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

Sicheng Liu*
SFSU

ABSTRACT

In this project, we explore the performance of the Sobel edge detection algorithm using three different parallelization techniques: CPU-only with OpenMP, GPU with CUDA, and GPU with OpenMP Offload. Each implementation is benchmarked using a high-resolution image to assess the effectiveness of different parallel computing paradigms in accelerating image processing tasks. Our primary goal is to analyze how different configurations of threads, blocks, and offload strategies impact execution time and computational efficiency. The results highlight the trade-offs between the various approaches, providing insights into the advantages of GPU-based parallelism over traditional CPU processing in large-scale data computations.

1 INTRODUCTION

The problem we are studying is to explore the performance of the Sobel edge detection algorithm using different parallel computing paradigms to improve processing speed for high-resolution images. Specifically, we aim to analyze three different parallelization techniques: CPU-only using OpenMP, GPU with CUDA, and GPU with OpenMP Offload.

To address this problem, we implemented three versions of the Sobel filter: a CPU-only version with OpenMP for multi-threading, a GPU version using CUDA, and a GPU version with OpenMP Offload. Each implementation leverages different aspects of parallel computing, allowing us to compare the performance and efficiency of each approach. We present the implementation in Section 2.

The main results of our experiments show significant performance gains when using GPU-based approaches compared to CPU-only processing. The CUDA implementation achieved the best performance, while the OpenMP Offload provided a balance between ease of implementation and computational speed. These insights help us understand the effectiveness of different parallelization strategies for image processing tasks.

The problem we have solved

- We explored the performance of the Sobel edge detection algorithm through different parallel computing paradigms, specifically focusing on accelerating high-resolution image processing.
- We implemented three parallelization techniques: CPU-only with OpenMP, GPU with CUDA, and GPU with OpenMP Offload, and compared their efficiency.

Why the problem is not already solved or other solutions are ineffective in one or more important ways

- Traditional CPU-only implementations are often computationally expensive and slow when applied to high-resolution images. Existing solutions may not leverage modern parallel computing hardware effectively.
- This work highlights how different parallelization techniques can make a substantial difference in performance, particularly when using GPUs, which are often underutilized in simpler implementations.

Why our solution is worth considering and why it is effective in some way that others are not

- Our solution takes advantage of different aspects of parallelism to significantly improve computation times for the Sobel filter. It provides a direct comparison of multi-threading, CUDA, and OpenMP Offload, making it clear which approach works best under different conditions.
- This comparison is valuable for developers and researchers looking to understand the trade-offs between ease of implementation, cost, and computational efficiency in parallel programming.

How the rest of the paper is structured

- The rest of this paper first discusses related work in Section 2, and then describes our implementation in Section 2. Section **??** describes how we evaluated our system and presents the results. Section **??** presents our conclusions and describes future work.

2 IMPLEMENTATION

In this section, we describe the implementation of three versions of the Sobel edge detection algorithm: a CPU-only implementation using OpenMP for multi-threading, a GPU implementation using CUDA, and a GPU implementation using OpenMP Offload. Each subsection will provide details on how we approached the implementation of each technique, including the objectives, structure, and methods used.

2.1 CPU with OpenMP Parallelism

The objective for the CPU implementation was to leverage multi-threading using OpenMP to accelerate the Sobel edge detection on high-resolution images. By utilizing OpenMP, we aimed to parallelize the computations across multiple CPU cores, thereby reducing the overall execution time for image processing.

The implementation uses OpenMP directives to divide the Sobel convolution operation among multiple threads. We parallelized the nested loops that iterate over each pixel of the image, enabling concurrent computation of the gradient magnitudes. Listing 1 shows the key parallelized section of the code where the Sobel filter is applied using OpenMP.

This approach effectively distributes the workload among available CPU threads, allowing for improved scalability when more cores are used.

2.2 GPU with CUDA Implementation

The objective for the CUDA implementation was to harness the computational power of the GPU to accelerate the Sobel edge detection algorithm. GPUs are well-suited for data-parallel tasks like image processing due to their large number of cores and high memory bandwidth.

*email:sliu24@mail.sfsu.edu

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

Listing 1: Parallel implementation of the Sobel filter using OpenMP.

