David Han
11/30/2017

Assignment 4

Naive Implementation

```
kernel_code_template = """
#include <stdio.h>
#include <math.h>

__global__ void naiveHisto(int *data,int* histogram,int size)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if(col < size && row < size){
        int index = col + row * size;
        int value = data[index];
        int bIndex = value/10;

        int rowRegion = row/1024;
        int colRegion = col/1024;

        int numBox = size/1024;

        int binRegion = colRegion + rowRegion * numBox;
        bIndex += binRegion*18;

        atomicAdd(&histogram[bIndex],1);
    }
}
"""
```

Above is a screenshot of the naive kernel implemented in CUDA. It is also implemented the same way in OpenCL. The naive kernel was run with a configuration consisting of a block size of 32x32 and grid size of matrixSize/32 x matrixSize/32 with matrixSize being either 2^10, 2^13, or 2^15.

The kernel gets the appropriate row and column and calculates the 1D index that it corresponds to on the array called **data.** Then, the value using this index is retrieved from **global memory** and divided by 10 to find the right bin on the histogram from 0 to 17. C already rounds down since we are converting to an int.

Then, my approach to calculating a 1D 1x18 histogram vector for each 2^10x2^10 subregion was to allocate a 1 x 18*(matrixSize/2^10) histogram output vector that would contain all of the subregion histograms.

This brings me to the next part where I calculate exactly where in this very large output histogram vector OR which position and which histogram vector does the kernel increment. That is done by finding which row and column this data is located on the **data** array if it was indexed 2-dimensionally in boxes of sizes 1024x1024 (subregion). After finding the appropriate rowRegion and colRegion, I then proceed to find the number of total boxes which is the **size** argument or also the matrixSize, divided by 1024. Finally, the bin index of where to increment on the **histogram** argument can be calculated by colRegion + rowRegion * numBox added to the original index found earlier in the kernel. A global atomic add is used to increment that index in the global array **histogram.**

Optimized Implementation

```
kernel_opt_template = """
#include <stdio.h>
#include <math.h>

__global__ void optimizeHisto(int *data,int* globalHisto,int size)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;


    __shared__ unsigned int localHisto[18];

    for(int i = 0;i<18;i++){
        localHisto[i] = 0;
    }
    __syncthreads();

    if(col<size && row<size){
        int index = col + row * size;
        int value = data[index];
        int bIndex = value/10;
        atomicAdd(&localHisto[bIndex],1);
    }
    __syncthreads();


    //index into right 18 bin set

    if(threadIdx.x < 18 && threadIdx.y==0){

        int rowRegion = row/1024;
        int colRegion = col/1024;

        int numBox = size/1024;
        int binRegion = colRegion + rowRegion * numBox;
        int gIndex = threadIdx.x + binRegion*18;

        atomicAdd(&globalHisto[gIndex],localHisto[threadIdx.x]);
    }
}
"""
```

Above is a screenshot of the optimized kernel implemented in CUDA. It is also implemented the same way in OpenCL. The arguments passed into the kernel were a input data array, globalHistogram array, and size which just gave the row/column length of the matrix.

Like in the naive kernel, I find what row and column the thread is in. The optimization here mainly consists of relying on shared memory. As stated before, I had blocks of 32x32 threads. Thus, each block contained a shared histogram that the threads updated in shared memory instead of having to go back and forth to global memory. Although there is still a global memory access to retrieve the value in the global data array, the atomic add operation only interacts with this histogram in shared memory.

After all threads in the block have finished this first stage, the second stage is to basically update the global histogram with all of these shared memory histograms. This is done by conditioning that only the first 18 threads of each block are allowed to access the histogram, which is a. These 18 threads use its own index to retrieve the appropriate value from the shared histogram. Like the previous kernel, I am writing to a global histogram kernel that consists of multiple 1x18 histogram vectors concatenated. Thus, indexing must be used to find the appropriate 1x18 vector within this global histogram kernel. This is done exactly the same way as in naive kernel.

