Study of GPU Stencil Operations with CUDA and OpenMP Offload performance.
Assignment #5, CSC 746, Fall 2024

Ruxue Jin*
SFSU

ABSTRACT

This study evaluates the performance of Sobel filter stencil operations across three parallel implementations: CPU-OpenMP, GPU-CUDA, and CPU-OpenMP-Offload, measuring metrics including runtime, GPU occupancy, and memory bandwidth utilization. The research systematically explores various combinations of thread block counts and threads per block for the GPU-CUDA implementation. Analysis reveals that while CPU multi-threading shows significant speedup, the GPU-CUDA implementation achieves optimal performance with 256 threads per block and 4096 blocks, outperforming the OpenMP-Offload approach.

1 INTRODUCTION

The Sobel filter is an edge detection algorithm that employs stencil operations through convolutional multiplication across image pixels. This computationally intensive operation, which requires processing each pixel based on its neighborhood, is inherently suitable for parallel execution on Graphics Processing Units (GPUs).

This study evaluates and compares the performance characteristics of three parallel implementations: a CPU-based OpenMP implementation and two GPU-based approaches (CUDA and OpenMP offload). For the CPU-OpenMP implementation, we analyze runtime and speedup metrics across varying thread counts. For the GPU-CUDA implementations are examined by systematically adjusting two key parameters: the number of thread blocks and threads per block. Performance analysis utilizes NVIDIA's NCU (NVIDIA Compute Profiler) to collect critical metrics including GPU runtime, achieved occupancy percentage, and memory bandwidth utilization.

Our analysis reveals that the CPU-OpenMP implementation demonstrates near-linear scaling, with runtime approximately halving as thread count doubles. In the GPU implementations, configurations with limited parallelism (1-4 blocks or 32 threads per block) exhibit poor performance due to insufficient GPU resource utilization. The optimal configuration utilizes 4096 blocks with 256 threads per block, significantly outperforming the OpenMP offload implementation, which shows comparable performance to the configuration using 64 blocks with 1024 threads.

2 IMPLEMENTATION

2.1 Sobel Filtering with CPU/OpenMP

The function `sobel_filtered_pixel` applies the Sobel filter to a given pixel location (i, j). It takes several parameters: `image`, a source 2D image represented by the 1D array; `i, j`, the pixel location where the filter will be applied; `ncols, nrows`, the dimensions of the input and output image buffers; and `gx, gy`, arrays of length 9 containing the Sobel filter weights in the x- and y-directions, respectively.

*email:rjin@sfsu.edu

The nested `for` loops iterate over the neighborhood by using offsets -1 to 1 for both rows and columns. For each neighboring pixel, it checks if the indices are within the image bounds. If valid, it multiplies the pixel's value by the corresponding filter weight from `gx` and `gy`, accumulating results in `Gx` and `Gy`. The gradient magnitude, `G`, is then calculated as $\sqrt{Gx^2 + Gy^2}$ and returned. The codes are shown in Listing 1.

```
sobel_filtered_pixel(image,i,j,ncols,nrows,Gx,Gy):

    Gx_sum = Gy_sum = 0
    index = 0

    // Process 3x3 Sobel kernels
    For di in [-1,0,1]:
      For dj in [-1,0,1]:
        if (i+di) and (j+dj) are within bounds:
          pixel = image[(i+di)*ncols + (j+dj)]
          Gx_sum += pixel * Gx[index]
          Gy_sum += pixel * Gy[index]
        index++

    Return sqrt(Gx_sum^2 + Gy_sum^2)
}
```

Listing 1: sobel_filtered_pixel:perform the sobel filtering at a given i,j location.

The algorithm `do_sobel_filtering` iterates over all input image pixels and invoke the sobel_filtered_pixel() function at each (i,j) location of input image `in`.

The function defines Sobel kernels `Gx` and `Gy`, then uses `#pragma omp parallel for collapse(2)` to parallelize the nested loop over pixels (i, j). For each pixel, `sobel_filtered_pixel` computes the horizontal and vertical gradients, and the resulting gradient magnitude is stored in `out`. The codes are shown in Listing 2.

2.2 Sobel Filtering with GPU-CUDA

The function `sobel_filtered_pixel` in GPU-CUDA is identical to the one shown in Listing 1, with the addition of the `__device__` qualifier to indicate that the code is executed on the device (GPU in this case).

