Assignment #5, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

ABSTRACT

We explored GPU programming by implementing a Sobel filter in CUDA and using OpenMP with device offload. We also implemented a CPU version in order to compare the performance. We tested different configurations of number of blocks and number of threads per block on CUDA and compared the performance.

1 INTRODUCTION

In this assignment, we applied Sobel filter on an image by using stencil to compute gradients of the image. We compared the performance between CPU-OpenMP, GPU-CUDA and GPU-OpenMP-offload to explore GPU. We used chorono timer and SM-efficiency to evaluate the performance. We tested different configuration of block dimension and thread dimension and find out the best performed configuration. We also compared the best performed GPU-CUDA version with GPU-OpenMP-offload and found that GPU-OpenMP-offload performed the faster.

2 IMPLEMENTATION

We implemented a Sobel filter which takes an input photo (2D array), apply convolution on each pixel and output the photo.

2.1 Program 1 (CPU)

The Sobel filter is implemented on CPU with OpenMP parallelism (see Listing 1).

2.2 Program 2 (GPU with CUDA)

The Sobel filter is implemented on GPU with CUDA (see Listing 2). The sobel_filtered_pixel() function is the same with program 1.

2.3 Program 3 (OpenMP offload to the GPU)

The Sobel filter is implemented by using OpenMP offload to the GPU (see Listing 3). It maps input image and other variables to the GPU and did the computation at GPU. The sobel_filtered_pixel() function is the same with program 1.

```
float
sobel_filtered_pixel(float *s, int i, int j , int dims[], float *gx, float *gy)
{
   float Gx=0.0;
   float Gy=0.0;
   int cols = dims[0];
   int rows = dims[1];
   int index = 0;
   for (int x = i-1; x < i+2; x++) {
      for (int y = j-1; y < j+2; y++) {
         if (x >= 0 && x < rows && y >= 0 && y < cols) {
            Gx += s[x*cols+y] * gx[index];
            Gy += s[x*cols+y] * gy[index];
         }
         index++;
      }
   }
   return sqrt(Gx*Gx + Gy*Gy);
}

void do_sobel_filtering(float *in, float *out, int dims[2])
{
   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};
   int cols = dims[0];
   int rows = dims[1];
   #pragma omp parallel for collapse(2)
   for (int i=0; i<rows; i++) {
      for (int j=0; j<cols; j++) {
         out[i*cols+j] = sobel_filtered_pixel(in, i, j , dims, Gx, Gy);
      }
   }
}
```

Listing 1: record the run time of Program 1 (basic VMM)

3 RESULTS

3.1 Computational platform and Software Environment

All tests were running on Cori KNL node. Program 1 was running on CPU nodes and program 2 and 3 was running on GPU nodes. The processor was Intel Xeon Phi Processor 7250. The clock rate was 1.4 GHz. The size of L1 cache was 64 KB. The size of L2 cache was 1MB. The memory size was 96 GB for DDR4 and 16 GB for MCDRAM. The GPU node on Cori has 80 SM, 64 warps per SM, 32 threads per warp in double precision. Cmake version was version 3.14.4. GCC version was version 8.3.0. Optimization level was O3 by default. [1]

*email:zliu15@mail.sfsu.edu

3.2 Methodology

We first checked the validity of program 1, 2 and 3 by comparing the output image with the correct output image (figure 1). The dimension of the input image is 7112 * 5146 (col * row).

There are 2 metrics that we measured: run time and SM efficiency (GPU only).

We record the run time among all programs and configurations to indicate the performance. In program 1, we used chrono timer to record the run time. We ran 5 tests per concurrency to get the mean run time for each concurrency. In program 2 and 3, we used nvprof to get the run time.

For program 1, we tested on a KNL node at different concurrency levels P = 1, 2, 4, 8, 16. For program 2, we tested on thread block size B = 1, 4, 16, 64, 256, 1024, 4096 and number of threads T = 32, 64, 128, 256, 512, 1024.

By default, OpenMP statically assigns loop iterations to threads.

```
__global__ void
sobel_kernel_gpu(float *s, float *d, int n, int
    rows, int cols, float *gx, float *gy)
{
      int index = blockIdx.x * blockDim.x +
    threadIdx.x;
      int stride = blockDim.x * gridDim.x;
      for (int i=index; i<rows; i+=stride) {
          for (int j=0; j<cols; j++) {
              d[i*cols+j] = sobel_filtered_pixel(s
    , i, j, rows, cols, gx, gy);
          }
      }
}
```

Listing 2: record the run time of Program 2 (OpenMP)

```
#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])
{
      #pragma omp target teams distribute parallel
     for collapse(2)
      for (int i=0; i<height; i++) {
         for (int j=0; j<width; j++) {
            out[i*width+j] = sobel_filtered_pixel(
    in, i, j, width, height, Gx, Gy);
          }
      }
  }
}
```

Listing 3: record the run time of program 3 (CBLAS)

Concurrency	Runtime
1	0.1778
2	0.1559
4	0.1389
8	0.1343
16	0.1328

Table 1: Run time decreased as concurrency went higher.