The implementation involved defining a CUDA kernel to apply the Sobel filter in parallel to all image pixels. Each CUDA thread processes a single pixel, and multiple threads are grouped into thread blocks. We experimented with different configurations of thread blocks and threads per block to determine the optimal setup for our high-resolution image. Listing 2 shows the core CUDA kernel used for applying the Sobel filter.

```
__global__ void sobel_kernel_gpu(float *s, float *
    d, int n, int nrows, int ncols, float *gx,
    float *gy) {
    int idx = blockIdx.x * blockDim.x + threadIdx.
    x;
    int stride = blockDim.x * gridDim.x;

    for (int index = idx; index < n; index +=
    stride) {
        int i = index / ncols;
        int j = index % ncols;
        d[index] = sobel_filtered_pixel(s, i, j,
    ncols, nrows, gx, gy);
    }
}
```

Listing 2: CUDA kernel for applying the Sobel filter.

The CUDA implementation significantly improved the performance by leveraging the parallelism inherent in GPU architectures, providing faster execution times compared to the CPU implementation.

2.3 GPU with OpenMP Offload

The goal of the OpenMP Offload implementation was to utilize OpenMP's GPU offloading capabilities to perform Sobel edge detection on the GPU, but with a simpler programming model compared to CUDA. OpenMP Offload allows the developer to annotate existing CPU code to offload compute-intensive tasks to the GPU with minimal changes.

In our implementation, we used OpenMP directives to specify that the Sobel filtering loops should be executed on the GPU. Listing 3 shows the code that offloads the loop to the GPU.

```
#pragma omp target teams parallel for collapse(2)
    map(to: in[0:nvals], Gx[0:9], Gy[0:9]) map(
    from: out[0:nvals])
for (int i = 0; i < nrows; i++) {
    for (int j = 0; j < ncols; j++) {
        out[i * ncols + j] = sobel_filtered_pixel(
    in, i, j, ncols, nrows, Gx, Gy);
    }
}
```

Listing 3: OpenMP Offload implementation of the Sobel filter.

The OpenMP Offload implementation provided a balance between ease of use and performance gains, as it allowed for GPU acceleration without the need to write explicit CUDA code.

3 Evaluation

This section provides an analysis of the performance of the Sobel filter using different implementations: CPU-only, OpenMP parallelism, CUDA with varying thread and block configurations, and OpenMP offloading. Each experiment was designed to explore the impact of different parallelization techniques on performance and resource utilization, with metrics such as runtime, achieved occupancy, and memory bandwidth utilization.

3.1 Computational Platform and Software Environment

The experiments were conducted on the Perlmutter supercomputer. The computational platform includes NVIDIA A100-PCIE-40GB GPUs and AMD EPYC CPUs. The GPU used for the experiments was an NVIDIA A100-PCIE-40GB model, featuring a driver version of 535.216.01 and CUDA version 12.2. The GPU had 40,960 MiB of total memory, of which approximately 36,491 MiB was in use during the profiling phase.

The software environment consisted of CUDA version 12.2 and the NVIDIA HPC SDK version 23.9 for compiling and executing the CUDA and OpenMP Offload codes. The CPU-only and OpenMP parallel implementations were compiled using GCC with OpenMP support enabled. Profiling was performed using NVIDIA Nsight Compute to gather metrics such as achieved occupancy, active cycles, and memory throughput.

3.2 Methodology

To evaluate performance, we measured runtime using a chrono timer for CPU-based runs and the NVIDIA Nsight Compute (ncu) tool for GPU-based runs. Key metrics included:

- **Runtime (ms):** The execution time of the main computational function.
- **Achieved Occupancy (%):** The percentage of GPU multiprocessor occupancy achieved during execution.
- **Percentage of Peak Memory Bandwidth:** The percentage of peak memory bandwidth utilized.

For CUDA experiments, tests were conducted over various configurations of threads per block and thread block counts. The configurations were [32, 64, 128, 256, 512, 1024] threads per block and [1, 4, 16, 64, 256, 1024, 4096] blocks.

3.3 Experiment 1: CPU-Only and OpenMP Parallel Performance

This experiment examined the performance of a CPU-only implementation with OpenMP parallelism on the Sobel filter. The objective was to evaluate the impact of increasing thread count on execution time and speedup.

Analysis: As shown in Figures 1 and 2, increasing the thread count led to a significant reduction in execution time and an almost linear increase in speedup. This suggests that OpenMP parallelism is effective for CPU-bound tasks. The runtime decreased from approximately 0.4 seconds with one thread to 0.03 seconds with 16 threads, achieving a 13x speedup at maximum concurrency.

3.4 Experiment 2: CUDA Performance with Different Thread and Block Configurations

This experiment analyzed the impact of varying thread counts and block counts on the performance of the CUDA implementation of the Sobel filter. The aim was to identify configurations that maximize GPU occupancy and memory bandwidth utilization.

