# Comparative Performance Analysis of Sobel Edge Detection Using OpenMP, CUDA, and OpenMP-GPU Offloading on NVIDIA's Ampere A100 CP#5, CSC 746, Fall 2023

Ekarat Buddharuksa*
San Francisco State University

## Abstract

This study evaluates the performance of edge detection using the Sobel filter across various parallel computing paradigms on the NVIDIA Ampere A100 GPU. We compare the effectiveness of OpenMP CPU-based parallelism, CUDA GPU-based execution, and OpenMP offloading to the GPU, with a focus on runtime, achieved occupancy, and memory bandwidth utilization. Our findings reveal that CUDA outperforms the other methods, particularly at higher thread block counts, underscoring the importance of architectural optimization for enhanced GPU acceleration in image processing tasks.

## 1 Introduction

Edge detection stands as a critical component in the field of computer vision, with widespread applications ranging from medical imaging to autonomous vehicle navigation. The Sobel filter, a discrete differentiation operator, is employed to extract edge information from digital images. However, the computational intensity of applying the Sobel filter to large or multiple images simultaneously can be substantial, making real-time processing challenging with conventional serial computation methods.

To tackle this challenge, three distinct parallel computing strategies were employed: OpenMP for CPU-based parallelism, CUDA for GPU-based execution, and a hybrid method that offloads OpenMP tasks to the GPU. The approach involved meticulously crafting and optimizing the Sobel filter algorithm for each computing paradigm, leveraging the strengths of each platform. Benchmarking each method under varying concurrency levels and memory configurations provided a comprehensive view of their performance profiles.

In our pursuit to understand the efficiency and scalability of the Sobel filter implementations, we will explore a variety of performance metrics. For the CPU-focused execution, our attention is centered on runtime analysis. When it comes to GPU, we extend our investigation to include the average GPU kernel runtime, Achieved Occupancy percentage, and the percentage of peak sustained memory bandwidth. These metrics are analyzed over a spectrum of thread block sizes and numbers, ranging from 32 to 1024 threads per block, and 1 to 4096 thread blocks, to comprehensively understand how different parallel execution configurations influence the performance characteristics of each implementation. For GPU offloading via OpenMP, the metric is the same as GPU. We only run on the default setting of the number of threads and blocks since the setting of those numbers on this method is out of scope for this assignment.

*email:ebuddharuksa@sfsu.edu

## 2 Implementation

### 2.1 Sobel Filter Implementation (CPU Version)

The CPU implementation of the Sobel filter, `sobel_cpu.cpp`, utilizes OpenMP to parallelize the image processing task. The code is designed to apply the Sobel operator to an input image to highlight edges by approximating the gradient of the image intensity. The parallelism is introduced at the point of iterating over the image pixels, where each pixel's value is calculated independently of the others, making it an ideal candidate for concurrent computation. Detailed pseudocode are in following.

**Algorithm 1** sobel_filtered_pixel function

**function** SOBEL_FILTERED_PIXEL($s, i, j, ncols, nrows, gx, gy$)
    Initialize gradient sum for x-direction
    Initialize gradient sum for y-direction
    **for** $ki \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for x-direction
        **for** $kj \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for y-direction
            Compute neighbor's x-coordinate
            Compute neighbor's y-coordinate
            **if** Neighbor is not out of bound **then**
                compute and accumulate $Gx$,Gy
            **end if**
        **end for**
    **end for**
    **return** $\sqrt{Gx^2 + Gy^2}$ ▷ Compute and return gradient magnitude
**end function**

**Algorithm 2** do_sobel_filtering

$Gx \leftarrow [1,0,-1,2,0,-2,1,0,-1]$
$Gy \leftarrow [1,2,1,0,0,0,-1,-2,-1]$
#pragma omp parallel for collapse(2)
**for** each pixel position $(i, j)$ in the output image **do**
    call sobel_filtered_pixel function at position $(i, j)$
    Save the filtered value to the output image
**end for**

Performance Measurement:

