GPU Stencil Operations with CUDA, OpenMP Offload Assignment #5, CSC 746, Fall 2024

Shun Usami[*]
SFSU

Abstract

This assignment explores the implementation and performance analysis of the Sobel filter on both CPU and GPU platforms. The Sobel filter, a widely used edge-detection algorithm in image processing, was tested using three parallelization strategies: a CPU-based approach with OpenMP as a CPU baseline, a GPU-optimized approach using CUDA to assess GPU performance, and an OpenMP offload implementation serving as a GPU baseline. Each method was evaluated on the Perlmutter supercomputer, with measurements taken for runtime, achieved occupancy, and memory bandwidth utilization. The results demonstrate CUDA's superior performance and highlight the highly parallelizable nature of both the GPU architecture and the Sobel filter algorithm.

1 Introduction

This assignment investigates the implementation and performance of the Sobel filter, an essential edge-detection algorithm in image processing, on both CPU and GPU platforms. Edge detection is a critical operation in computer vision applications, aiding in feature extraction, object recognition, and other image analysis tasks. Given the computationally intensive nature of the Sobel filter, this assignment seeks to explore how parallelization techniques can enhance its performance across different hardware architectures.

To study this problem, we developed three parallelized implementations of the Sobel filter. The first, a CPU-based approach using OpenMP, establishes a baseline for comparison. The second implementation uses CUDA to fully leverage the GPU's parallel processing capabilities and assess the efficiency of GPU-specific optimizations. The third approach, an OpenMP offload to the GPU, provides a simpler, portable method of GPU parallelization, enabling us to compare this with the CUDA approach. Each of these implementations is detailed in Section 2.

Our experiments, conducted on the Perlmutter supercomputer, measure runtime, achieved occupancy, and memory bandwidth utilization for each implementation. Results demonstrate CUDA's superior performance, revealing that optimized GPU processing offers significantly faster execution times than both CPU and OpenMP offload. These findings underscore the highly parallelizable nature of the Sobel filter and highlight the impact of platform-specific optimizations on computational efficiency.

2 Implementation

This section describes the implementation of the Sobel filter, a foundational edge-detection algorithm. To assess the effectiveness of parallelization and thread distribution strategies, we implemented three approaches to accelerate the Sobel filter. The subsections cover the core Sobel filter algorithm (Sobel Operator), followed by three parallelization techniques: a CPU-based implementation using OpenMP to establish a CPU baseline, a CUDA-based GPU implementation focused on optimizing performance through strided memory access, and an OpenMP offload to the GPU to serve as a GPU baseline.

2.1 Sobel Operator

The Sobel operator approximates the gradient of image intensity for edge detection [3]. It uses two 3×3 kernels, convolved with the original image A, to compute horizontal and vertical gradients G_x and G_y:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A, \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

The gradient magnitude at each pixel is computed as:

$$G = \sqrt{G_x^2 + G_y^2}$$

The function `sobel_filtered_pixel()` computes the Sobel filter at a pixel location, as shown in Listing 1. The arrays gx and gy represent the kernel weights for the horizontal and vertical filters, respectively.

```
Function sobel_filtered_pixel(img[width, height],
    x, y, gx[3, 3], gy[3, 3]) {
    if x or y is at the boundary of the img
        return 0
    Gx = 0.0, Gy = 0.0;
    for j in {0, 1, 2}
        for i in {0, 1, 2}
            xx = x - 1 + i;
            yy = y - 1 + j;
            Gx += img[xx, yy] * gx[i, j];
            Gy += img[xx, yy] * gy[i, j];
    return sqrt(Gx * Gx + Gy * Gy);
}
```

Listing 1: Sobel filtered pixel computation. Computes the Sobel filter at a specific pixel location.

2.2 CPU with OpenMP Parallelism

This implementation examines CPU parallelization of the Sobel filter using OpenMP, providing a baseline for GPU parallelization. OpenMP directives help distribute computations across multiple threads to improve performance.