<u>Comparing</u>

Custom Print Time
Time Taken by Kernels:

	small	medium	large
python	0.0773771	4.50626	70.7308
naive	0.000198126	0.00026083	0.000351906
opt	0.000226974	0.000140905	0.000114202

Custom Print Speedup
Speedup(Naive/Optimized):

small_image	medium_image	large_image
0.872899	1.8511	3.08142

```
Custom Histogram Equal
Size 2^10x2^10
All the histograms are equal!!
Size 2^13x2^13
All the histograms are equal!!
Size 2^15x2^15
All the histograms are equal!!
```

CUDA:
Above is a screenshot of the CustomPrintTime, CustomPrintSpeedUp, and CustomHistogramEqual functions.

To begin, the CustomHistogramEqual block shows that for the different array sizes, the python, naive, and optimized arrays were all equal! The naive and optimized kernels work!

For the CustomPrintTime:

1. Python: the sequential time goes up as the array size increases.
2. Naive: the naive time also goes up as the array size increases.
3. Optimized: Interestingly, the optimized kernel time goes down as the array size increases. Must have been a good kernel.

For CustomPrintSpeedup:

1. Small Image: We don't observe a speedup as the optimized time actually takes longer than the naive time. However, this is probably because the optimized kernel needs a large enough data set to exploit.
2. Medium Image: We do observe a speedup of 1.8511 better for the optimized kernel.
3. Large Image: We observe an even greater speedup of 3.08142 better for the optimized kernel !

```
--------------------------------------------------------------------------------
Custom Print Time
Time Taken by Kernels:
```

	small	medium	large
python	0.0766201	4.50269	70.7001
naive	0.000207901	0.00019002	0.000174046
opt	0.000152111	0.000178099	0.000203848

```
--------------------------------------------------------------------------------
Custom Print Speedup
Speedup(Naive/Optimized):
```

small_image	medium_image	large_image
1.36677	1.06693	0.853801

```
--------------------------------------------------------------------------------
Custom Histogram Equal
Size 2^10x2^10
All the histograms are equal!!
Size 2^13x2^13
All the histograms are equal!!
Size 2^15x2^15
All the histograms are equal!!
--------------------------------------------------------------------------------
```

OpenCL:

Above is a screenshot of the CustomPrintTime, CustomPrintSpeedUp, and CustomHistogramEqual functions.

To begin, the CustomHistogramEqual block shows that for the different array sizes, the python, naive, and optimized arrays were all equal! The naive and optimized kernels work!

For the CustomPrintTime:

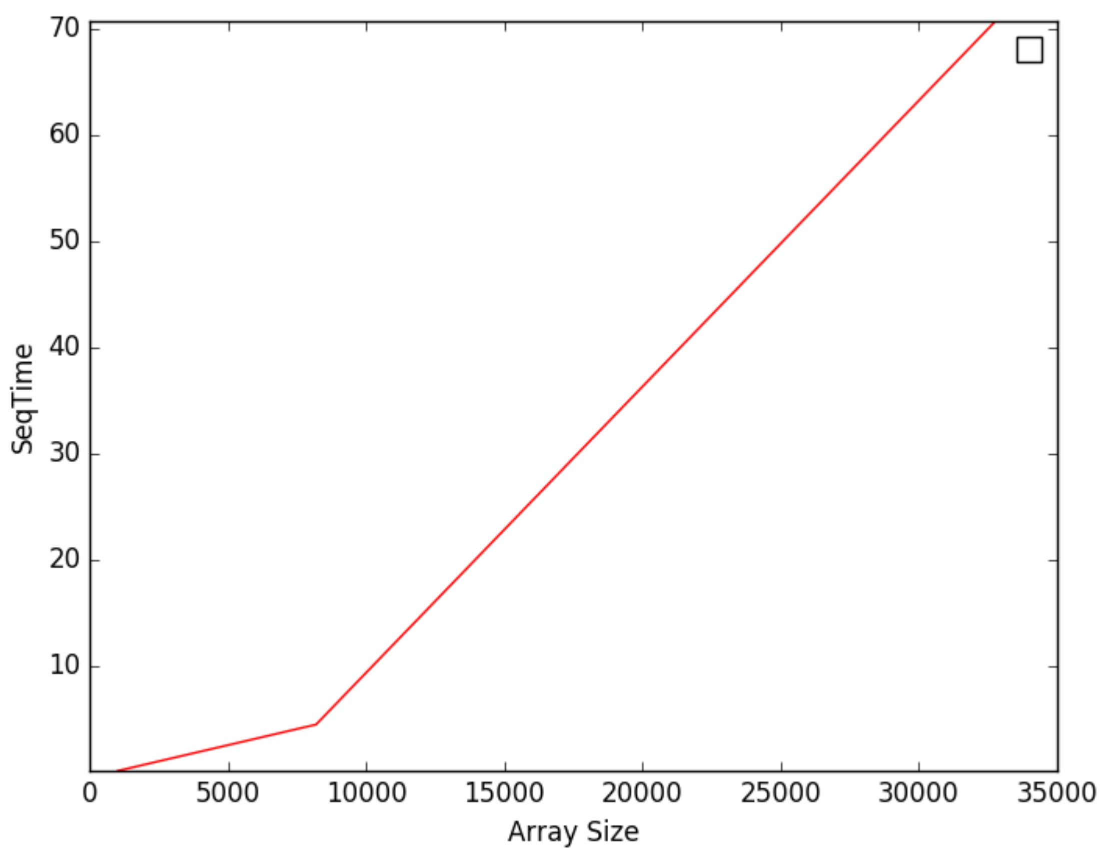
1. Python: the sequential time goes up as the array size increases.
2. Naive: the naive time interestingly goes down as the array size increases.
3. Optimized: The optimized kernel time goes up as size increases.

