# Implementation of Sobel Filter using GPU programming Assignment #5, CSC 746, Fall 2022

Sanket Naik*
SFSU

**ABSTRACT**

Sobel Filtering is an edge detection algorithm which can be used for edge detection. The algorithm simply performs a convolution between a pair of 3x3 kernels and a part of the image. This emphasizes the regions of high spatial frequency and hence helps us to find the edges in an image. In this assignment, we study the performance of the Sobel filter program with 3 different programming methods. We use the OpenMP library to run the code in parallel on a CPU and use CUDA to run the code on a CORI GPU node. We also use OpenMP's device offload to use OpenMP to run the code on the GPU. We measure the runtime for all programs and SM Efficiency for all GPU related programs and compare our metrics to form a better understanding of massively parallel GPU code.

## 1 INTRODUCTION

The goal of this assignment is to study the implementation of a Sobel filter which is an edge detection algorithm using different GPU programming methods. We use CUDA and OpenMP with device offload with various parameters and compare our results with a CPU-only program which makes use of OpenMP. Finally, we compare the performance of the CUDA implementation with different parameters and compare the best implementation with OpenMP with device offload.
The problem being studied

- We are trying to understand the performance of a Sobel filter algorithm on a CPU and on the GPU using different programming methods.
- We make use of the chrono library and nvprof to obtain runtime and SM (Streaming Multiprocessor) efficiency metrics that we will use to study the performance of the algorithm with different parameters.

The approach used for studying this problem

- We implement 3 programs which run the Sobel filter using 3 different methods as follows - 1) CPU parallel using OpenMP, 2) GPU parallel using CUDA, 3) OpenMP parallel with device offload.
- We use the chrono library to measure the runtime of the CPU code and nvprof to measure metrics such as runtime and SM efficiency of the GPU code.
- We measure the speedup achieved when we run the CPU code in parallel.
- We compare the different variations of the GPU implementation for different number of blocks and block sizes.
- We compare the best GPU implementation to the OpenMP with device offload implementation.

Main Results

- We find that the GPU code is much faster than the CPU parallel code or the OpenMP-offload code.
- We find that the runtime of the GPU code improves as we increase the number of blocks and the number of threads per block.
- We find that the SM Efficiency of the GPU code improves when the number of blocks increases but stays relatively the same when the number of threads per block increases.
- We find that the OpenMP-offload code is much slower than the GPU code which may be because of the increased overhead of transferring data from the CPU to the GPU.

## 2 IMPLEMENTATION

Three programs were implemented in this assignment to apply a Sobel filter over an image to perform edge detection. The programs implemented were - (1) CPU parallel code with OpenMP, (2) GPU parallel code with CUDA and (3) OpenMP parallel code with device offload. Each program calls the **sobel_filtered_pixel()** function which performs the convolution of the sobel-x and sobel-y kernels and one pixel. The pseudo code of the convolution code was as shown below in Listing 1. This code remains the same between all three programs and the only difference is the addition of the **__device__** keyword for the CUDA version.

```
float sobel_filtered_pixel(float *s, int i, int j,
      int ncols, int nrows, float *gx, float *gy)
{
  Gx, Gy = 0.0f;

  if (0 > i > nrows - 1 && 0 > j > ncols - 1)
    for k = 0 to 3
      for l = 0 to 3
        currentPixel = s(i+k-1, j+l-1);
        Gx += gx(k,l) * currentPixel;
        Gy += gx(k,l) * currentPixel;
      }
    }
  }
  return sqrt(Gx*Gx + Gy*Gy);
}
```

Listing 1: Sobel Filtered Pixel: Performs convolution of the sobel-x and sobel-y kernels and one pixel

### 2.1 Part 1 - Sobel Filter on CPU with OpenMP Parallelism

This program implements the sobel filter using OpenMP for parallelizing the problem. The pixel values for the image were stored in the column-major order which changes the way we access

*email:snaik@sfsu.edu

the elements. Moreover, we use the OpenMP pragmas to specify which code block has to run in parallel on different threads. In this case we use the collapse clause which essentially merges the to loops of size N into one large loop of size NxN and which then gets divided according to the schedule clause. Since we do no specify the schedule clause, OpenMP will use the default static schedule with an iteration chunk size of 1. In our case, OpenMP will parallelize application of the sobel filter over a single pixel by running the **sobel_filtered_pixel()** function over different pixels.