The function `sobel_kernel_gpu` Listing 3 is decorated with the `__global__` qualifier, indicating it is a CUDA kernel that executes on the GPU and can be invoked from host (CPU). It assigns unique thread IDs using CUDA built-in variables, where `threadIdx.x + blockIdx.x * blockDim.x` calculates the initial 1D index for each thread. The `stride` (`blockDim.x` $\times$ `gridDim.x`) represents both the total number of parallel threads and the distance between elements processed by each thread. Using a grid-stride loop pattern, threads process multiple pixels by incrementing their index by `stride`. Each 1D index is mapped to 2D image

```
do_sobel_filtering(in, out, ncols, nrows):
  // Initialize Gx and Gy
  Gx = [1.0, 0.0, -1.0, 2.0, 0.0, -2.0, 1.0, 0.0,
    -1.0]
  Gy = [1.0, 2.0, 1.0, 0.0, 0.0, 0.0, -1.0, -2.0,
    -1.0]

  // Apply parallelization to nested loops for all
     pixels
  #pragma omp parallel for collapse(2)
  for i from 0 to nrows - 1:
    for j from 0 to ncols - 1:
      // Compute the Sobel filter result for the
    pixel at (i, j)
      out[i * ncols + j] = sobel_filtered_pixel(in
    , i, j, ncols, nrows, Gx, Gy)
```

Listing 2: sobel_filtering_cpu: iterate over all input image pixels and invoke the sobel_filtered_pixel() function at each (i,j) location

coordinates (`row = i ÷ ncols`, `col = i % ncols`), and the `sobel_filtered_pixel` function is applied to each valid pixel within the image boundaries.

```
__global__
sobel_kernel_gpu(source_image, destination_image,
    n, nrows, ncols, gx, gy):
    // Calculate thread index and stride
   idx = blockIdx.x * blockDim.x + threadIdx.x
   stride = blockDim.x * gridDim.x

   // Process pixels in grid-stride loop
   for (i = idx; i < n; i += stride):
       row = i / ncols
       col = i % ncols
       if (row < nrows && col < ncols):
           dst[i] = sobel_filtered_pixel(src, row,
     col, ncols, nrows, gx, gy)
```

Listing 3: sobel_kernel_gpu: GPU kernel implementation that applies Sobel filter across the image using grid-stride loop pattern

The main function Listing 4 first reads the input image file and allocates unified memory for both input and output data, converting input bytes to floating-point values (0-1 range). Then, it initializes Sobel filter kernels (Gx, Gy) and prefetches all data to GPU memory for processing. Next, it configures GPU execution parameters based on command-line arguments or default values and launches the Sobel kernel. Finally, it converts the processed output back to bytes (0-255 range) and writes the result to an output file.

2.3 Sobel Filtering with OpenMP-offload

The function `sobel_filtered_pixel` in OpenMP-offload is identical to the one shown in Listing 1.

In the function `sobel_filtering_offload` Listing 5,`#pragma omp target data` manages data transfer between host and device, using `map(to ...)` to send input data (image array and Sobel kernels) to the device, and `map(from ...)` to retrieve the processed output array. Then, `#pragma omp target teams distribute parallel for collapse(2)` enables parallel execution of the nested loops, where `collapse(2)` merges two loops into a single parallel region, thereby optimizing workload distribution and enabling concurrent pixel processing.

```
main(argc, argv):
    // 1. Load image and prepare GPU memory
    nvalues = width * height
    in_data_bytes = read_input_file(input_fname,
    nvalues)
    in_data_floats = cudaMallocManaged(nvalues)
    out_data_floats = cudaMallocManaged(nvalues)
    convert_bytes_to_floats(in_data_bytes to
    in_data_floats)  // Scale: [0,255] to [0,1]

    // Initialize and copy Sobel kernels to GPU
    Gx, Gy = initialize_sobel_kernels()
    device_gx, device_gy = copy_to_gpu(Gx, Gy)

    // Prefetch data to GPU memory
    cudaMemPrefetchAsync(in/out_data,
    device_kernels)

    // Configure and launch kernel
    set_kernel_configuration(argc, argv)
    launch_sobel_kernel<<<nBlocks,
    nThreadsPerBlock>>>
    cudaDeviceSynchronize()

    // Process output and save
    convert_and_save_output(out_data_floats)    //
     [0,1] to [0,255]
```

Listing 4: sobel_kernel_gpu_main: iterate through source array, calling the sobel_filtered_pixel function at each location

3 Result

This section describes the testing environment for the three programs and outlines the performance metrics used in their evaluation: CPU runtime, GPU runtime, achieved occupancy on the GPU, and percentage of peak sustained memory bandwidth.

