CS426/525
Fall 2022
Project 4
Due 4/1/2023

In Project 3, we have seen parallelization of convolution operation and implemented it using OpenMP. This project will be an extension of previous one. However, this time, we will be utilizing CUDA. You will implement 3 versions of parallel convolution and compare them in terms of performance using Nvprof.

Problem Statement

We will focus on 2D convolution this time.

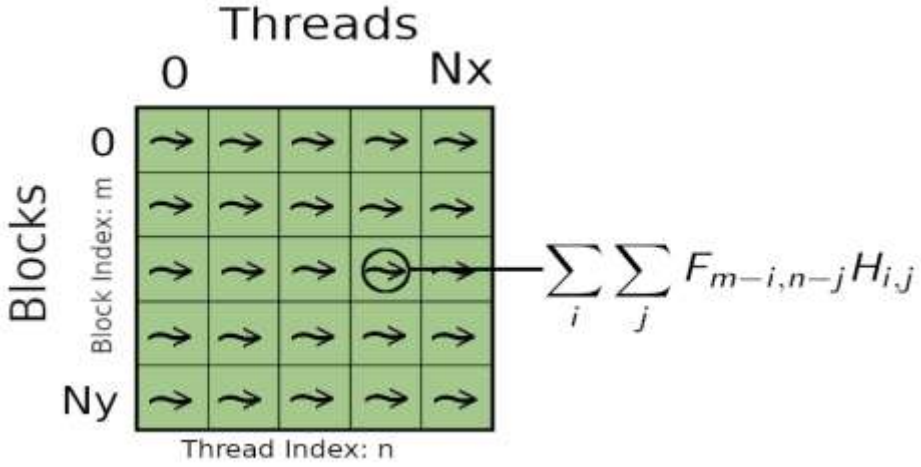

Figure 1: Data decomposition among threads.

A naive approach based on the data decomposition given in Figure 1 can be summarized as follows. Each element in the output grid can be computed independently so it would make sense to have one thread per output index which is responsible for its computation. This means that this thread must

1. Read the input elements in its neighborhood.
2. Read the convolutional kernel.
3. Perform the convolution computation.
4. Write the result of the computation to its index in the output array.

The problem with this approach is that, both the source and the kernel matrices are in global memory, which results global memory calls each time we need to shift and multiply (inner loop of the convolution algorithm).

An optimization can be done by loading our kernel into constant memory. CUDA runtime will initially read the convolutional kernel from global memory as before however now it will cache it since it knows it will never be modified. Figure 2 gives the details of Nvidia GPU architecture.

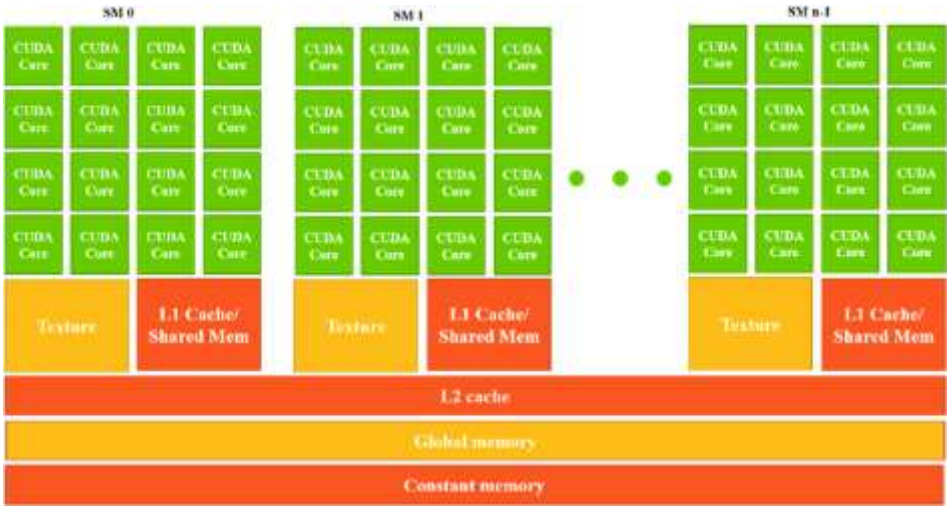

Figure 2: Nvidia GPU architecture and the memory model.

Further optimization can be done by using shared memory (tiled convolution as shown in Figure 3). Shared memory is memory that is shared within threads in each block and is much faster to read from than global memory. Since each streaming multiprocessor is assigned to each block, shared memory in a streaming multiprocessor is used for reusing the pixel data. Storing the pixel data is carried out by the following two steps:

- Transferring an image from a host to global memory of a device,
- Copying the pixel data in a block to shared memory by threads.