The function `do_sobel_filtering()` applies the Sobel filter across the image, as shown in Listing 2. It iterates over all pixels, calling `sobel_filtered_pixel()` for each one. Adding `#pragma omp parallel for` before the outer loop parallelizes row processing, allowing threads to work concurrently on different image rows.

[*]email:susami@sfsu.edu

```
void do_sobel_filtering(float *in, float *out, int
     width, int height) {
  float gx[] = {...}, gy = {...};
#pragma omp parallel for
  for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
      out[x, y] = sobel_filtered_pixel(in, x, y,
    gx, gy);
    }
  }}
```

Listing 2: OpenMP implementation of Sobel filtering. `do_sobel_filtering()` applies the Sobel filter, parallelizing the row loop with OpenMP.

2.3 GPU Implementation Using CUDA

The goal of implementing the Sobel Filter with CUDA is to exploit GPU parallelism to maximize computational efficiency. Among the three implementations, this CUDA-based approach is expected to yield the highest efficiency.

In this approach, the CUDA kernel `sobel_kernel_gpu()` (shown in Listing 3) applies the Sobel filter to the image in parallel. The Sobel operator is defined as a device function, `sobel_filtered_pixel()`, using the `__device__` qualifier. Each thread is assigned a subset of the pixels, iterating over them with strides based on the total thread count. The Sobel filter kernel weights are stored in constant memory with `__constant__` for faster access.

```
__constant__ float gx[] = {...};
__constant__ float gy[] = {...};

__global__
void sobel_kernel_gpu(
    const float *in,
    float *out,
    int width, int height)
{
  int index = blockIdx.x * blockDim.x + threadIdx.
    x;
  int stride = blockDim.x * gridDim.x;
  int x, y;
  for (int i = index; i < width * height; i +=
    stride)
  {
    x = i % width;
    y = i / width;
    out[i] = sobel_filtered_pixel(in, x, y, width,
     height, gx, gy);
  }
}
```

Listing 3: CUDA Implementation of Sobel Filtering. The `sobel_kernel_gpu()` function applies the Sobel filter with strided access and utilizes constant memory for optimal GPU performance.

2.4 OpenMP Offload to the GPU

The goal of the Sobel Filter with OpenMP GPU offload is to assess the efficiency of using OpenMP for GPU offloading compared to CUDA. OpenMP offloading is simpler than raw CUDA code, though it may be less efficient.

In this implementation, data transfer between host and GPU is declared with `#pragma omp target data`. The variables `in`, `width`, `height`, `gx`, and `gy` are mapped as read-only, while `out` is mapped as write-only. The nested loops over rows and columns are collapsed to optimize memory access.

```
void do_sobel_filtering(float *in, float *out, int
     width, int height) {
  float gx[] = {...}, gy[] = {...};
  int nvals = width * height;
#pragma omp target data \
    map(to : in[0 : nvals], width, height, gx[0 :
    9], gy[0 : 9]) \
    map(from : out[0 : nvals])
  {
#pragma omp target teams distribute parallel for
    collapse(2)
    for (int y = 0; y < height; ++y) {
      for (int x = 0; x < width; ++x) {
        out[x, y] = sobel_filtered_pixel(in, x, y,
     gx, gy);
      }
    }
  }
}
```

Listing 4: OpenMP offload to the GPU implementation of Sobel filtering. `do_sobel_filtering()` applies the Sobel filter, parallelizing the collapsed row/column loop with OpenMP.

3 Results

This section presents the results of experiments testing the performance of different implementations of the Sobel filter on CPU and GPU platforms. We begin with an overview of the hardware and software setup on the Perlmutter supercomputer. The experiments measure runtime, achieved occupancy, and memory bandwidth usage. We examine how well the CPU implementation scales with additional threads, then explore different configurations on the GPU to find the best settings for CUDA. Finally, we compare the performance of OpenMP offloading to GPU against the optimal CUDA configuration, highlighting differences in efficiency.

