# Project 4: GPU Optimizations

Zachary Taylor (zctaylor@ncsu.edu)

December 4, 2016

## 1 Performance Issues

The original application had a few different performance problems. Norm would run a nested for loop over a data structure and calculate the same sum for each block. Using this sum, the output was modified based on it's location in the array.

The nested for loop was the first problem, performing more instructions than were necssary. The application also did not make use of shared memory or memory coalescing. This means that values from the input array were read multiple times and if the value was already calcuated it was not cached for future calls. Additionally the warps of the cuda cluster were not optimized, achieving only 74.4% occupancy of the available threads.

Just to summarize, the following performance issues were noticed:

- Poor loop performance (too many instructions)
- No shared memory usage or caching
- No memory coalesing
- Poor warp performance

## 2 Optimizations

To optimize the applicaton, two of the four categories were used. With the most effort put into utilizing shared memory.

### 2.1 Optimization 1: Reducing calls

Optimization 1 reduced and reorganized the operations so one for loop was unrolled. This is apart of the Minimize the number of instructions/operations type of performance enhancement. Additionally the multiplication was reduced so it only had to be applied once to a series of additions instead to each addition.

## 2.2 Optimization 2: Using shared memory

Optimization 2 was using shared memory to not perform duplicate calculations. Shared memory was used in stages. Stage 1 loaded the multiplciation data into each thread for it's memory location.

Listing 1: 'Stage 1 of Shared Memory'

```
int i = threadIdx.x;
mulData[i] = mul[i];

__syncthreads();
```

Stage 2 performed and stored multiplication once per thread to a shared memory location.

Listing 2: 'Stage 2 of Share Memory'

```
for (int j = 0; j < BLOCK_SIZE; j++) {
      mySum += in[start + i*width + j] * mulData[j];
}

inData[i] = mySum;
__syncthreads();
```

Finally in stage 3 the shared memory data was summed to provide the total.

Listing 3: 'Stage 3 of Shared Memory'

```
float total = 0.0f;

for (int j = 0; j < BLOCK_SIZE; j++) {
  total += inData[j];
}

__syncthreads();
```

After that the writes to out were performed as usual. This is apart of hte memory optimizations type.

## 2.3 Optimization 3: Using shfl

Optimization 3 was using shfl to sum. Fold can be done in two ways, Kepler introduced functions called __shfl that performs better than standard fold. This optimization reduces instructions on kepler systems and also doesn't use shared memory. However on Fermi the shfl is writen using shared memory. This also performs better but not quite as well as the __shfl on Kepler. The source code was written to optimize the shfl based on the architecture. Fold is a little of both, an instruction optimizer and a shared memory user.

Listing 4: 'Opt 3: Shfl'

```
for (int offset = warpSize/4; offset > 0; offset /= 2)
{
        mySum += __shfl_down(mySum, offset);
}
```

# 3 Performance

For each of the hardware platform, report the execution times of the original GPU kernel and every version of the kernel that you have optimized for that hardware.

## 3.1 GTX480

### 3.1.1 Norm

```
[temp553@compute-0-49 csc512-project4]\$ ./norm
kernel time 7.970432s
results checking passed!
```

### 3.1.2 Optimization 1

```
[temp553@compute-0-82 csc512-project4]\$ ./opt1
kernel time 7.381248s
results checking passed!
```

### 3.1.3 Optimization 2

```
[temp553@compute-0-49 csc512-project4]\$ ./opt2
kernel time 5.037600s
results checking passed!
```

### 3.1.4 Optimization 3

```
[temp553@compute-0-82 csc512-project4]\$ ./opt3
kernel time 5.078176s
results checking passed!
```

### 3.1.5 Optimization All

```
[temp553@compute-0-82 csc512-project4]\$ ./optAll
kernel time 2.876192s
results checking passed!
```

## 3.2 GTX780

Note: Had to use my local GTX 780 TI due to ARC jobs never coming back

### 3.2.1 Norm

```
PS C:\Users\Zach\Classes\csc512\projects\csc512-project4> .\x64\
    Debug\norm.exe
kernel time 42.817665s
results checking passed!
```

### 3.2.2 Optimization 1

```
PS C:\Users\Zach\Classes\csc512\projects\csc512-project4> .\x64\
    Debug\opt1.exe
kernel time 27.906784s
results checking passed!
```

#### 3.2.3 Optimization 2

```
PS C:\Users\Zach\Classes\csc512\projects\csc512-project4> .\x64\
    Debug\opt2.exe
kernel time 7.825248s
results checking passed!
```

#### 3.2.4 Optimization 3

```
PS C:\Users\Zach\Classes\csc512\projects\csc512-project4> .\x64\
    Debug\opt3.exe
kernel time 7.356032s
results checking passed!
```

#### 3.2.5 Optimization All

```
PS C:\Users\Zach\Classes\csc512\projects\csc512-project4> .\x64\
    Debug\optAll.exe
kernel time 5.222944s
results checking passed!
```

## 4 Experience on Tool

The tool has some direct links to Nvidia's documentation so it served a general purpose of helping to track down official sources. The style of the tool is hard to consume however. I feel like finding information in it is a little overwhelming. I used many sources to perform my work in optimizing the appplication but the tool wasn't heaavily relied upon. In the future; adding examples/explaining examples may serve a better purpose than linking.