HPCA Programming Assignment 2023-2024
Optimizing Performance of Dilated Convolution

In this assignment, your task is to optimize the problem of "Dilated Convolution (DC)" which involves an input matrix and a kernel matrix. The sample algorithm is pictorially depicted below. A sample single- threaded unoptimized implementation is also provided via Github — check the bottom of this document for the same. (The Github repository also has an animation depicting how the algorithm works).

Input:
A. Input Matrix of dimensions: Input_Row x Input_Column.
B. Kernel Matrix of dimensions: Kernel_Row x Kernel_Column.
Output:
An Output Matrix of dimensions: (Input_Row-Kernel_Row+1)
x (Input_Column – Kernel_Column +1)

You will have to work on the programming assignment **in groups of at most two.** The assignment should be submitted on **Gradescope**.

Explanation and algorithm of DC:

Dialated Convolution (DL) is a variant of the convolution operation. Following is the explanation of the algorithm:

Let us assume that for the explanation of the algorithm that the dimensions of the Input Matrix and Kernel Matrix are as follows:

Input Matrix (I): 4x4

Kernel Matrix (K): 2x2

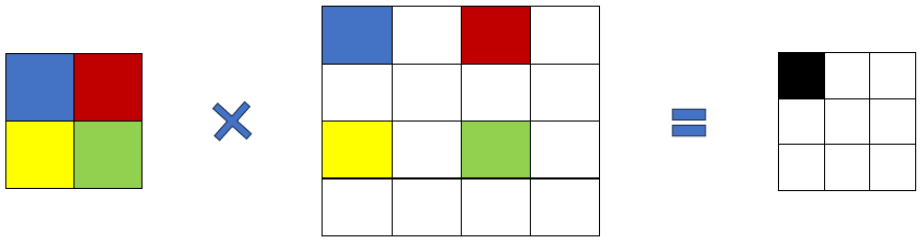
So, dimensions of the Output Matrix (O) shall be (based on calculation): 3x3.

[Output_Row = Input_Row – Kenrel_Row + 1 = 4 – 2 + 1 = 3. Calculation is similar for Output_Column]

The Kernel Matrix (K) slides over the Input Matrix (I) and produces each cell of the Output Matrix (O) as shown below. In the diagrams below, The Kernel Matrix (K) is the left-most, the Input-Matrix (I) is in the middle, and the Output Matrix (O) is the right-most.

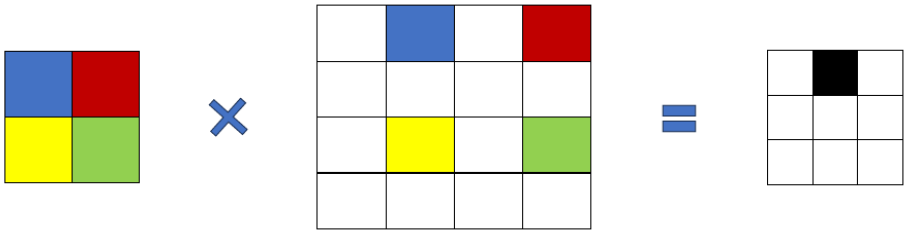
Step 1: The Kernel Matrix(K) is super imposed on the Input Matrix(I) as shown in the figure below. The elements of K and I are multiplied and accumulated to get the cell in the 0^{th} row and 0^{th} column of the Output Matrix(O) as follows:

$$O_{00} = I_{00} * K_{00} + I_{02} * K_{01} + I_{20} * K_{10} + I_{22} * K_{11}$$



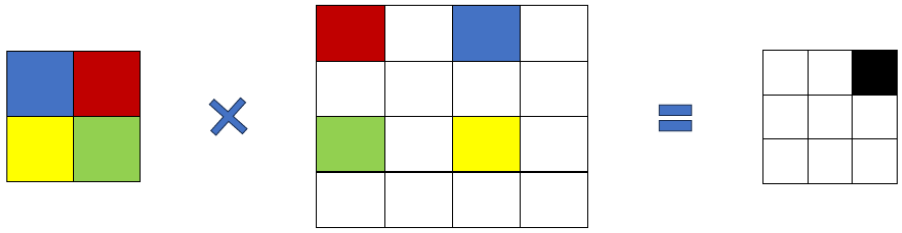
Step 2: Next the Kernel Matrix (K) slides by 1 column on the right to generate the cell in the 0^{th} row and 1^{st} column of the Output Matrix (O) as follows:

$O_{01} = I_{01} * K_{00} + I_{03} * K_{01} + I_{21} * K_{10} + I_{23} * K_{11}$



Step 3: Next the Kernel Matrix (K) slides by 1 column more on the right to generate the cell in the 0th row and 2nd column of the Output Matrix (O) as shown below. Note how the Kernel Matrix (K) while super imposition wraps around.