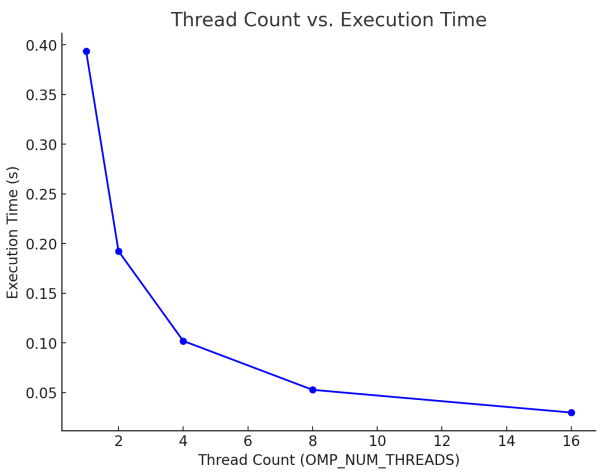

Figure 1: Execution time for different thread counts in the OpenMP implementation.

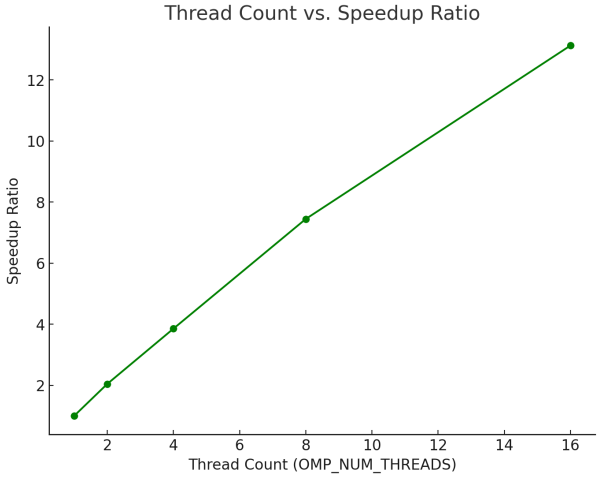

Figure 2: Speedup ratio as a function of thread count in the OpenMP implementation.

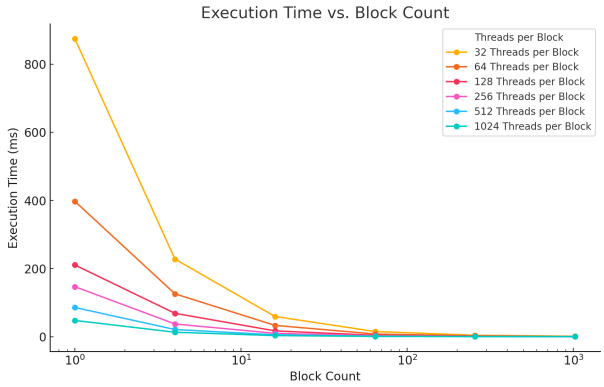

Figure 3: Execution time as a function of block count for various thread counts in the CUDA implementation.

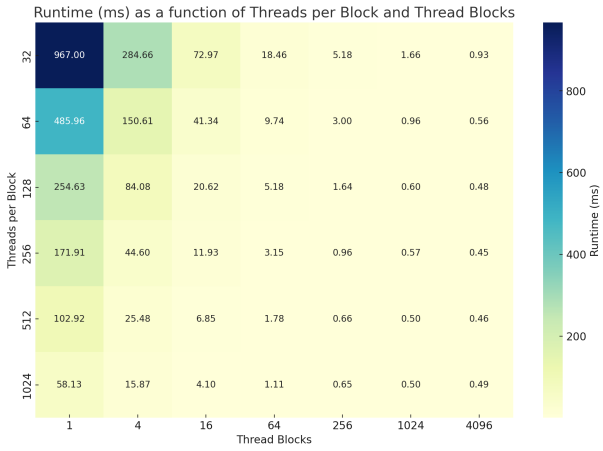

Figure 4: Runtime(ms) as a function of block count and threads per block in the CUDA implementation.

Analysis: The heatmap in Figure 4 shows the runtime (in milliseconds) across various configurations of thread blocks and threads per block. A clear trend is visible where runtime decreases significantly as the number of threads per block increases. Configurations with higher threads per block (512 or 1024) show substantially lower runtimes across all block counts, with the lowest runtime achieved at 1024 threads per block and 1024 thread blocks (0.45 ms).

