GPU Programming
Assignment #5, CSC 746, Fall 2021

Arulselvan Madhavan*
SFSU

ABSTRACT

This paper aims to study the effects of parallelizing a computational problem using GPU(Graphics Processing Units). GPUs architectures lend itself well for parallelization, and this has opened up the opportunity to do GPGPU programming(General Purpose GPU). In this paper, we take a well known problem - applying Sobel filter on images for edge detection, and run it on a GPU using two different techniques - an architecture specific compiler and using OpenMP [2]. The results show that GPU code significantly outperform CPU code. And between the GPU implementations, the architecture specific implementation provides better runtime at the expense of platform portability.

1 INTRODUCTION

The main objective of this work is to study the effects of GPU programming on well known computation problems by leveraging the additional computational hardware that GPUs provide. The problem studied in this work is Sobel's algorithm for detecting edges in an image. Sobel's edge detection algorithm was implemented and studied in both CPU and GPU(Nvidia [3]). The CPU version was parallelized using OpenMP [2] and the GPU version was implemented using two methodologies - CUDA [4] and using OpenMP's device offload feature. The primary reason for using two different techniques to leverage GPU resources was to compare and contrast the effort that would be needed to turn a CPU program into a GPU program, and how much resource utilization each method offers right out of the box without having to tweak for performance. The GPU and CPU implementations were run with different number of threads, to find the best configuration for the implementation. In my experiments, GPU implementations significantly outperform CPU implementation, and between the GPU implementations, the CUDA implementation outperformed OpenMP device offload. However the OpenMP device offload significantly is smaller and provided the best bang for the buck while maintaining platform portability of the code.

2 IMPLEMENTATION

There were three implementations of the Sobel edge detection algorithm.

- CPU with OpenMP parallelization
- GPU with CUDA
- GPU with OpenMP device offload

2.1 Sobel Filter on CPU with OpenMP Parallelism

The CPU implementation applies Sobel's edge detection filter to a 2D image of arbitrary dimensions. The CPU implementation provides the baseline for the other implementations. The CPU implementation was parallelized using OpenMP. The implementation is parallelized over both rows and columns of the image. The parallelization pragmas is shown in Listing:1. The kernel that convolves sobel's filter on the image is shown in Listing:2. The border pixels that can't be convolved with the sobel filters are set of zero. The sobel filters used are shown in Listing:3. This is the standard sobel 2D filter shown here for convenience.

```
1  void do_sobel_filtering(float *in, float *out, int
       dims[2]) {

3  #pragma omp parallel default(none) shared(in, out,
       dims, Gx, Gy)
4    {
5  #pragma omp for collapse(2)
6      for (int y = 1; y < dims[1] - 1; y++) {
7        for (int x = 1; x < dims[0] - 1; x++) {
8          int outIdx = y * dims[0] + x;
9          out[outIdx] = sobel_filtered_pixel(in, x,
       y, dims, Gx, Gy);
10       }
11     }
12   }
13 }
```

Listing 1: Sobel implementation CPU - OpenMP

```
14 float sobel_filtered_pixel(float *in, int x, int y
       , int dims[], float *gx,
15                            float *gy) {
16   int gi = 0;
17   float sum = 0.0;
18   float gytemp = 0.0;
19   float gxtemp = 0.0;
20   for (int ky = -1; ky < 2; ky++) {
21     for (int kx = -1; kx < 2; kx++) {
22       int cxIdx = x + kx;
23       int cyIdx = y + ky;
24       int cyPos = cyIdx * dims[0];
25       int cxy = cyPos + cxIdx;
26       float sxy = in[cxy];
27       gxtemp += gx[gi] * sxy;
28       gytemp += gy[gi] * sxy;
29       gi++;
30     }
31   }
32   return sqrt((gxtemp * gxtemp) + (gytemp * gytemp
       ));
33 }
```

Listing 2: Sobel Kernel implementation

2.2 Sobel Filter on the GPU using CUDA

The aim of this implementation is to take the Sobel kernel and run it on GPU. CUDA was chosen since it's the recommended way to

*email:amadhavan@sfsu.edu

```
// Rows * columns
float Gx[] = {
  1.0, 0.0, -1.0,
  2.0, 0.0, -2.0,
  1.0, 0.0, -1.0
};
float Gy[] = {
  1.0, 2.0, 1.0,
  0.0, 0.0, 0.0,
  -1.0, -2.0, -1.0
};
```