3.1 Computational platform and Software Environment

The experiments were conducted on a NERSC Perlmutter [1] compute node equipped with both CPU and GPU capabilities. The CPU configuration consists of dual AMD EPYC 7763 (Milan) processors operating at a base frequency of 2.45 GHz with boost capability up to 3.5 GHz. The system features 512 GB of DDR4 memory and a three-level cache hierarchy: 64 KB L1, 512 KB L2 per core, and 256 MB shared L3 cache. The GPU computations were performed on an NVIDIA A100-PCIE-40GB accelerator with CUDA 12.2 support. The software environment included Ubuntu 22.04 operating system with Linux kernel 5.15, and code compilation was done using GNU C++ compiler version 7.5.0.

3.2 Methodology

For the CPU-OpenMP algorithm, the paper evaluates performance with different thread configurations: {1, 2, 4, 8, 16}, measuring elapsed time using `chrono::timer()`.

For the GPU-CUDA algorithm, various configurations of threads per block ({32, 64, 128, 256, 512, 1024}) and thread blocks ({1, 4, 16, 64, 256, 1024, 4096}) are tested. Performance is analyzed using the NVIDIA Nsight Compute (ncu) tool with the command `ncu --set default --section SourceCounters --metrics smsp__cycles_active.avg.pct_of_peak_sustained_elapsed, dram__throughput.avg.pct_of_peak_sustained_elapsed, gpu__time_duration.avg --section Occupancy a.out <args>`. This command is focusing on three key metrics: `gpu__time_duration`,

```
sobel_filtering_offload(in, out, ncols, nrows):
    // Initialize Sobel kernels
    Gx = [1, 0, -1, 2, 0, -2, 1, 0, -1]  //
    Horizontal edge detection
    Gy = [1, 2, 1, 0, 0, 0, -1, -2, -1]  //
    Vertical edge detection

    nvals = ncols * nrows  // Total image size

    // Map data between host and device
    #pragma omp target data
        map(to: in[0:nvals], Gx[0:9], Gy[0:9])
    // Input data to device
        map(from: out[0:nvals])
     // Output data from device
    {
        // Parallel processing of image pixels
        #pragma omp target teams distribute
    parallel for collapse(2)
        for row = 1 to nrows-1:
            for col = 1 to ncols-1:
                out[row * ncols + col] =
    sobel_filtered_pixel(in, row, col, ncols,
    nrows, Gx, Gy)
    }
}
```

Listing 5: sobel_filtering_offload: Parallel Sobel filter using OpenMP offload

Number of Threads	Runtime (ms)	Speedup
1	0.383	1
2	0.194	1.97
4	0.098	3.92
8	0.052	7.42
16	0.027	14.01

Table 1: CPU-OpenMP parallel performance with varying thread counts.

which measures the total time taken by the GPU to execute the kernel; `Achieved Occupancy Percentage`, which reflects the efficiency of the kernel's use of available GPU resources by indicating the number of active warps relative to the maximum supported occupancy; and `dram__throughput.avg.pct_of_peak_sustained_elapsed`, which measures memory throughput as a percentage of the GPU's peak bandwidth, providing insight into memory access efficiency.

For the OpenMP-offload configuration, the `ncu` tool is used once to capture these same performance metrics, helping assess GPU utilization and memory throughput during the offloading process.

3.3 CPU-OpenMP: runtime for different threads

Performance was tested with 1, 2, 4, 8, and 16 threads, recording the runtime for each configuration. Speedup was calculated as the ratio of serial runtime to parallel runtime. Results show a clear trend: starting from a single thread, the runtime consistently decreases by roughly half with each doubling of threads. This proportional speedup indicates that the implementation effectively utilizes the available CPU cores, with each increase in thread count yielding nearly twice the performance.