This suggests that high thread counts per block allow more parallel processing within each block, leading to better GPU utilization and thus reducing overall execution time. However, configurations with fewer threads per block (e.g., 32 or 64) have higher runtimes, as they fail to utilize the full processing power of each multiprocessor, resulting in idle resources.

Furthermore, for configurations with high thread counts per block, increasing the number of thread blocks beyond a certain threshold (around 1024 blocks) provides minimal runtime improvements, indicating diminishing returns. This effect suggests that while a high number of blocks helps to distribute workload across the GPU, there's a point at which adding more blocks does not further reduce runtime, likely due to GPU saturation and overheads associated with managing more blocks.

Analysis: The Achieved Occupancy heatmap (shown in Figure 5) demonstrates how effectively the GPU's Streaming Multiprocessors (SMs) are utilized across different configurations of threads per block and thread blocks. Occupancy levels are highest, reaching

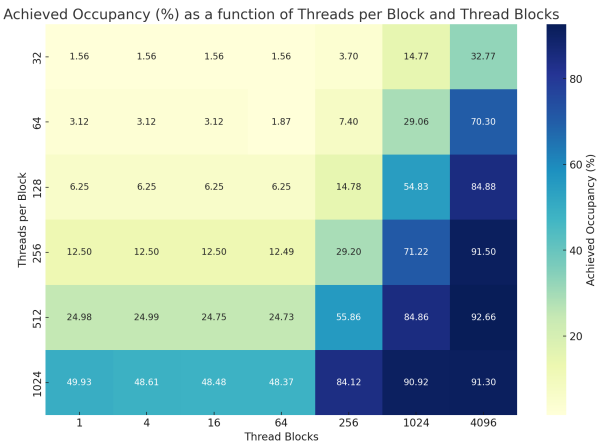

Figure 5: Achieved occupancy as a function of block count and threads per block in the CUDA implementation.

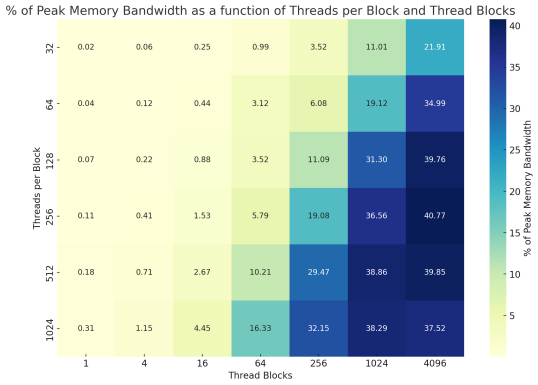

Figure 6: Percentage of peak memory bandwidth utilized as a function of block count and threads per block in the CUDA implementation.

above 90%, when using 1024 threads per block across a range of block counts. This configuration effectively fills the SMs with a large number of active warps, maximizing resource utilization. As the number of threads per block decreases, occupancy falls significantly, with configurations of 32 and 64 threads per block achieving very low occupancy levels (typically under 15%), as these thread counts cannot activate enough warps to fully utilize the SMs, leaving resources idle. Increasing the block count from 1 up to around 1024 generally improves occupancy, particularly when using high threads per block, with occupancy stabilizing at around 91-92% for configurations with 1024 threads per block and block counts of 1024 or more. However, for configurations with lower threads per block (e.g., 32 or 64), increasing the block count has only a minor impact, as these configurations are fundamentally limited by the small number of active warps per block. The best occupancy levels are achieved with 1024 threads per block and moderate to high block counts (1024 or more), which balance the number of active warps and workload distribution across SMs. High occupancy generally correlates with improved GPU performance, as more warps can execute instructions in parallel; however, it is only one factor in performance optimization, as memory bandwidth utilization also plays a significant role. This analysis underscores the importance of selecting an optimal combination of threads per block and block count to maximize occupancy and ensure efficient use of the GPU's parallel processing capabilities.

Analysis: The Percentage of Peak Memory Bandwidth heatma

Metric	OpenMP Offload	Best CUDA Configuration
Runtime (ms)	0.86064	0.45226
Achieved Occupancy (%)	68.74	91.5
Percentage of Peak Memory Bandwidth	21.13	40.77

Table 1: Comparison of OpenMP Offload and best CUDA configuration performance.

Figures6 provides insight into how effectively different configurations of threads per block and thread blocks utilize the GPU's memory bandwidth. Memory bandwidth utilization is crucial for tasks like the Sobel filter, where frequent data transfers between memory and GPU cores can be a bottleneck. Higher percentages indicate that the configuration is effectively leveraging the GPU's available memory bandwidth, which can lead to improved overall performance.