3.3 Scaling study of CPU/OpenMP

Table 1 shows that the run time decreased as the concurrency goes larger. We ran 5 tests per concurrency to get the mean run time for each concurrency. Although the concurrency level accelerated the speedup, there's still some contention on the same cache line which impede the speedup.

3.4 GPU-CUDA performance study

Figure 2 shows the heat map of 2 factors which may influence the run time: block size (B) and threads per block (T). As we can see from the figure, the worst configuration is B = 1 and T = 32, and the best configurations are around higher B and lower T area (e.g., B = 4096 and T = 32). The best one is B = 1024 and T = 64.

Figure 3 shows the heat map of the SM efficiency. It is consistent with figure 1 that the worst configuration in figure 2 had the lowest SM efficiency (1.25%) whereas the best configurations in figure 2 had the highest SM efficiency (88.07%).

The reason that B = 1 and T = 32 performed the worst may because that it's the least paralleled configuration so it has least threads that can run concurrently.

One interesting observation is that B = 4096 and T = 1024 performed worse than B = 1024 and T = 64 which has less threads. It

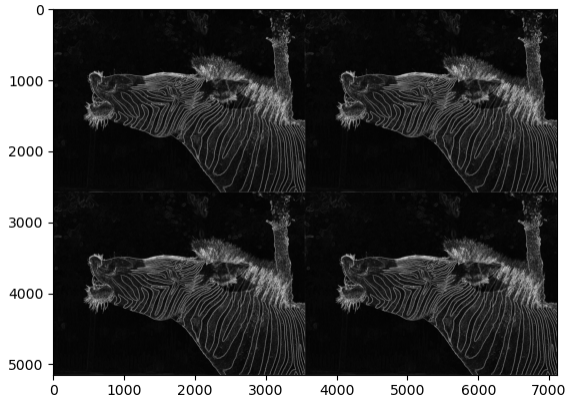

Figure 1: The outputs of all programs are the same with the correct output.

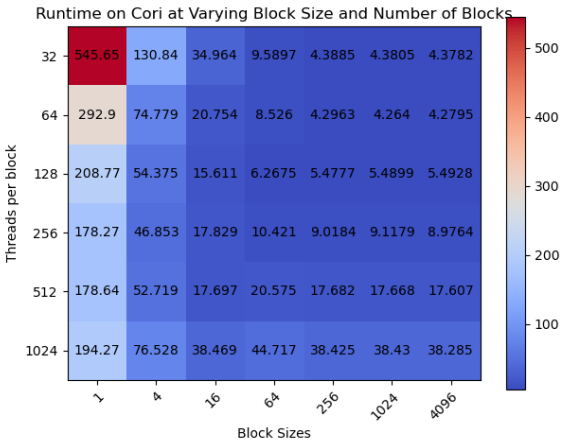

Figure 2: Larger block sizes accelerate the speed. This figure was produced by the program `plot.py` file.

	CUDA	OpenMP offload
run time	4.264	2.893
SM efficiency	87.26	99.76

Table 2: OpenMP offload is faster and more efficiently using SM than CUDA.

may because that there should be 1024 threads run on the same SM, since the input image has 5146 rows, there are only 5 SM are utilized (5146/1024 is approximately 5). Since there are 80 SM in GPU, the SM efficiency is only 6.35% (i.e., $(5146/1024)/80 = 6\%$). The configuration of B = 1024 and T = 64 utilized the most of the SM since the the thread per block is lower.

3.5 OpenMP-offload study

Table 2 shows the comparison between 2 ways that utilized GPU: CUDA and OpenMP offload. OpenMP offload is faster and used SM more efficiently than CUDA. It may because that in OpenMP offload, we specified target teams so that each threads is in its own contention group. Threads in the same warp in CUDA share memories so there was contentions which made it slower than OpenMP offload.

References

[1] https://docs.nersc.gov/systems/cori/. Cori system specification.

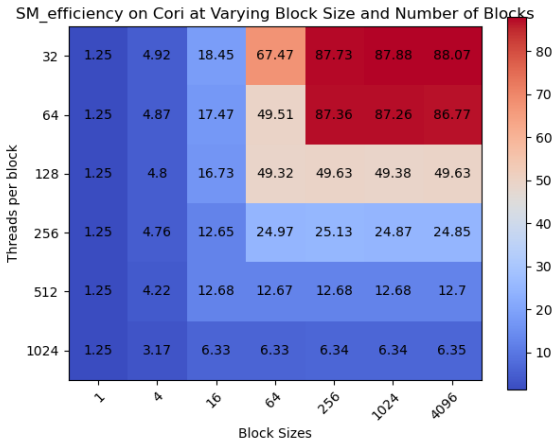

Figure 3: . This figure was produced by the program `plot.py` file.