For CustomPrintSpeedup:

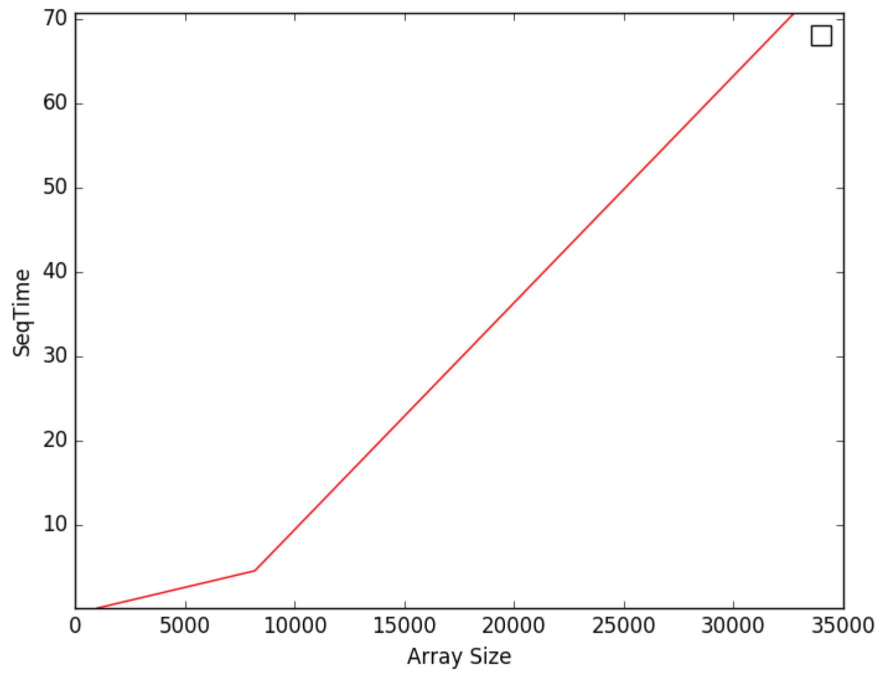
1. Small Image: We observe a speed up of 1.3677 times better for the optimized kernel.
2. Medium Image: We do observe a speedup of 1.06993 better for the optimized kernel.
3. Large Image: Unfortunately, we do not observe a speedup for the optimized kernel. This is interesting because in CUDA we did observe a speedup for the large image. This could mean that CUDA is a faster and more optimized software and thus a better choice for this GPU.

Overall, I saw that the optimized kernel was faster than the naive kernel for the medium and large image for CUDA and the small and medium image for OpenCL.

