Histogramming

A Comparison of OpenCL, CUDA, and Python

Submitted by Alexander Stein

EECSE4750
Dr. Zoran Kostic
Assignment 4
Columbia University
December 6th, 2016

Problem Statement:

This assignment focuses on the problem of parallelizing the histogramming problem on the GPU, and comparing the outputs and performance with a serial implementation.

We are to implement this operation serially, and in parallel, and to compare the results with a “naive” parallel kernel which was provided at the outset.

Section A: Programming

Platform Information:

GPU: Nvidia GTX 660M
CPU: Intel Core i7 3630QM @ 2.4GHz
OS: Ubuntu 14.04 LTS
RAM: 24 GB SODIMM DDR3 Synchronous 1333 MHz (0.8 ns)

Overall Structure Pseudocode:

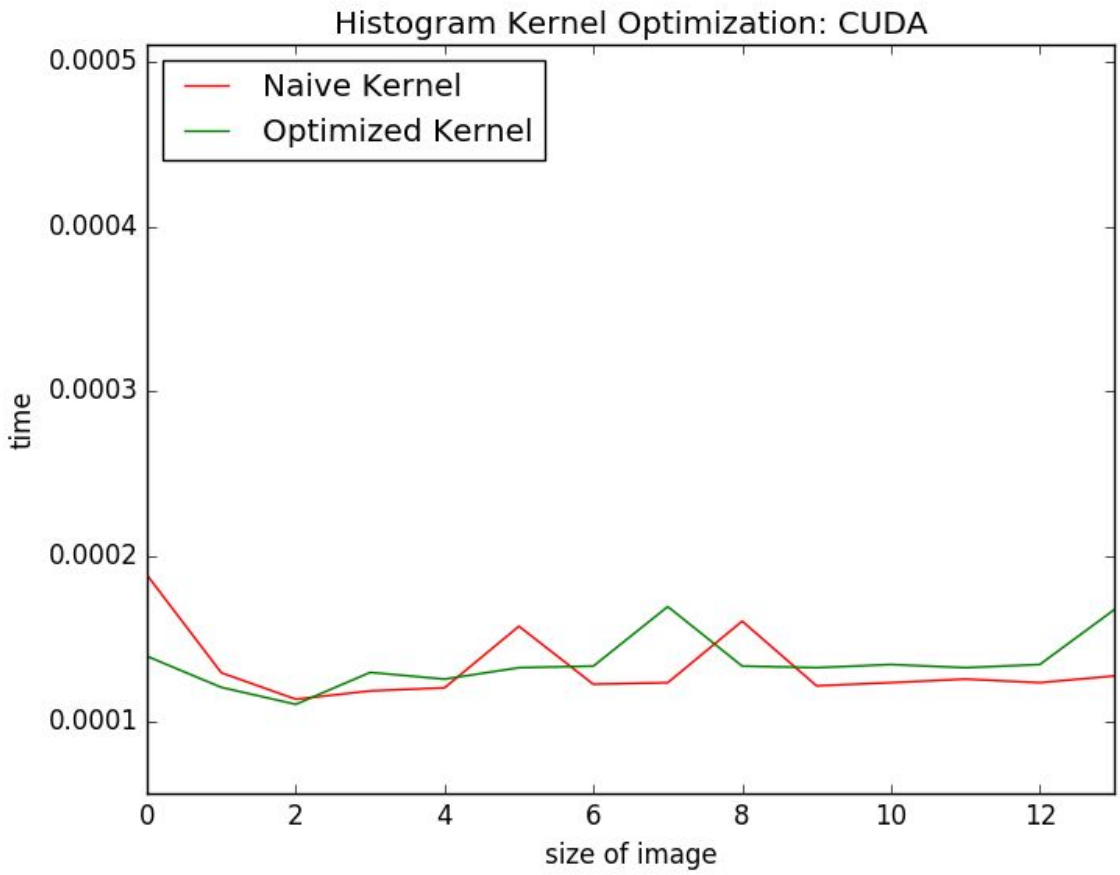
Since we have discussed the initial setup of GPU devices in detail through previous assignments, only the optimized kernel pseudo-code will be summarized here.

```
kernel( char* img, int* bins, int P, int R, int C){
    // P - warp size
    // R - number of rows in image
    // C - columns in image
    // Total Bytes in IMAGE = RxC

    #define TILE_WIDTH 16

    // global indices
    int gidx = global_id(rows);
    int gidy = global_id(cols);
```

