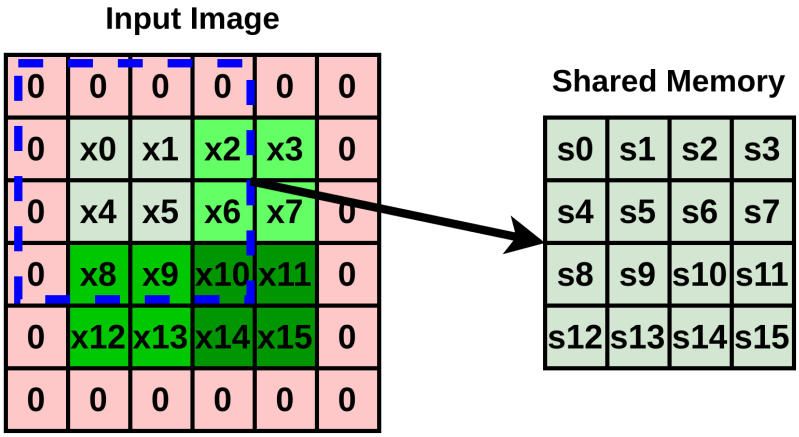
```
__shared__ float input_image[64];
```

The size of the shared memory array must be defined at compile time, much like C arrays.

You will use shared memory to store tiles of the input image. Let's walk through a simple example. Let's say the input and output images are 4x4, and the filter is 3x3. We first determine the thread block and grid dimensions. Let's say each TB computes a 2x2 tile of the output, which means we need 4 TBs, so the grid dimension is 2x2, as shown in the figure below,



The figure above shows the output image. Each shade of green represents a TB. Each thread within a TB computes one element of the output. When doing shared memory optimization, it is imperative to think in terms of thread blocks, because shared memory is "shared" among the threads of a thread block. Figure below shows the corresponding input image with padding. I annotated the elements **TB0(blockIdx.x=0, blockIdx.y=0)** needs to load into its shared memory.



Notice, even though the TB has 4 threads in it, it needs to load 16 elements from the input image, due to the halo of the convolution operation. This could mean that some threads will have to load more elements than the others, but that depends on your implementation.

Once the input tile is loaded into shared memory, you need to carry out the convolution operation, and store the results. Don't forget to do `__syncthreads()` between loading the input and performing the convolution, as you need to make sure all threads finished loading the data before doing the math.

2. Input format and expected output

Input and output formats are slightly different for this assignment. There will be two .txt files; one will contain the input images, the other will contain the filters. The first three lines in the input image file will be **H**, **W** and **N**, in that particular order. **H** and **W** will be the same for all images in the input file. Each following line will contain one entire row of an input image, where each element in the row is separated by one whitespace. The input format for the filter is similar. The first line in the filter file will be **K**. Each following line will contain one entire row of a filter. The names of the input .txt files are not fixed, so your program should just read them as arguments from the command line.

The output format is identical to the input format. Your program should print out all the output images, starting with the output generated by convolving the first input image with the first filter, followed by the output of the first image and the second filter, and so on.

Your program should only print out the output images, with format identical to the input. The values should be printed to **stdio** with at least 3 decimal points precision.

You can and should test your code with the python script we provide. When grading, your code will be tested using that script, so if your code does not pass the tests in the script, you will lose points. Feel free to tinker with the script during development, but ultimately, make sure your code passes all the tests with 0.002 error at most. On Hydra, you will need to install numpy first `` `pip3 install --user numpy` ``. Then you can simply run `` `python3 build_and_test.py` `` in your directory, and that will build your code and run the tests.