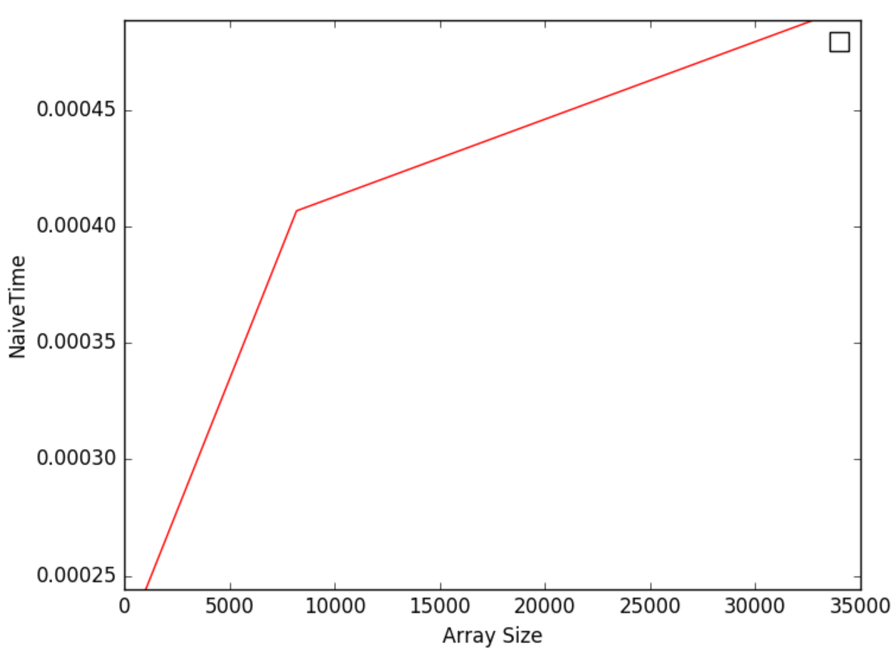
Graphs



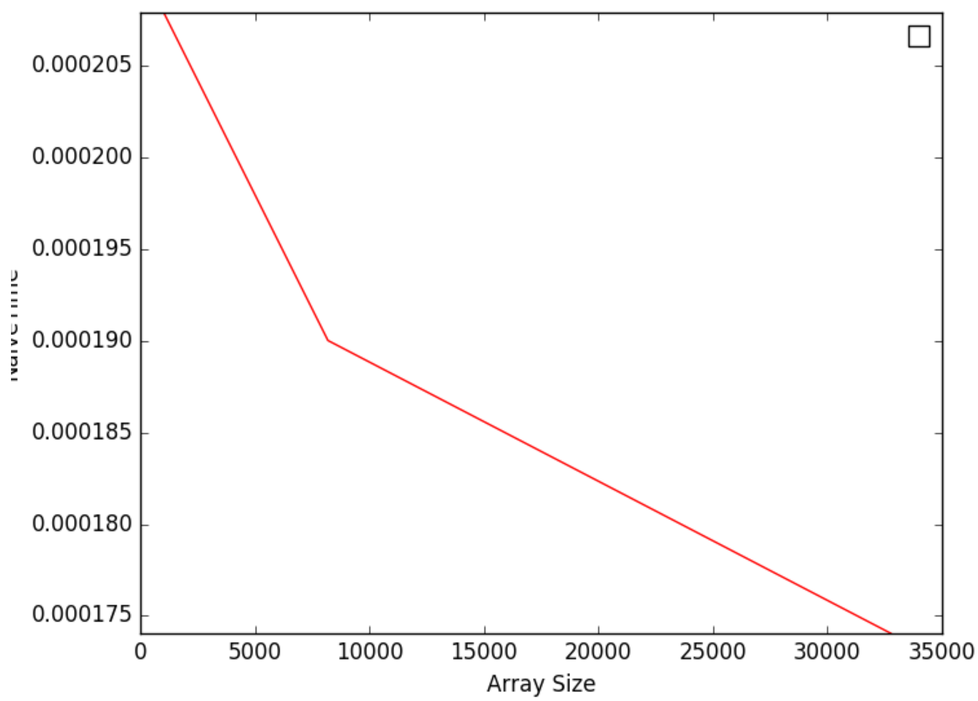
Sequential CUDA:



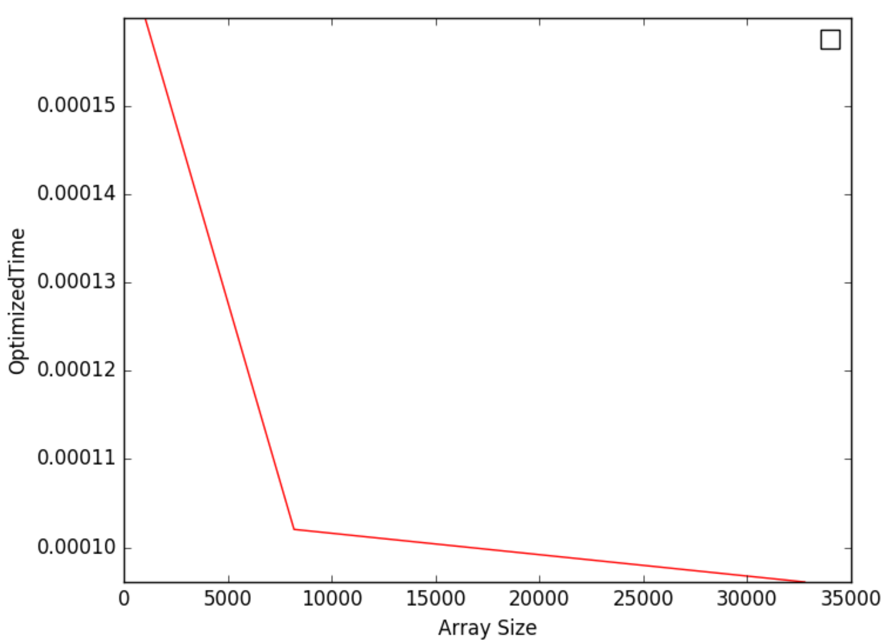
Sequential OpenCL:



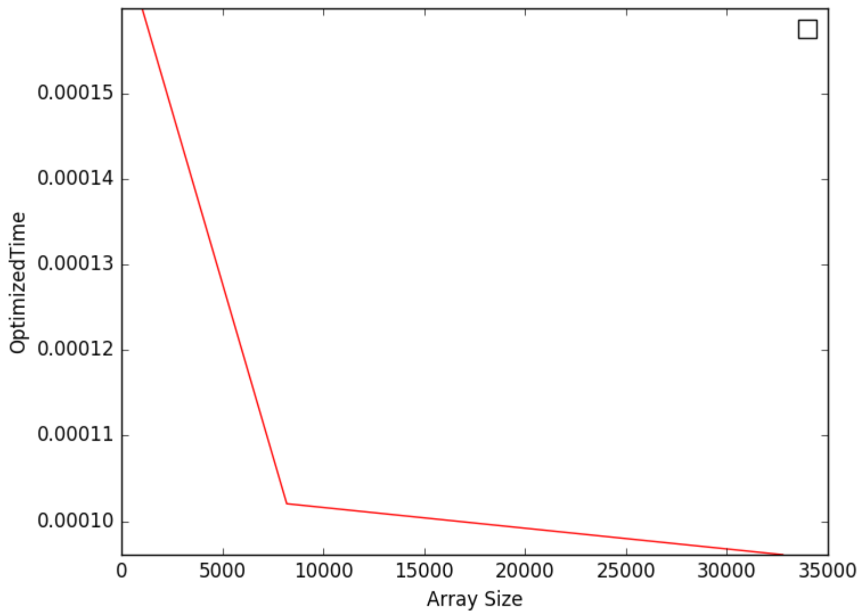
Naive CUDA:



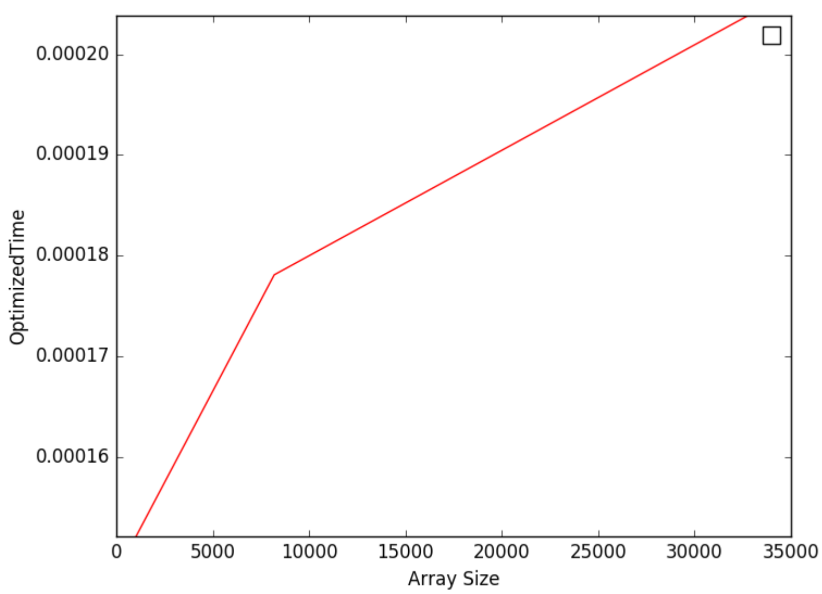
Naive OpenCL:



Optimized CUDA:



Optimized OpenCL:



CUDA Profiling:

Unfortunately, nvvp was extremely slow and could not detect the GPU, so I ran the command

```
sbatch --gres=gpu:1 --time=5 --wrap="nvprof ./histCuda2.py"
```

and the result was:

```
==29175== NVPROF is profiling process 29175, command: python ./histCuda2.py
CUDA
Sequential 2^10x2^10
--------------------------------------------------------------------------------
Sequential 2^13x2^13
==29175== Profiling application: python ./histCuda2.py
==29175== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 82.82%  3.15948s         3  1.05316s  1.6352ms  3.04807s  [CUDA memcpy HtoD]
  9.46%  361.05ms         3  120.35ms  436.23us  339.78ms  naiveHisto
  7.72%  294.33ms         3  98.109ms  275.33us  276.76ms  optimizeHisto
  0.00%  13.280us         6  2.2130us  1.7280us  2.9440us  fill

==29175== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 77.87%  3.16237s         3  1.05412s  2.1164ms  3.05005s  cuMemcpyHtoD
  7.21%  292.76ms         3  97.587ms  88.456us  292.50ms  cuModuleUnload
  5.88%  238.75ms         1  238.75ms  238.75ms  238.75ms  cuCtxCreate
  4.51%  183.03ms         1  183.03ms  183.03ms  183.03ms  cuCtxDetach
  4.10%  166.56ms         3  55.519ms  426.72us  165.63ms  cuModuleLoadDataEx
  0.26%  10.366ms         9  1.1518ms  26.201us  7.4909ms  cuMemAlloc
  0.16%  6.5500ms         9  727.78us  12.783us  4.9881ms  cuMemFree
  0.02%  733.02us        12  61.084us  29.363us  162.96us  cuLaunchKernel
  0.00%  43.504us        12  3.6250us  2.0180us  9.2610us  cuFuncSetBlockShape
  0.00%  29.892us        13  2.2990us     524ns  12.374us  cuCtxPopCurrent
  0.00%  28.333us        13  2.1790us     500ns  7.0530us  cuCtxPushCurrent
  0.00%  26.889us        34     790ns     322ns  2.5490us  cuDeviceGetAttribute
  0.00%  20.714us        13  1.5930us     927ns  4.6390us  cuCtxGetDevice
  0.00%  13.807us        21     657ns     346ns  1.5600us  cuDeviceComputeCapability
  0.00%  10.787us         3  3.5950us  2.9200us  4.9290us  cuModuleGetFunction
  0.00%  4.2530us         3  1.4170us     297ns  2.7820us  cuDeviceGetCount
  0.00%  2.8140us         3     938ns     560ns  1.1490us  cuDeviceGet
```

As seen from the screenshot, the naiveHisto and optimizeHisto are the kernel functions. The naiveHisto kernel function took longer time than the optimizeHisto kernel.