3.1 Computational platform and Software Environment

3.1.1 System Overview

The experiments were conducted on a GPU node of the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC). Each node is equipped with a single AMD EPYC 7763 (Milan) CPU and four NVIDIA A100 (Ampere) GPUs.

3.1.2 CPU Specifications

The AMD EPYC 7763 processor features 64 cores running at a clock rate of 2.45 GHz. It supports Simultaneous Multi-threading (SMT), enabling two threads per core. Each core has a 32 KiB L1 cache and a 512 KiB L2 cache, while every 8 cores share a 32 MiB L3 cache. The processor also offers 8 memory channels per socket, with 2 DIMMs per channel, and 4 NUMA domains per socket (NPS=4). The system is supported by 256 GiB of DDR4 DRAM, providing a CPU memory bandwidth of 204.8 GiB/s [1, 4].

3.1.3 GPU Specifications

The NVIDIA A100 GPU is composed of multiple GPU Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), and HBM2 memory controllers. Key features include 7 GPCs with 7 or 8 TPCs per GPC, 2 SMs per TPC, and up to 16 SMs per GPC, totaling 108 SMs. Each SM contains 64 FP32 CUDA cores, summing up to 6,912 FP32 CUDA cores per GPU, and 4 third-generation Tensor Cores, totaling 432 Tensor Cores per GPU. The GPU also has 5 HBM2 stacks with 10 512-bit memory controllers, 192 KB of combined shared memory and L1 data cache per SM, a 40 MB Level 2 (L2) cache, and 40 GiB of HBM2 memory (as reported by the `nvidia-smi -q -d MEMORY` command).

Each GPU offers a peak computational throughput of 19.5 TFLOPS for FP32, 9.7 TFLOPS for FP64, 155.9 TFLOPS for TF32 (tensor), 311.9 TFLOPS for FP16 (tensor), and 19.5 TFLOPS for FP64 (tensor). The GPU memory bandwidth is 1,555 GiB/s per 40 GiB HBM2 GPU and 2,039 GiB/s per 80 GiB HBM2e GPU. Each pair of GPUs is connected by four third-generation NVLink connections, offering 25 GiB/s per direction for each link [4, 5].

3.1.4 Software Environment

The system runs on a Linux operating system optimized for high-performance computing. The compilers and tools used in this work are the NVHPC 23.9.0 C++ compiler (`nvc++`) and the NVIDIA 12.2.91 CUDA Compiler (`nvcc`).

3.1.5 Compilation Flags

For the CPU with OpenMP implementation, the code was compiled using the flags `-O3`, `-fast`, and `-fopenmp`. Here, `-O3` enables high-level optimization, `-fast` activates a set of compiler optimizations for performance, and `-fopenmp` enables OpenMP directives for multi-threading on CPUs.

In the GPU with CUDA implementation, the compilation employed the flags `-O3`, `-std=c++14`, and `-forward-unknown-to-host-compiler`. The `-O3` flag enables high-level optimization, `-std=c++14` specifies the C++ standard version, and `-forward-unknown-to-host-compiler` passes unknown flags to the host compiler.

For the GPU with OpenMP Offloading implementation, the key compilation flags included `-O3`, `-fast`, `-mp=gpu`, and `-Minfo=mp,accel`. In this case, `-O3` and `-fast` optimize performance, `-mp=gpu` enables OpenMP offloading to GPUs, and `-Minfo=mp,accel` provides compiler feedback on OpenMP and accelerator optimizations.

3.2 Methodology

To evaluate the Sobel filter implementations, we tested a grayscale image of size 7112×5146. For the CPU OpenMP implementation, we experimented with thread counts of 1, 2, 4, 8, and 16 to observe the impact of varying concurrency levels on performance. For the CUDA GPU implementation, we evaluated our code with different configurations of threads per block $[32, 64, 128, 256, 512, 1024]$ and numbers of thread blocks $[1, 4, 16, 64, 256, 1024, 4096]$. For the OpenMP-offload code, we collected data from a single run.