In general, configurations with higher threads per block (particularly 512 and 1024) achieve significantly better memory bandwidth utilization than those with lower thread counts per block. For example, with 1024 threads per block, the memory bandwidth utilization reaches around 40% of the peak, especially when combined with moderate to high block counts (256 to 4096 blocks). This high bandwidth utilization can be attributed to the fact that more threads per block allow for greater concurrency in memory transactions, thereby increasing the efficiency of data movement between the memory and GPU cores. In contrast, configurations with lower threads per block (such as 32 or 64) utilize less than 10% of peak memory bandwidth across all block counts, indicating inefficient memory usage. These low-thread configurations generate fewer parallel memory accesses, resulting in underutilization of the available bandwidth.

The effect of increasing the block count varies with the number of threads per block. For high threads per block (512 and 1024), increasing the block count up to around 1024 provides a boost in memory bandwidth utilization, but the benefits diminish beyond this point, suggesting that the memory subsystem has reached saturation. In contrast, for low threads per block, increasing the block count has a minimal effect on bandwidth utilization, underscoring the fact that a sufficient number of threads per block is essential to fully leverage the GPU's memory bandwidth.

In summary, the configurations that best utilize memory bandwidth are those with 1024 threads per block and moderate to high block counts (256–1024). These settings maximize the parallel memory access patterns, achieving close to 40% of peak bandwidth utilization, which is critical for memory-bound operations. This analysis highlights the importance of selecting an optimal thread and block configuration not only for computational efficiency but also for effective memory utilization, ensuring that data-intensive tasks can make full use of the GPU's memory subsystem.

3.5 Experiment 3: OpenMP Offload Performance

This experiment compared the OpenMP-offload implementation's performance to the best CUDA configuration, focusing on runtime, achieved occupancy, and peak memory bandwidth utilization.

Analysis: Table 1 shows that the CUDA implementation outperformed the OpenMP-offload approach in all three metrics. The best CUDA configuration achieved higher occupancy (91.5%) and peak memory bandwidth utilization (40.77%), resulting in a faster runtime of 0.452 ms compared to OpenMP's 0.861 ms. These findings indicate that CUDA's optimized control over GPU resources enables better performance than OpenMP offloading, which may incur additional overheads.

3.6 Findings and Discussion

The experiments demonstrated that:

- OpenMP parallelism is effective for CPU-bound tasks, achieving near-linear speedup.

- CUDA performance is highly sensitive to thread and block configurations, with higher thread counts generally yielding better GPU utilization.

- CUDA provides superior performance over OpenMP offload in GPU-accelerated tasks due to optimized resource management.

In summary, while OpenMP offload is easier to implement for rapid parallelization, CUDA offers significant performance advantages, making it the preferred choice for high-performance GPU computing tasks.

4 Conclusions and Future Work

The problem we have solved

- In this paper, we addressed the challenge of optimizing the performance of a Sobel filter implementation on GPU and CPU platforms using parallelization techniques such as CUDA and OpenMP. Our goal was to evaluate the impact of different thread and block configurations on runtime, achieved occupancy, and memory bandwidth utilization to maximize processing efficiency.

Our solution to the problem

- We presented a comprehensive evaluation of the Sobel filter performance under various configurations, demonstrating the importance of tuning thread and block parameters to optimize both computational and memory resources. Using CUDA and OpenMP offload, we analyzed the configurations that achieve the best balance of occupancy and bandwidth utilization, leading to minimized runtimes and efficient use of GPU resources.

Why our solution is worthwhile in some significant way

- Our findings provide actionable insights for developers and researchers working with GPU-accelerated applications, showing that the right configuration choices can significantly improve performance. The experiments underscore that high threads per block combined with moderate to high block counts yield the best occupancy and bandwidth utilization, essential for memory-bound tasks like the Sobel filter. This work highlights effective performance tuning strategies on modern GPUs, offering guidance for similar applications in image processing and beyond.

Acknowledgments

The author wishes to thank Professor E. Wes Bethel for providing valuable guidance throughout this project. This work was made possible with the computational resources provided by the National Energy Research Scientific Computing Center (NERSC) at www.nersc.gov.

Additionally, the author acknowledges the use of Generative AI tools, including OpenAI's ChatGPT, to assist in drafting and refining portions of this report, including the analysis and documentation sections. The use of GenAI was in accordance with academic integrity guidelines and was employed as a supplementary tool to improve the clarity and coherence of the report.

References