Listing 3: Sobel Filters

target Nvidia GPU architectures. The CUDA implemenation of the CPU kernel is shown in Listing:4. The number of kernels launched were controlled with two environment variables. One for configuring the number of blocks and other for the number of threads each block contained. The grim and the blocks were 1D. Number of Blocks tried were 1, 4, 16, 64, 256, 1024, 4096. Number of threads within each block were tried for 32, 64, 128, 256, 512, 1024. Total kernels launched was blocks*threads. Each pixel was given one of these kernels to calculate its pixel value from applying the sobel filter.

```
__global__ void sobel_kernel_gpu(
    float *s, // source image pixels
    float *d, // dst image pixels
    int n,    // size of image cols*rows,
    int rows, int cols, float *gx,
    float *gy) // gx and gy are stencil weights
    for the sobel filter
{
  int index = blockIdx.x * blockDim.x + threadIdx.
    x;
  int stride = gridDim.x * blockDim.x;
  for (int xy = index; xy < n; xy += stride) {
    int y = xy / cols;
    int x = xy - y * cols;
    int outIdx = y * cols + x;
    if (y == 0 || y == rows - 1 || x == 0 || x ==
    cols - 1) {
      d[outIdx] = 0
    } else {
      d[outIdx] = sobel_filtered_pixel(s, x, y,
    rows, cols, gx, gy);
    }
  }
}
```

Listing 4: Sobel GPU - Cuda

2.3 Sobel Filter with OpenMP offload to the GPU

The aim of this implementation is to using OpenMP pragmas to generate the GPU implementation of the Sobel edge detection algorithm. The pragmas used are shown in Listing:5. The code looks very similar to the CPU implementation and the only addition are the pragmas that control data movement and parallelization.

3 Evaluation

The aforementioned implementations were run on a 2D image of size: 7112x5146. The experiments were run on the Cori cluster. The CPUs used were Intel's Xeon Gold 6148(skylake) Processor. The GPU used was Nvidia's Tesla V100. The specifications of the Cori Xeon Gold processors are given in section 3.1. Only one GPU was used to run in the experiments.

3.1 Computational platform and Software Environment

The specifications were taken from `https://docs-dev.nersc.gov/cgpu/hardware/` CPU specs:

- CPU: Intel Xeon Gold 6148 Processor
- CPU clock frequency(base): 2.40 GHz
- CPU clock frequency(turbo): 3.70GHZ
- Number of cores used: 10
- 27.5 MB of L3 Cache
- 768 GB of MCDRAM (spread over 8 channels)
- 384 GB DDR4 memory
- OS: SUSE Linux Enterprise Server 15
- OpenMP version: 201511

GPU Specs:

- GPU: NVIDIA Tesla V100 ('Volta')
- Compute Capability: 7.1
- 15.7 Single Precision teraFLOPS
- Streaming Multiprocessors: 80
- Number of FP32 cores: 5120

Compiler specs:

- Clang version: 13rc3
- Optimization: O3
- nvcc(for CUDA)

Profiler specs:

- nvprof

3.2 Methodology

The experiments were run with the intention of finding the implementation that offers the best runtime and resource utilization. The CPU code was measured for its runtime using a timing library called chrono timer. The GPU code was profiled using nvprof [5]. The profiler was configured to provide data on two metrics - runtime and sm efficiency(Streaming multiprocessor usage efficiency). nvprof was run twice on the same code to collect the metrics separately. CPU code that was parallelized using OpenMP was configured to run on threads 1, 2, 4, 8, 12, 16. The GPU CUDA code was tried on a number of block and thread configurations. Number of Blocks tried were 1, 4, 16, 64, 256, 1024, 4096. Number of threads within each block were tried for 32, 64, 128, 256, 512, 1024. The OpenMP device offload version just used the default settings for the architecture.

3.3 Scaling study CPU code with OpenMP parallelization

The result of running the Sobel filter kernel in CPU with OpenMP parallelization is shown in Figure:1. As the parallelization increased, the runtime decreased steadily. There is a bump in runtime for P=12. In my Serval WS - AMD Ryzen CPU which has 12 cores, P=12 performed the best outperforming all other options. But the Intel Skylake processor in Cori performed worse on 12. So, the bump could be explained by the difference in processor architectures. As number of threads increased, OpenMP had more cores to move the work around. This explains the decrease in runtime as number of cores increased.

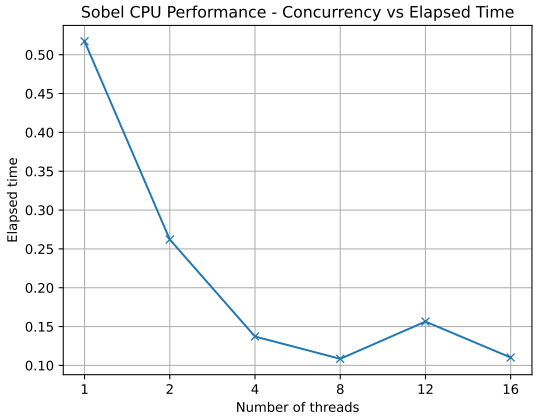

Figure 1: Sobel CPU - Parallelization Factor vs Runtime

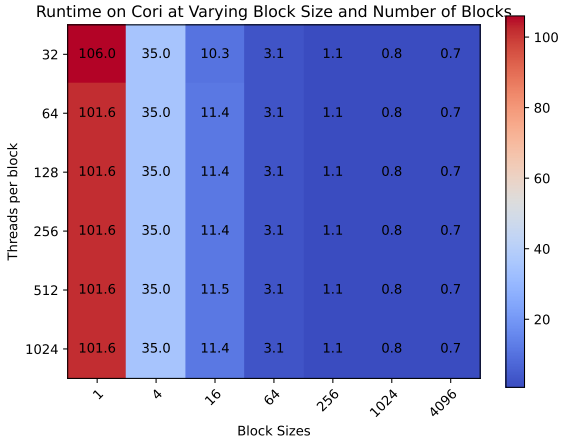

Figure 2: Sobel GPU - CUDA - Runtime(ms)