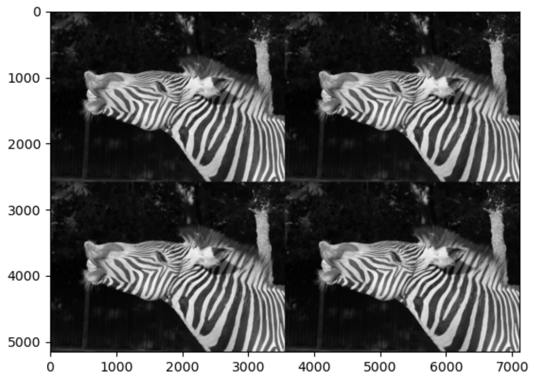
We evaluated the performance of the Sobel filter implementations on both CPU and GPU platforms. To capture the necessary data for this analysis, we collected the following metrics: Runtime, Achieved Occupancy, and Percentage of Peak Sustained Memory Bandwidth.

Figure 1a shows the original grayscale image, while Figure 1b displays the result after applying the Sobel filter.

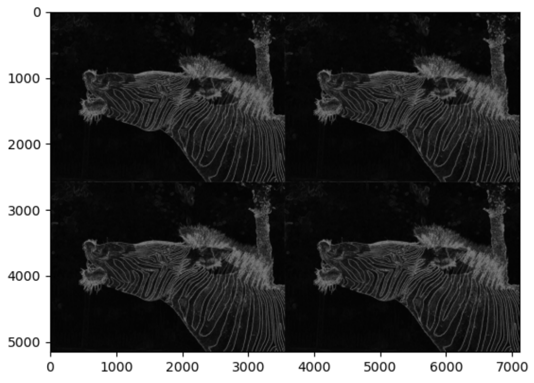
3.2.1 Runtime

- **CPU:** The runtime on the CPU was measured using C++ `chrono_timer()`, which surrounds the line of code calling the `do_sobel_filtering` function.

- **GPU:** For the GPU, runtime was recorded using the NVIDIA Compute Utility (`ncu`), which provides precise measurement of kernel execution time. This allows us to capture the duration of GPU processing without overhead from CPU-GPU data transfers.



(a) Original Grayscale Image



(b) Sobel Filter Output

Figure 1: Example results showing the original grayscale image and the resulting image after applying the Sobel filter.

3.2.2 Achieved Occupancy

The achieved occupancy, a metric specific to GPU performance, was recorded using `ncu`. This value indicates the ratio of active warps to the maximum number of possible warps on a Streaming Multiprocessor during kernel execution, helping us understand how effectively the GPU resources were utilized.

3.2.3 Percentage of Peak Sustained Memory Bandwidth

This GPU-only metric, reported by `ncu`, measures the percentage of peak memory bandwidth utilized by the kernel. This value indicates how efficiently the Sobel filter accesses memory.

3.3 Scaling Study for CPU with OpenMP Parallelism

This experiment aims to analyze the scalability of the Sobel Filter and identify optimal memory access patterns on the CPU. Table 1 presents the runtime and speedup as thread count increases.

The results show that runtime decreases with more threads, highlighting the Sobel Filter's high parallelizability. As Amdahl's Law [2] suggests, the small serial portion of this function enables efficient workload distribution across threads. This parallel nature also makes it well-suited for GPU acceleration, where additional performance gains are achievable.

We also tested an alternative parallel approach by applying *collapse(2)* to the OpenMP pragma, collapsing the nested loop structure.