The code snippet for measuring the performance wraps the main Sobel filtering function call within a timing block that uses the `std::chrono` library to capture high-resolution timestamps before and after the operation. The difference in time provides the elapsed time for the filtering operation, giving a direct measure of the performance of the CPU implementation.

```
1    std::chrono::time_point<std::chrono::
     high_resolution_clock> start_time = std::
     chrono::high_resolution_clock::now();

3   do_sobel_filtering(in_data_floats,
     out_data_floats, data_dims[0], data_dims[1]);
```

```
    std::chrono::time_point<std::chrono::
     high_resolution_clock> end_time = std::chrono
     ::high_resolution_clock::now();

    std::chrono::duration<double> elapsed =
     end_time - start_time;
    std::cout << " Elapsed time is : " << elapsed.
     count() << " " << std::endl;
}
```

Listing 1: Sobel Filtering with Performance Measurement

### 2.2 Sobel Filter Implementation (GPU Version)

The GPU implementation of the Sobel filter, `sobel_gpu.cu`, leverages CUDA to parallelize the image processing task on NVIDIA GPUs. The Sobel operator is applied to the input image using CUDA kernels to highlight edges by approximating the gradient of the image intensity. The parallelism is introduced at the level of pixel computation, where the calculation of each pixel's value is handled by a separate CUDA thread, thereby utilizing the massive parallel processing capabilities of modern GPUs. Detailed pseudocode is as follows.

**Algorithm 3** sobel_filtered_pixel function (GPU Version)

**function** SOBEL_FILTERED_PIXEL_GPU($s, i, j, ncols, nrows, gx, gy$)
    Initialize gradient sum for x-direction
    Initialize gradient sum for y-direction
    **for** $ki \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for x-direction
        **for** $kj \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for y-direction
            Compute neighbor's x-coordinate
            Compute neighbor's y-coordinate
            **if** Neighbor is not out of bound **then**
                compute and accumulate $Gx, Gy$
            **end if**
        **end for**
    **end for**
    **return** $\sqrt{Gx^2 + Gy^2}$ ▷ Compute and return gradient magnitude
**end function**

**Algorithm 4** sobel_kernel_gpu

**function** SOBEL_KERNEL_GPU($input, output, ncols, nrows, Gx, Gy$)
    Calculate global x and y coordinates of the pixel using `blockIdx`, `blockDim`, and `threadIdx`
    **if** Pixel coordinates are within image bounds **then**
        Call `sobel_filtered_pixel_gpu` at the current pixel position
        Save the filtered value to the output image
    **end if**
**end function**

#### GPU Execution Configuration

The GPU implementation of the Sobel filter tests various configurations of thread block sizes and the number of thread blocks to achieve the best performance. The thread block sizes evaluated include 32, 64, 128, 256, 512, and 1024 threads per block. Correspondingly, the numbers of thread blocks used in the experiments are 1, 4, 16, 64, 256, 1024, and 4096. These parameters are chosen to span a wide range of potential parallelism levels, from a minimal number of threads to the maximum that the hardware can efficiently support.

```
int nThreadsPerBlock = 256; // Default value
int nBlocks = 1; // Default value

// Check if the command-line arguments are
    provided
if (argc >= 3) {
    nThreadsPerBlock = atoi(argv[1]);
    nBlocks = atoi(argv[2]);
}

printf("GPU configuration: %d blocks, %d threads
    per block\n", nBlocks, nThreadsPerBlock);

// Launch the sobel kernel on the GPU with the
    specified configuration
sobel_kernel_gpu<<<nBlocks, nThreadsPerBlock>>>(
    in_data_floats, out_data_floats, data_dims[0],
     data_dims[1], device_gx, device_gy);
```