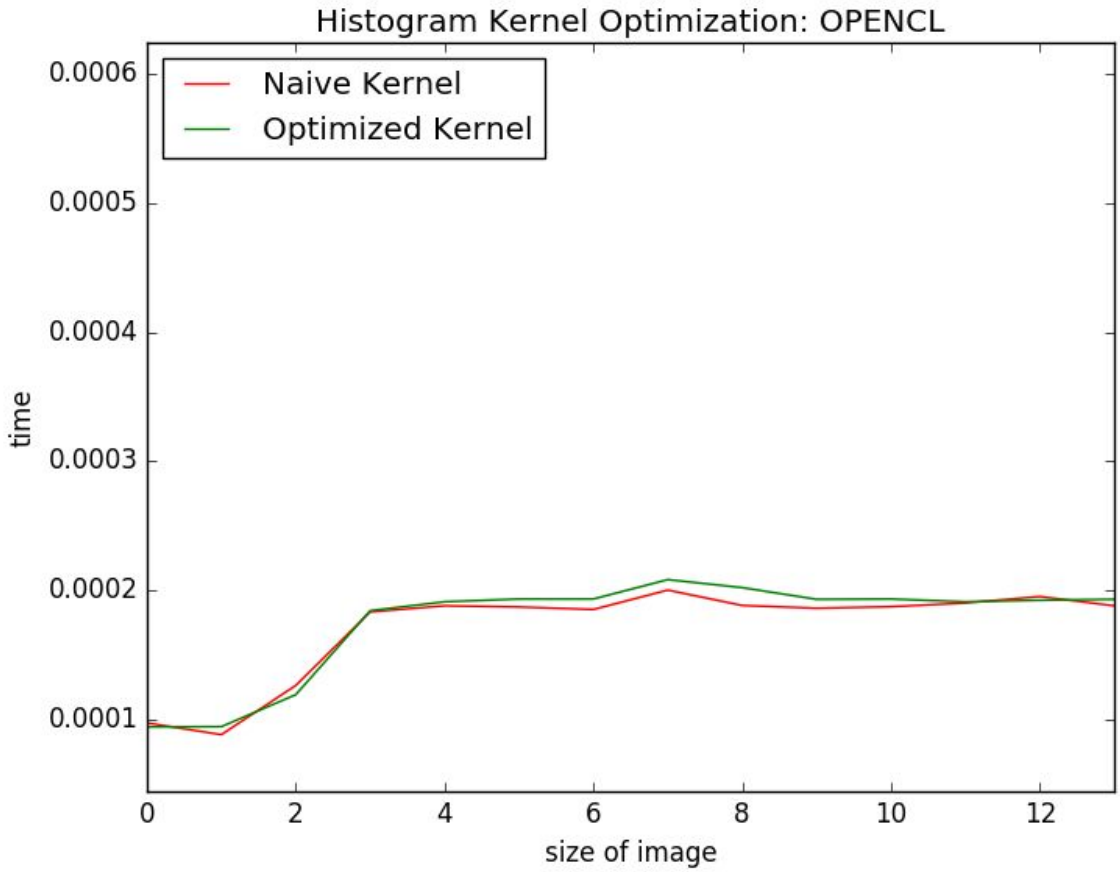
```
    // local indices
    int tile_id = local_id(x) + local_id(y)*TILE_WIDTH

    // shared memory used for local bins and local image tile
    volatile __shared unsigned char bins_loc[256];
    volatile __shared unsigned char img_tile[TILE_WIDTH][TILE_WIDTH];

    // fill local image tile with items from global image
    for(j=0;j<TILE_WIDTH;j++)
        for(k=0;k<TILE_WIDTH;k++)
            img_tile[j][k] = img[...]

    // Create Private Histogram from tile
    for(j=0; j<TILE_WIDTH; j++){
        for(k=0; k<TILE_WIDTH; k++){
            if(tile_idx==0) // once per tile!
                ++bins_loc[img_tile[j][k]];
        }
    }
    __barrier_synchronize();

    // Add tiled results to global output (once per tile!)
    if(tile_idx==0){
        for (k=0; k<256; k++)
            atomicAdd(&bins[k], bins_loc[k]);
    }

}
```

Results: Graphs with Explanation

Optimized vs Naive kernels in CUDA



Optimized vs Naive kernels in OPENCL



Results: Analysis

Optimization Explanation:

- Despite great effort and different approaches, none of my methods achieved even nominal speedup over the naive kernel supplied with the assignment instructions.
- The attempted method of optimization was tiling, in which I broke the input image up into blocks of 16x16. I think it would have been possible to get speedup by only deploying (R/16,C/16,1) threads, but wasn't able to get this working ontime.
- The OPENCL implementation does not produce the same level of correctness as the CUDA version, despite the fact that both implementations are exactly the same aside from the nuances between languages.