$O_{02} = I_{02} * K_{00} + I_{00} * K_{01} + I_{22} * K_{10} + I_{20} * K_{11}$



And so on...

You can find the code for this algorithm in the Github repository. Also, the Github repository has a test directory, which contains a test program where you can play around with the sizes of the Input-Matrix (I) and Kernel-Matrix (K) to get a flavor of how each cell of Output-Matrix(O) is generated. A stripped output of test executable mentioned above for the example discussed:

```
abhishek@abhishek:~/Documents/Research_Codes/hpca-assignment-2023/test$ ./dilated_convolution
output[0][0] = input[0][0] * kernel[0][0] + input[0][2] * kernel[0][1] + input[2][0] * kernel[1][0] + input[2][2] * kernel[1][1]
output[0][1] = input[0][1] * kernel[0][0] + input[0][3] * kernel[0][1] + input[2][1] * kernel[1][0] + input[2][3] * kernel[1][1]
output[0][2] = input[0][2] * kernel[0][0] + input[0][0] * kernel[0][1] + input[2][2] * kernel[1][0] + input[2][0] * kernel[1][1]
output[1][0] = input[1][0] * kernel[0][0] + input[1][2] * kernel[0][1] + input[3][0] * kernel[1][0] + input[3][2] * kernel[1][1]
output[1][1] = input[1][1] * kernel[0][0] + input[1][3] * kernel[0][1] + input[3][1] * kernel[1][0] + input[3][3] * kernel[1][1]
output[1][2] = input[1][2] * kernel[0][0] + input[1][0] * kernel[0][1] + input[3][2] * kernel[1][0] + input[3][0] * kernel[1][1]
output[2][0] = input[2][0] * kernel[0][0] + input[2][2] * kernel[0][1] + input[0][0] * kernel[1][0] + input[0][2] * kernel[1][1]
output[2][1] = input[2][1] * kernel[0][0] + input[2][3] * kernel[0][1] + input[0][1] * kernel[1][0] + input[0][3] * kernel[1][1]
output[2][2] = input[2][2] * kernel[0][0] + input[2][0] * kernel[0][1] + input[0][2] * kernel[1][0] + input[0][0] * kernel[1][1]
```

Details for submitting:

There are three major activities in this assignment, divided in two parts (PartA, PartB):

1. **[PartA-I] Optimize single-threaded DC (CPU):** In this part, you will hand optimize the single-threaded implementation provided on Github.
   Tips: Identify performance bottleneck in your code using hardware performance counters (e.g., via perf) and optimize the source accordingly.
   Relevant file: PartA/header/single_thread.h
2. **[PartA-II] Implement and optimize multi-threaded DC (CPU):** In this part, you will implement a multi-threaded version of DC. Test the scalability of your implementation by varying the number of threads and optimize it to achieve maximum scalability (an ideal

implementation will achieve a speedup of 2x if the number of cores are increased by a factor of 2, and so on). It is mandatory to use "pthreads" for multi-threaded implementation. Do not use any other custom libraries.
Relevant file: PartA/header/multi_thread.h

3. **[PartB] Implement and optimize DC in CUDA (GPU):** In the final part, you are required to implement DC for a GPU using CUDA.
Relevant file: PartB/header/gpu_thread.h

Deliverables: Submission will be in two parts—one for the CPU-based implementations (this will include both single threaded and multi-threaded implementations), and the other for GPU-based implementation. For each part, you are required to submit a report along with your code. Your report should contain details such as the runtime of unoptimized code, and the effect of each of your optimizations on the runtime over different input sizes [say, n = 4k, 8k and 16k, you can take square matrices for both Input Matrix (I) and Output Matrix(O)]. (The Github repository shared has corresponding **Makefile**s which will give you an idea of how to run the executables with example dimensions. **Note that those particular sizes are for your testing. Evaluation shall be done on different Input and Kernel Matrix sizes.**) You should also include details such as how you identified the bottlenecks in the code (e.g., your analysis via hardware performance counters), and which optimizations you have implemented. For the multi-threaded implementation, your report must include an evaluation of the scalability of your implementation (e.g., how does the runtime change with different thread count), along with important details of your implementation.