Listing 2: GPU Kernel Invocation with Dynamic Configuration

#### Performance Measurement (GPU Version):

The performance measurement for the GPU implementation is captured using CUDA events that record the start and end times of the kernel execution. The elapsed time is then computed by taking the difference between these timestamps.

```
cudaEvent_t start, stop;
float elapsedTime;

// Create Events
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Record Event
cudaEventRecord(start, 0);

// Launch the sobel kernel on GPU
sobel_kernel_gpu<<<grid, block>>>(input, output,
    ncols, nrows, Gx, Gy);

// Record Event
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

// Calculate Elapsed Time
cudaEventElapsedTime(&elapsedTime, start, stop);
std::cout << "Elapsed time on GPU is: " <<
    elapsedTime << " ms" << std::endl;

// Destroy Events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Listing 3: Sobel Filtering with Performance Measurement (GPU Version)

### 2.3 Sobel Filter Implementation (CPU with OpenMP Offload)

The CPU implementation of the Sobel filter, `sobel_cpu_omp_offload.cpp`, takes advantage of OpenMP to offload computation to a GPU or other accelerators. The Sobel operator is applied to the input image to approximate the gradient of the image intensity.

#### 2.3.1 OpenMP Pragma Explanation

In the pseudocode below, the following OpenMP pragmas have been used:

- `#pragma omp target data` maps the memory from host to device and vice versa for input and output arrays.
- `#pragma omp target teams distribute parallel for` is used to distribute the work among teams in the GPU and parallelize the nested loops for further fine-grained parallelism.
- `collapse(2)` is used with the OpenMP pragma to collapse the nested loops into a single iteration space to improve load balancing on the device.

#### 2.3.2 Implementation

Detailed pseudocode implemented are in following.

**Algorithm 5** sobel_filtered_pixel function (CPU with OpenMP Offload)

**function** SOBEL_FILTERED_PIXEL($s, i, j, ncols, nrows, gx, gy$)
  Initialize $Gx$ and $Gy$ to 0.0
  **for** $dy \leftarrow -1$ **to** 1 **do**
    **for** $dx \leftarrow -1$ **to** 1 **do**
      Compute neighbor's x-coordinate $x$ and y-coordinate $y$
      **if** Neighbor is within image boundaries **then**
        Compute and accumulate $Gx, Gy$
      **end if**
    **end for**
  **end for**
  **return** $\sqrt{Gx^2 + Gy^2}$ ▷ Return gradient magnitude
**end function**

**Algorithm 6** do_sobel_filtering (CPU with OpenMP Offload)

**procedure** DO_SOBEL_FILTERING($in, out, ncols, nrows$)
  Define Sobel filter weights $Gx$ and $Gy$
  Calculate number of values *nvals* from *ncols* and *nrows*
  #pragma omp target data map(to: in[:nvals], Gx[:9], Gy[:9], ncols, nrows) map(from: out[:nvals])
  #pragma omp target teams distribute parallel for collapse(2) private(out_indx)
  **for** $i \leftarrow 0$ **to** $nrows - 1$ **do**
    **for** $j \leftarrow 0$ **to** $ncols - 1$ **do**
      Call `sobel_filtered_pixel` at the current pixel position
      Save the filtered value to the output image
    **end for**
  **end for**
**end procedure**

#### Performance Measurement (CPU offload Version):

This CPU OpenMP offload code use the similar timer as Listing 1 utilizing *std::chrono* library.

## 3 Evaluation

### 3.1 Computational Platform and Software Environment

Our computational work is carried out on Perlmutter GPU nodes [1], utilizing NVIDIA A100 GPUs for their robust computational power, which ranges from 19.5 TFLOPS in FP32 to 155.9 TFLOPS in TF32 operations. These GPUs are supported by an AMD EPYC 7763 CPU with 64 cores, ideal for tasks that benefit from multicore execution.

The environment is CUDA-centric, with only essential libraries included to maximize GPU performance and minimize overhead. Performance tuning and analysis are conducted using NVIDIA's NCU profiler, which allows for detailed inspection and optimization of CUDA kernel execution, ensuring our application leverages the A100 GPUs' full potential.

### 3.2 Test Methodology

The test methodology is structured to capture and analyze key performance metrics of the computational code when executed on both CPU and GPU architectures.