The logic of the program was as shown below in Listing 2.

```
16 void do_sobel_filtering(float *in, float *out, int
       ncols, int nrows)
17 {
18    float Gx[] = {1.0, 0.0, -1.0, 2.0, 0.0, -2.0,
       1.0, 0.0, -1.0};
19    float Gy[] = {1.0, 2.0, 1.0, 0.0, 0.0, 0.0,
       -1.0, -2.0, -1.0};

21    #pragma omp parallel for collapse(2)
22    for (int i = 0; i < nrows; i++){
23       for (int j = 0; j < ncols; j++){
24          out[i*ncols + j] = sobel_filtered_pixel(
       in, i, j, ncols, nrows, Gx, Gy);
25       }
26    }
27 }
28 }
```

Listing 2: Sobel Filter with OpenMP Parallelism: Performs the sobel

### 2.2 Part 2 - Sobel Filter on the GPU using CUDA

This program uses the CUDA to implement the sobel filter. CUDA is parallel computing platform and model from Nvidia which can be used to run massively parallel programs on a GPU. We use the **__global__** and **__device__** keywords to specify which blocks of code have to run in parallel. We also make use of the following CUDA provided keywords

- **threadIdx.x**: Index of the current thread within its block.
- **blockIdx.x**: Index of the current thread block in the grid.
- **blockDim.x**: Number of threads in a block
- **gridDim.x**: Number of blocks in the grid.

We use these keywords to calculate the index and the stride which we then use to loop over all of the pixels in the image. This allows us to parallelize the operation over each pixel and our concurrency is determined by the number of threads in each block and the total number of blocks.

The logic of the program was as shown below in Listing 3.

### 2.3 Part 3 - Sobel Filter with OpenMP offload to the GPU

This program is similar to the CPU parallel program but instead of running the sobel operation on the CPU threads, we run it on the GPU. We make use of several OpenMP pragmas in this program and some of them are as follows -

- **target**: Specify the device that the code should run on (in this case, the GPU).

```
30 __global__ void
31 sobel_kernel_gpu(float *s,  // source image pixels
32       float *d,  // dst image pixels
33       int n,  // size of image cols*rows,
34       int nrows,
35       int ncols,
36       float *gx, float *gy) // gx and gy are
       stencil weights for the sobel filter
37 {

39    int index = blockIdx.x * blockDim.x +
       threadIdx.x;
40    int stride = blockDim.x * gridDim.x;
41    for (int a = index; a < n; a += stride) {
42      int i = a / ncols;
43      int j = a % ncols;
44      d[a] = sobel_filtered_pixel(s, i, j, ncols,
       nrows, gx, gy);
45   }

47 }
```

Listing 3: Sobel Filter on GPU using CUDA: Uses CUDA to parallelize the sobel operation of each pixel

- **map**: We use map to send and recieve a variable from the device. *to* is used when we want to send the variable to the device. *tofrom* is used when we want to send the variable to the device and get it back after execution.
- **teams**: Used to create multiple teams of threads.
- **distribute**: Used to distribute the work among the different teams.

The program was as shown in Listing 4.

## 3 Evaluation

The programs were evaluated by running with different levels of concurrency and different metrics such as the runtime and the SM efficiency were noted. These metrics were used to compare the different programs.

The CPU code was compiled and run on a KNL node on CORI and the runtimes for different levels of concurrency were recorded. Each KNL node on CORI has an Intel Xeon Phi Processor 7250 which has 68 cores and 272 threads (4 threads per core) and each core runs at a clock rate of 1.4GHz. It has 64 kB of L1 cache per core and 1MB of L2 cache which is shared between 2 cores. The processor has 16 GB of MCDRAM with a bandiwidth of >460 GB/s which serves as the last level cache and each node has 96 GB of DDR4 Memory with a memory bandwidth of 102 GiB/s. Each core has a peak performance of 38.6 GFlops [2].