Threads (T)	Runtime (s)	Speedup
1	0.185	1.0
2	0.093	1.992
4	0.046	3.979
8	0.024	7.746
16	0.013	14.558

Table 1: Runtime performance of CPU with OpenMP parallel implementation of the Sobel Filter. As the number of threads increases, runtime decreases substantially, illustrating the parallelizable nature of the operation.

The results, shown in Table 2, indicate that this approach leads to degraded performance and lower speedup. This is due to the fact that CPU threads require sequential memory access patterns to optimize cache utilization and minimize cache misses.

Threads (T)	Runtime (s)	Speedup
1	0.204	1.0
2	0.103	1.981
4	0.055	3.708
8	0.032	6.433
16	0.021	9.910

Table 2: Runtime performance of CPU with collapsed OpenMP parallel implementation of the Sobel Filter. When collapsing the nested loop, performance degrades due to less efficient memory access patterns.

3.4 Scaling Study for GPU using CUDA

The objective of this experiment is to analyze performance variations across different configurations of thread block counts and threads per block to identify the optimal configurations for achieving better GPU performance.

As shown in Figure 2, runtime performance is significantly poorer with smaller thread block counts or lower threads-per-block configurations. In contrast, configurations using larger block sizes and higher thread counts per block yield notably improved runtimes. This pattern suggests that runtime efficiency is highly correlated with the total number of threads.

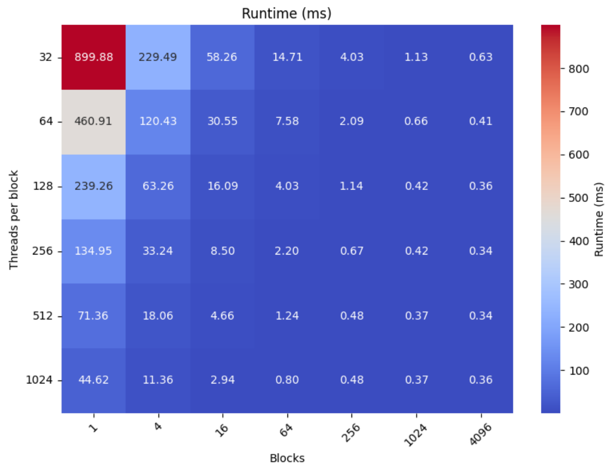


Figure 2: Heatmap for Runtime (ms)

Figure 3 shows that Achieved Occupancy improves significantly with larger thread block sizes and a higher number of threads per block. With more threads per block, more warps are available, enabling greater concurrent warp scheduling. Consequently, occupancy increases as threads per block increase. However, increasing the number of thread blocks does not enhance occupancy until the count reaches 256, as the GPU contains 108 SMs (see Section 3.1). Below this threshold, each SM handles only one thread block, limiting occupancy gains. Beyond this point, additional thread blocks can be allocated per SM, further increasing occupancy.

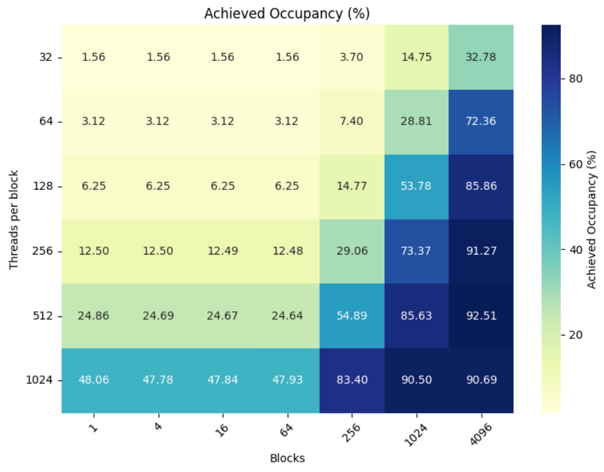


Figure 3: Heatmap for Achieved Occupancy (%)