What/how to submit:

Clone the GitHub repository on your machine and follow its README.md for instructions on how to compile and run the programs. All your code must be implemented in "header/". DO NOT MODIFY main.cpp or main.cu
You need to submit your assignment as a zip file named as per the last five digits of your SR number (e.g., 15964.zip).
It should contain two directories – i.e., "header/" that would contain your implementation and "reports/". Name your reports as "reports/Report_PartA.pdf" and "reports/Report_PartB.pdf".

WARNING: IF YOU DO NOT SUBMIT THE CODE AS SPECIFIED ABOVE THEN YOUR ASSIGNMENT WILL NOT BE EVALUATED.

Deadline:

24th of November 2023. 11:59 pm.

Link to Github repository:

https://github.com/Abhishekghosh1998/hpca-assignment-2023

Tips for analyzing the performance of your software:

A little bit about the perf tool and hardware performance counters: Modern processors come with an extensive set of performance monitoring counters. It is also called the performance monitoring unit or PMU. These counters typically count occurrences of specific events, e.g., cache miss, TLB miss, etc. Tools like Linux's perf tools provide users with an easy-to-use command-line interface to program and measure these performance counters. There are other tools like Likwid that could do the same or even more.

To see what all performance counters are exposed through perf tool, you can run the following command

```
$ perf list
```

For measuring the number of L1-cache-load misses (from user side) for a command say, `sleep`

1 (sleeps for 1 second) you can use the following command

```
$ perf stat -e L1-dcache-load-misses:u sleep 1
```

There are tons of information and tutorials online on perf. A few examples are https://perf.wiki.kernel.org/index.php/Tutorial and http://www.brendangregg.com/perf.html

Performance measurement tips: Real hardware numbers are not deterministic. It changes from run to run. So we strongly suggest that run each measurement at least 5 times and report the mean (and standard deviation if you wish to).

Logistics: Where to run your CUDA programs?

For the CPU part of the assignment, you can use the desktop machines in the common lab or your laptop as long as they support the relevant performance counter. We will not be evaluating on absolute performance numbers you get but the optimizations you do.

For the GPU part, there are two options. You can use NVIDIA GPUs or run it on a GPU simulator (GPGPU-sim). If you want but don't have access to a system with NVIDIA GPUs, then you please contact department's system administrator, Mr. Akshay Nath: **akshaynath@iisc.ac.in,** for an account on Well Fargo server. However, expect the GPUs to be very busy during the end of the semester. Therefore, we recommend using GPGPU-sim (see below for installation instructions). This is a software that runs on the CPU but emulates a GPU. It runs CUDA application out-of-the-box. You can install this simulator on your local machine (your own laptop or machines in the common lab, running the x86 instruction set architecture) and do your homework.

You would do this assignment in a group of (at most) two.

Prerequisites:

To compile CUDA code (whether for a GPU or GPGPU-Sim), you will need to install the CUDA compiler (nvcc). You can download and install this from the official NVIDIA website (https://developer.nvidia.com/cuda-11.0-download-archive). You do NOT need to install the driver if you plan to use GPGPU-sim. GPGPU-Sim currently works with CUDA versions 4.5 - 11.0.Instructions for using GPGPU-sim:

- Clone GPGPU-sim using the following link:

  https://github.com/gpgpu-sim/gpgpu-sim_distribution

- Run your code for the SM7_TITANV configuration. You can find the configuration file at : gpgpu-sim_distribution/configs/tested-cfgs/

- You can use GPGPU-sim profiler to optimize the performance your CUDA application. If you are running on the GPU hardware, look for NSight and/or NVprof tools.