# Homework 3

*Part 2: GPU Algorithm*

William Young

29 November 2016

## 1 INTRODUCTION

For this portion of the assignment, I chose to implement 2D Convolution on the GPU. A brief web query showed that 2D convolution is a mathematical matrix transformation typically used in image processing to achieve a variety of visual effects such as edge detection, embossing, sharpening, shadowing, and highlighting (among others). A typical image kernel is usually a 3x3 matrix, but can be expanded to perform certain transformations.

## 2 ALGORITHM: INHERENT PARALLELISM

The algorithm to perform a 2D Convolution is well-suited to parallel computation. For each "pixel" of the input (in this project, a float-type), we produce a corresponding output "pixel" whose value is the sum of the multiplications resulting from the overlap of the kernel and input matrices.

Since each output pixel depends ONLY on the original input, we can parallelize the computation of the output pixels. The parallel algorithm is relatively simple: instead of iterating over each input pixel (requiring four FOR-loops) in a serial manner, we dispatch an individual thread to handle each pixel. We also declare a device function on the GPU to provide "clamping" functionality.

In this manner, each thread on the GPU calculates the output of a single input pixel. The GPUs used in this assignment are documented to support up to 16 million threads, thus we can compute 16 million pixels in “parallel” (roughly). Since a megapixel consists of 1024x1024 pixels, we are thus able to transform nearly 16 megapixels in one operation.

## 3 EXPECTED OPERATING PARAMETERS

Based on the thread limitations posed by the GPU hardware, this tool can support up to 16 million "pixels" and as large of a kernel as time permits (this is often limited by the CPU algorithm, which needs on the order of 100x more time to complete, running after

the GPU algorithm). In this scenario, an input matrix of 4000x4000 (16 million cells) represents a 61MB problem size (type FLOAT = 4 bytes).

Brief web queries indicate that a convolution kernel is typically 3x3 or 5x5. A high definition image runs on the order of 2048x2048 pixels (4 megapixels). As a result, we expect the "average" problem size of a 2D convolution (with respect to image processing) to be approximately 16 megabytes with a standard 3x3 kernel. For the purposes of measuring performance, we also test some configurations with larger kernels.

## 4 PERFORMANCE EVALUATION

I chose to test eleven different configurations, of which the first eight are non-intensive image processing workloads. The final three configurations are designed to significantly expand the problem complexity. Problem size is calculated by multiplying the pixel count by pixel size, which in this assignment is 4 bytes (float). In every tested case, the outputs of the GPU and CPU algorithms agree.

| Input Size | Kernel Size | Problem Size | Mean Kernel Runtime (s) | STDEV (s) | Mean CPU Runtime (s) | STDEV (s) | Mean Performance Improvement |
|---|---|---|---|---|---|---|---|
| 32x32 | 3x3 | 4 KB | 0.0000575 | 0.0000050 | 0.0000300 | 0.0000000 | 0.4887736 |
| 48x48 | 3x3 | 9 KB | 0.0000575 | 0.0000050 | 0.0000600 | 0.0000000 | 1.0550000 |
| 128x128 | 3x3 | 64 KB | 0.0000525 | 0.0000050 | 0.0004000 | 0.0000000 | 6.7800000 |
| 512x512 | 3x3 | 1 MB | 0.0001875 | 0.0000050 | 0.0068625 | 0.0002717 | 37.0600000 |
| 1024x1024 | 3x3 | 4 MB | 0.0005850 | 0.0000058 | 0.0326300 | 0.0079136 | 55.8075000 |
| 2048x2048 | 3x3 | 16 MB | 0.0021750 | 0.0000191 | 0.1445000 | 0.0508013 | 66.6650000 |
| 4000x4000 | 3x3 | 61 MB | 0.0080700 | 0.0000469 | 0.5306500 | 0.1620606 | 65.5875000 |
| 4000x4000 | 5x5 | 61 MB | 0.0221375 | 0.0000263 | 1.3673100 | 0.4770875 | 61.7425000 |
| 4000x4000 | 49x49 | 61 MB | 0.8688050 | 0.0001448 | 104.2920975 | 32.0930915 | 120.0375000 |
| 4000x4000 | 99x99 | 61 MB | 3.5098250 | 0.0065599 | 433.7021500 | 115.7187507 | 123.4525000 |
| 4000x4000 | 127x127 | 61 MB | 5.7908825 | 0.0060806 | 869.3049325 | 218.0555435 | 150.1900000 |

The results demonstrate that even for small workloads (10KB) the parallel GPU algorithm manages to outperform the serialized CPU algorithm. As workload increases to the size of an HD image (4 megapixels) the GPU's advantage over the CPU continues to grow, widening to a 67x performance increase. This advantage reaches a maximum with a kernel of size 127x127, topping out at 150x better than the CPU.

Note that with an input size of 16 megapixels (61MB here), the GPU produces output in 22ms. If we extend this value, we discover that the algorithm could theoretically batch-process nearly 125 16-megapixel images per second using a standard 3x3 kernel--a throughput of almost 7.5 GB/s based on the total problem size (excluding transfer times; see below).

$$125 \frac{\text{images}}{\text{sec}} \times 61 \frac{\text{MB}}{\text{image}} = 7625 \frac{\text{MB}}{\text{sec}} = 7.45 \text{ GB/s}$$

In reality, each pixel in a basic RGB image is comprised of 3 bytes (one each for red, green, and blue). As a result, a true 16-megapixel image is approximately 48MB uncompressed. If we modify the previous computation to account for this change, it turns out that we could actually process up to 160 16-megapixel images per second. The serial algorithm on the CPU pales in comparison, achieving only 2.2 images per second (135 MB/s). Adjusting for the 48MB image size, this equates to 2.8 images per second.