Figure 4 reveals that the Percentage of Peak Sustained Memory Bandwidth is higher with larger threads per block and increased thread block counts, indicating a positive correlation with the total thread count. However, maximum memory bandwidth utilization does not occur at the highest configuration of (T, B) = (1024, 4096). Instead, configurations such as (T, B) = (512, 4096) and others with slightly lower counts (e.g., (512, 1024) or (256, 4096)) perform better. This discrepancy is likely due to the image size used in the experiment (7112×5146), where each thread is assigned 8 or 9 pixels when (T, B) = (1024, 4096). In this setup, threads assigned 8 pixels complete their tasks faster and must wait for threads assigned 9 pixels, leading to idle time. This overhead becomes significant when fewer pixels are assigned per thread, reducing the efficiency gains from the higher thread count as idle time increases.

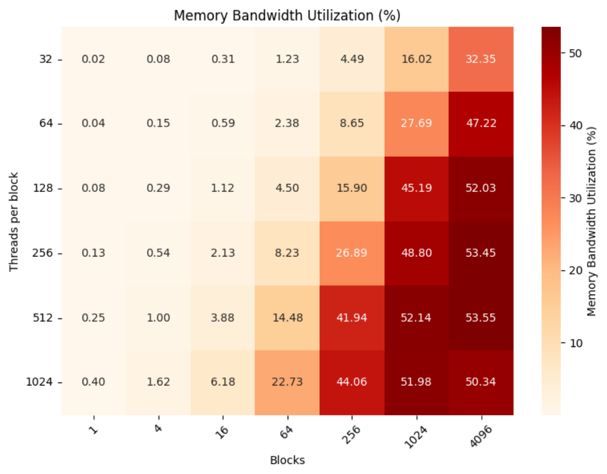


Figure 4: Heatmap for Percentage of Peak Sustained Memory Bandwidth (%)

3.5 Study for OpenMP offload to the GPU

The objective of this experiment is to evaluate the efficiency of OpenMP offloading compared to optimized CUDA configurations.

According to the `ncu` output, the OpenMP offload setup generates 285,925 thread blocks with 128 threads per block. The thread blocks count corresponds to the total number of collapsed iterations 7112×5146 divided by 128.

Table 3 presents a performance comparison between OpenMP offload on the GPU and the optimal CUDA configuration, which uses 512 threads per block and 4096 thread blocks. The optimized CUDA code outperforms the OpenMP offload by approximately 2x across all metrics. This performance difference is due to OpenMP's approach, which assigns each iteration to a thread, resulting in an excessive number of total threads. As discussed in Section 3.4, creating too many threads can introduce significant overhead, sometimes outweighing the benefits of parallelization. Nonetheless, the capability of OpenMP to deliver competitive performance with straightforward pragmas and plain C++ code demonstrates its potential for portable and accessible parallelization.

Configuration	Runtime	Occupancy	Mem Bandwidth
CUDA	0.34 (ms)	92.51 (%)	53.55 (%)
OpenMP	0.62 (ms)	55.61 (%)	29.27 (%)

Table 3: Performance comparison of OpenMP offload to the GPU with the optimal CUDA configuration (T=512, B=4096).

Acknowledgments

This paper was edited with the assistance of *ChatGPT (GPT-4o and GPT-4o with canvas)* (accessed November 2024), which was primarily used for correcting grammatical errors and rephrasing. This research was supported by resources from the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility, under NERSC award DDR-ERCAP m3930 for 2024.

References

[1] AMD. HPC Tuning Guide for AMD EPYC 7003 Series Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/high-performance-computing-tuning-guide-amd-epyc7003-series-processors.pdf, 2022. Accessed: November 2024.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), p. 483–485. Association for Computing Machinery, New York, NY, USA, 1967. doi: 10.1145/1465482.1465560

[3] N. Kanopoulos, N. Vasanthavada, and R. L. Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.

[4] NERSC. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/, 2024. Accessed: November 2024.

[5] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020. Accessed: November 2024.