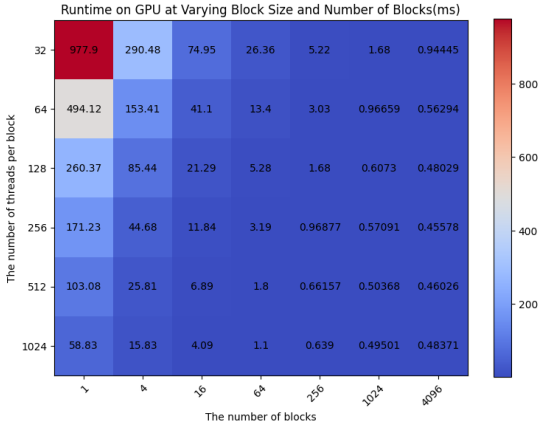

Figure 1: GPU-CUDA: gpu__time_duration

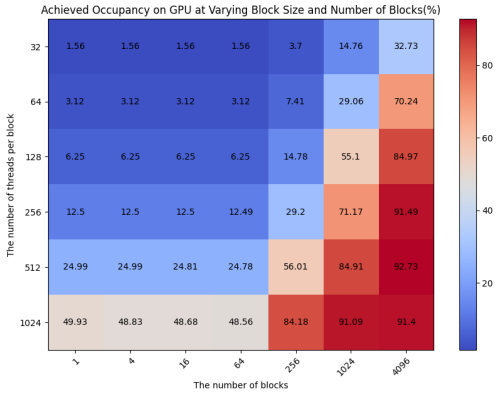

Figure 2: GPU-CUDA: Achieved Occupancy Percentage.

3.4 GPU-CUDA with various block number and threads number

Figures 1, 2, and 3 display heatmaps of the results for varying numbers of thread blocks (1, 4, 16, 64, 256, 1024, 4096) and threads per block (32, 64, 128, 256, 512, 1024). These heatmaps show the GPU time duration, achieved occupancy percentage, and memory bandwidth percentage.

Configurations with few threads (32-64) or few blocks (1-16) demonstrate poor performance, with the worst case being 1 block and 32 threads, resulting in a 977.9ms runtime, 1.56% occupancy, and 0.02% memory bandwidth utilization due to severe underutilization of GPU resources. The runtime improves proportionally (approximately 4X with 4X more blocks, 2X with 2X more threads) until reaching 256 blocks and 256 threads, after which the improvement rate diminishes. The optimal configuration utilizes 256 threads per block with 4096 blocks, achieving the best runtime (0.45578ms), highest memory bandwidth utilization (40.48%), and excellent occupancy (91.4%). Similar high performance is maintained across configurations using 256-1024 threads per block with 1024-4096 blocks, suggesting that once sufficient parallelism is achieved, additional increases in thread or block counts yield only marginal improvements.

3.5 OpenMP-Offload

Shown in the Table 2 Compared with the optimal GPU configuration of 4096 blocks and 256 threads per block, OpenMP-offload shows lower performance, with a longer runtime of 1.23 ms versus the

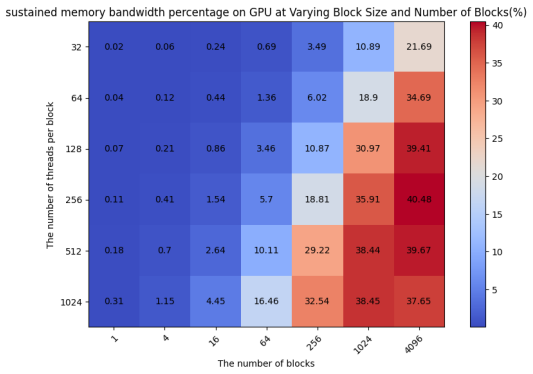

Figure 3: GPU-CUDA: Memory Bandwidth Percentage

program	Runtime(ms)	Achieved Occupancy%	memory bandwidth%
GPU(4096,256)	0.45578	91.49	40.48
OpenMP-Offload	1.23	68.55	14.83

Table 2: Comparison of performance optimal GPU and OpenMP-Offload

GPU's 0.45 ms. Additionally, the OpenMP-offload utilizes GPU resources less effectively, achieving only 68% occupancy compared to 91% with the GPU configuration. Memory bandwidth utilization is also lower for OpenMP-offload at 14.8% compared to 40/48% with GPU, indicating less efficient data transfer, which contributes to the performance difference.

ACKNOWLEDGMENTS

The author would like to acknowledge the assistance of ChatGPT, a language model developed by OpenAI, for helping to articulate the interpretation of our data and refine the techinical writing.

REFERENCES

[1] N. E. R. S. C. C. (NERSC). Perlmutter system overview. https://www.nersc.gov/users/systems/perlmutter/, 2024. Accessed: 2024-10-07.