The GPU code and OpenMP code with device offload was compiled and run on a GPU node. We note the runtimes and the SM efficiency for different number of threads per block and different number of blocks for the GPU code. Each GPU node on CORI has two sockets of 20-core Intel Xeon Gold 6148 (Skylake) processor which has 20 cores and 40 threads (2 threads per core) and each core runs at a clock rate of 2.40GHz. It has 1.25 MB of L1 cache, 20 MB of L2 cache and 27.5 MB of L3 cache. Each node has 384 GB of DDR memory and 8 NVIDIA Tesla V100 (Volta) GPUs. For all of our runs, we only use 1 GPU and 1/8th of the cores and memory available in each node [1].

```
48 void
49 do_sobel_filtering(float *in, float *out, int
       ncols, int nrows)
50 {
51     float Gx[] = {1.0, 0.0, -1.0, 2.0, 0.0, -2.0,
        1.0, 0.0, -1.0};
52     float Gy[] = {1.0, 2.0, 1.0, 0.0, 0.0, 0.0,
        -1.0, -2.0, -1.0};

54     off_t out_indx = 0;
55     int width, height, nvals;

57     width=ncols;
58     height=nrows;
59     nvals=width*height;

61     #pragma omp target data map(to:in[0:nvals]) map
        (to:width) map(to:height) map(to:Gx[0:9]) map(
        to:Gy[0:9]) map(tofrom:out[0:nvals])
62     {
63       #pragma omp target teams distribute parallel
       for collapse(2)
64       for (int i = 0; i < nrows; i++){
65          for (int j = 0; j < ncols; j++){
66             out[i*ncols + j] = sobel_filtered_pixel
       (in, i, j, ncols, nrows, Gx, Gy);
67          }
68       }
69     } // pragma omp target data
70 }
```

Listing 4: Sobel Filter with OpenMP offload to the GPU: Uses OpenMP to offload the sobel operation on the gpu using groups of CPU threads

The Clang++ compiler (13.0.0) and CUDA compiler (NVIDIA 11.2.142) were used to compile all programs. All nodes run on GNU Linux.

### 3.1 Methodology

The CPU program with OpenMP with compiled and run on the KNL node. The GPU program and OpenMP with device offload program were compiled and run on the GPU node along with nvprof and a metric argument for the SM Efficiency metric.

Performance metrics: We mainly look at two metrics (1) runtime and (2) SM Efficiency.

- **Runtime**: This allows us to measure the speedup achieved and allows us to compare the performance of two programs. In case of the CPU program and OpenMP device offload program, we use the chrono library to find the runtime of our program. For the GPU program, we use nvprof to find the runtime of the program.
- **SM Efficiency**: The SM Efficiency refers to the percentage of time that an SM (Streaming Multiprocessor) is active. It basically tells us how 'busy' an SM is when we run the program and a higher SM Efficiency will translate to better performance because we are making better use of the GPU.

Experimental design:

The CPU program was run on the following levels of concurrency - [1, 2, 4, 8, 16] and we note the runtime for each run.

The GPU program was run for the following values for number of threads per block - [32, 64, 128, 256, 512, 1024] and the following values for number of blocks - [1, 4, 16, 256, 1024, 4096] for each combination, we run the program using nvprof and note the runtime. We also run the program with an additional argument which asks for the SM Efficiency metric.

| Num. Threads | Runtime (ms) | Speedup |
|---|---|---|
| 1 | 3347.7 | 1 |
| 2 | 1718.43 | 1.95 |
| 4 | 892.618 | 3.75 |
| 8 | 478.755 | 6.99 |
| 16 | 270.63 | 12.37 |

Table 1: Runtime metric for the CPU program at different levels of concurrency

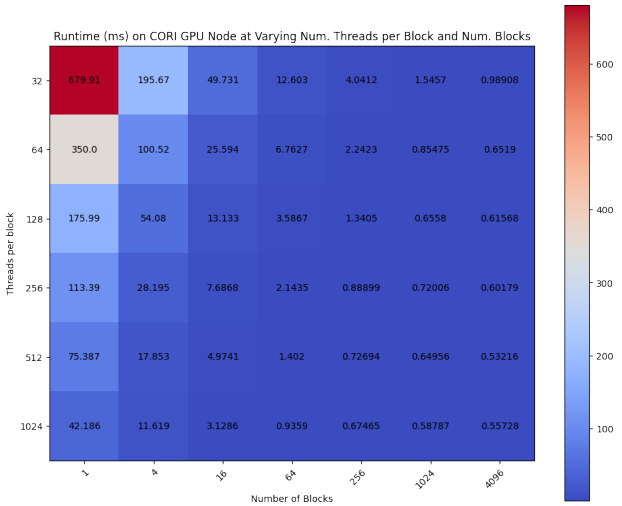


Figure 1: A heatmap showing the runtimes for all combinations of Number of Blocks and Number of Threads per Block for the GPU code.