3.4 Sobel GPU - CUDA

The results of the CUDA implementation for different block size and threads is shown in figure:**??**. As the number of blocks and number of threads the runtime continued to decrease. The best runtime was found for blocks = 4096 and threads = 1024. Figure:3 shows the efficiency with which the SM(Streaming Multiprocessors) were used. As expected, the efficiency of the streaming multiprocessor usage goes up as the number of blocks and threads goes up. The maximum efficiency achieved was around 98 percent. The runtime and improved efficiency is seen on higher number of blocks and threads because V100 has around 5120 single precision cores which can be leveraged in parallel. GPU offers zero cost context switching between blocks. In addition to all of this, the algorithm lends itself well to coalesced memory access.

3.5 Sobel GPU - OpenMP-offload

The results from running the OpenMP device offload implementation from 2.3 is shown in comparison with the performing CUDA configuration in Table:**??**tab:cudaomp. The OpenMP version is slower but shows high efficiency. The cuda compiler outperforms the OpenMP compilation that targets Nvidia architectures. This isn't surprising as the architecture specific compiler has a lot more freedom to introduce optimizations that the generic OpenMP version doesn't.

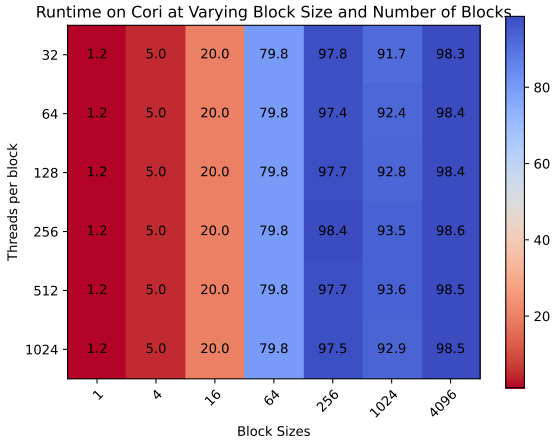

Figure 3: Sobel GPU - CUDA - Efficiency

Impl	Runtime(ms)	Efficiency(Percentage)
CUDA(Be=4096; T=1024)	0.7	98.6
OMP device offload	1.94	98.8

Table 1: Open MP device offload vs CUDA

REFERENCES

[1] Cori specifications - https://docs.nersc.gov/systems/cori/
[2] OpenMP specification - https://www.openmp.org/specifications/
[3] Nvidia - https://www.nvidia.com/en-gb/graphics-cards/
[4] CUDA - https://developer.nvidia.com/cuda-toolkit
[5] Nvprof - https://docs.nvidia.com/cuda/profiler-users-guide/index.html

```
65 void do_sobel_filtering(float *in, float *out, int
       dims[2]) {
66   float Gx[] = {1.0, 0.0, -1.0, 2.0, 0.0, -2.0,
       1.0, 0.0, -1.0};
67   float Gy[] = {1.0, 2.0, 1.0, 0.0, 0.0, 0.0,
       -1.0, -2.0, -1.0};

69   off_t out_indx = 0;
70   int width, height, nvals;

72   width = dims[0];
73   height = dims[1];
74   nvals = width * height;

76 #pragma omp target data map(to
                                \
77                             : in [0:nvals], width,
       height, Gx [0:9], Gy [0:9]) \
78     map(tofrom
                                \
79         : out [0:nvals])
80   {
81 #pragma omp target teams distribute parallel for
       collapse(2)
82     for (int y = 1; y < height - 1; y++) {
83       for (int x = 1; x < width - 1; x++) {
84         int idx = y * width + x;

86         int gi = 0;
87         float sum = 0.0;
88         float gytemp = 0.0;
89         float gxtemp = 0.0;
90         for (int ky = -1; ky < 2; ky++) {
91           for (int kx = -1; kx < 2; kx++) {
92             int cxIdx = x + kx;
93             int cyIdx = y + ky;
94             int cyPos = cyIdx * width;
95             int cxy = cyPos + cxIdx;
96             float sxy = in[cxy];
97             gxtemp += Gx[gi] * sxy;
98             gytemp += Gy[gi] * sxy;
99             gi++;
100          }
101        }
102        out[idx] = sqrt((gxtemp * gxtemp) + (
       gytemp * gytemp));
103      }
104    }
105  }
106 }
```

Listing 5: Sobel GPU - OpenMP