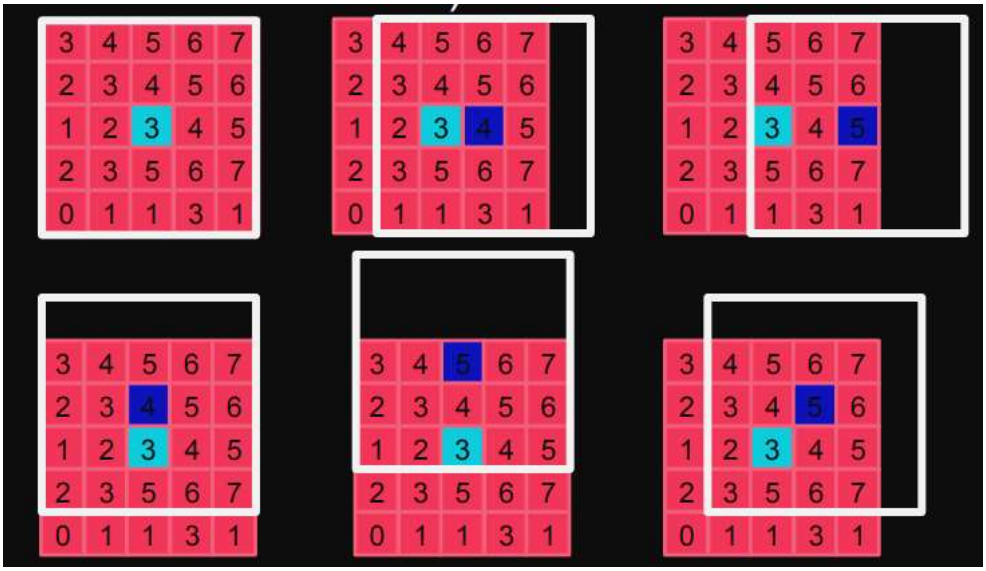

Figure 3: Each element in the tile is used in calculating other elements in the tile.

In order to perform filtering for the pixel data in a block correctly, the pixel data surrounding a block should be copied to shared memory. For a (2S+1) x (2S+1) filter, we should add S pixel(s) to a block as shown in Figure 4. A block with the surrounding data is called a tile here, and all the pixel data in a tile have to be copied.

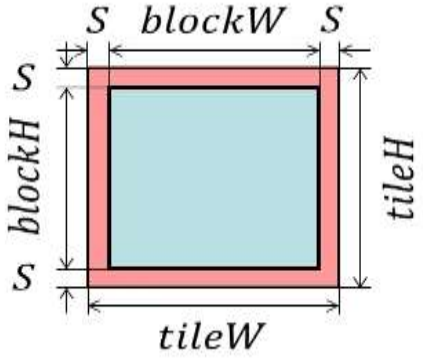

Figure 4: Tile for a single block.

Implementation Details

- There is no serial implementation for this project.
- Your program will read a source matrix and a kernel matrix, given as text ASCII files (same format as in Project 3).
- Border extend your input source matrix. You can use your code from previous project.
- Implement naive parallel 2D convolution. Use *nvprof* tool to analyze your parallel program running under GPU.
- Optimize your program by taking advantage of constant memory. Use *nvprof* tool to analyze your parallel program running under GPU.
- Further optimize your program by implementing tiled 2D convolution. Use *nvprof* tool to analyze your parallel program running under GPU.
- Prepare a Makefile that will compile your code along with any necessary files (such as util files) that will produce 3 executables:
    1. conv_naive
    2. conv_cons
    3. conv_tiled

- Input format for your program should be in the form of:

$ ./program source.txt kernel.txt output.txt

Assumptions

- Kernel is a square matrix and assume there is no negative value in the kernel matrix.
- All threads of a block should be assigned to all pixels in a tile, you can adjust block size in this manner. For more information, you can Google "*maximum allowed threads per block cuda*".

Language

You should use C or C++ language with CUDA (You can refer to course slides for specifications). You may use the following command to compile your files:

nvcc filename.cu -o program

Report

You should write a short report. **Reports should be in .pdf format**, submissions with wrong format will get 0. Your report should include:

- Detailed description of your implementations
- Detailed description of Nvprof's profiling outputs
- Analysis of your parallel implementation (communication cost, expected speedup etc.)
- Plots for execution times/speedups with various inputs
- What are your observations on relationship between kernel size and constant memory implementation?
- How would the performance vary by changing block size in tiled implementation? How it would affect the amount of data reuse?
- Discussion of your results, don't forget to use Nvprof's profiling outputs in your discussion.

Grading

- Naive implementation: 20
- Constant memory implementation: 20
- Tiled implementation: 30
- Report: 30

Submission

Put all relevant code, makefile, nvprof outputs and your report into a zip file.

Name the zip file: yourname_lastname_p4.zip

Submit to the respective assignment on Moodle.