### 3.2 Scaling study of CPU/OpenMP code

The CPU parallel code was run using the concurrency levels described in 3.1 and we note the runtime reported by the chrono library. Table 1 shows the runtime in milliseconds for the different levels of concurrency.

Based on the runtime we can also achieve the speedup for each level of concurrency. The runtime and the speedup for each level of concurrency was as shown in Table 1.

We can see from the table that the program scales well but does not scale linearly as there is a gap between the observed speed up and the ideal speedup.

### 3.3 GPU-CUDA Performance Study

The GPU code was run using the arguments specified in 3.1 and the runtime and SM Efficiency metrics were noted. Based on these metrics, we can generate heatmaps which show metrics and the efficiency.

Based on these heatmaps, we can see that the runtime for a higher number of blocks is better than the runtime tends to improve as the number of threads per block and the number of blocks increases. The most major improvements are observed when we go from a very low number of blocks or number of threads per block to a higher number (very low concurrency to a slightly higher concurrency).

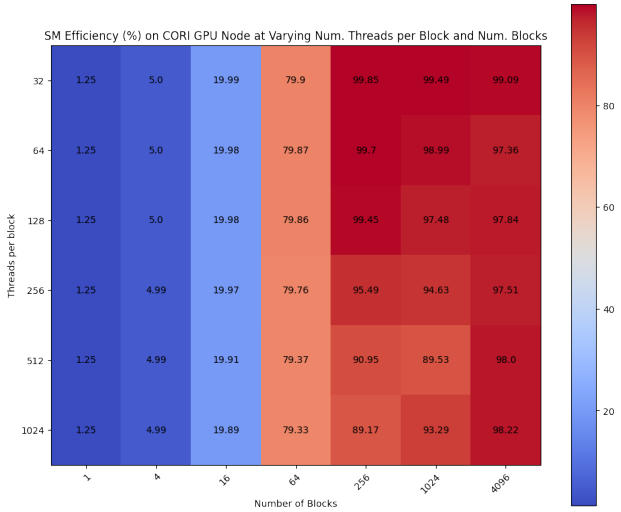


Figure 2: A heatmap showing the SM Efficiency for all combinations of Number of Blocks and Number of Threads per Block for the GPU code.

| Program | Runtime (ms) | SM Efficiency (%) |
|---|---|---|
| GPU | 0.53216 | 98 |
| OpenMP-offload | 460.078 | 99.46 |

Table 2: Runtime and SM Efficiency Comparison of GPU code and OpenMP-offload code

For example, when going from 32 threads per block to 64 threads per block and keeping the same block size, the runtime generally gets halved. Similarly, when going from 1 block to 4 blocks and keeping the same number of threads, the runtime was almost 1/4th the previous runtime. We do not see such improvements for higher concurrency levels which may be due to more overhead of moving data around.

The SM Efficiency metric indicates how busy an SM is when our code is running. Generally, a higher efficiency will translate into better performance because we will be making better use of each SM. We can see that we achieve higher SM efficiency as the number of blocks increases but we do not see any change in SM efficiency as the number of threads per block increases. When we look at the NVIDIA Tesla V100 GPU architecture, we can see that each Volta Streaming Multiprocessor is partitioned into 4 processing blocks [3]. This explains why increasing the number of blocks increases the SM Efficiency but increasing the number of threads per block does not impact the SM Efficiency at all.

### 3.4 OpenMP-offload Study

We only have a single datapoint for the OpenMP-offload program but we can compare its performance with the best performing GPU program - 4096 Blocks and 512 Threads per Block to understand how well the OpenMP-offload program performs. Table 2 shows the comparison between the GPU code and the OpenMP-offload code.

We can see that the runtime of the OpenMP-offload cure is significantly higher than that of the GPU code but it makes slightly better use of the SM. The higher runtime may be becuase of the overhead involved in moving the data to and from the GPU. We also have to consider that for the GPU program, we are running all of the code on the GPU so the overhead of transferring data is much lower.

## REFERENCES

[1] NERSC. Gpu compute nodes specification on cori. https://docs-dev.nersc.gov/cgpu/hardware/, 2021.

[2] NERSC. Knl compute nodes specification on cori. https://docs.nersc.gov/systems/cori/knl-compute-nodes, 2021.

[3] NVIDIA. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.