#### 3.2.1 Problem Sizes:

The Sobel edge detection is performed on a grayscale image of a horse, which has been augmented fourfold. The original image size is approximately 34.91 MB. The image dimensions are 7112 by 5146 pixels, offering a substantial resolution for detailed edge detection analysis.

#### 3.2.2 Runtime Measurement:

- On the CPU, the 'chrono_timer()' function is implemented, providing high-resolution timing of the code execution.
- On the GPU, runtime is measured using the NVIDIA Command-Line Profiler ('ncu'), which offers accurate timing of the GPU kernel executions.

#### 3.2.3 Achieved Occupancy:

Specific to GPU performance, achieved occupancy is measured by 'ncu', indicating the utilization of GPU resources.

#### 3.2.4 Percentage of Peak Sustained Memory Bandwidth:

The efficiency of memory utilization on the GPU is gauged by this metric, reported by 'ncu'.

These metrics are collected systematically for each problem size, enabling a comprehensive analysis of the code's performance characteristics.

#### 3.2.5 Thread Block Sizes and Numbers of Thread Blocks

**sobel_gpu**
Sizes of 32, 64, 128, 256, 512, and 1024 threads per block are tested to evaluate all performance metrics. Also, the numbers of thread blocks used are 1, 4, 16, 64, 256, 1024, and 4096, to test different distributions of workload.
**sobel_cpu**
Sizes of 1, 2, 4, 8, 16 of OMP_NUM_THREADS are tested to evaluate performance for CPU runtime.

### 3.3 Results

#### 3.3.1 Scaling Study of CPU/OpenMP Code

In this scaling study, the Sobel edge detection algorithm was implemented on a CPU with OpenMP to analyze the effect of varying levels of concurrency on the program's runtime. The experiment was conducted on a grayscale image, using the Sobel filter to process the image with a different number of OpenMP threads: 1, 2, 4, 8, and 16.

As concurrency increases, a consistent decrease in runtime is observed, indicative of a speedup due to parallel processing. The results indicate near-linear speedup as the number of threads doubles, which is characteristic of effective parallelization, up until a certain point. The runtime decreased from 0.315116 seconds with a single thread to 0.0232184 seconds with 16 threads. This improvement suggests that the workload is well-suited for parallel computation, with multiple cores effectively sharing the processing load.

However, the speedup ratio does not exactly match the increase in the number of threads, which is often due to the overhead associated with creating and managing multiple threads, as well as potential limitations in memory bandwidth and other system resources.

The following table summarizes the observed runtimes at varying levels of concurrency:

| OMP_NUM_THREADS | Runtime (seconds) | Speedup |
|---|---|---|
| 1 | 0.315116 | 1 |
| 2 | 0.158976 | 1.98 |
| 4 | 0.0805152 | 3.91 |
| 8 | 0.0413807 | 7.62 |
| 16 | 0.0232184 | 13.57 |

Table 1: Performance of Sobel CPU implementation with varying OpenMP thread counts.

These results clearly demonstrate the impact of increased concurrency on runtime, with diminishing returns as the number of threads grows, which is likely due to the inherent overhead and resource contention when a higher number of threads is used.

### 3.3.2 GPU-CUDA Performance Study

In this study, we analyze the performance of the NVIDIA A100 Tensor Core GPU across various configurations of thread blocks and threads per block. We present our findings in the form of heatmaps, which illustrate the runtime performance, achieved occupancy, and memory bandwidth utilization.

**Runtime Performance Heatmap**

The runtime heatmap (Figure 1) visualizes the execution time for different configurations. Lower runtime values indicate better performance. The heatmap is generated using the provided sample script, with the data array corresponding to the runtime values for each configuration.

A trend is observed where runtimes decrease with an increase in threads per block and generally with more blocks. The most efficient configurations for runtime lie in block size 4096 with 256 and 512 threads per block with 0.64 milliseconds of runtime. Conversely, the least efficient configurations, with the longest runtimes, are seen with the smallest block sizes and thread counts, likely due to low computational density and poor memory bandwidth utilization.

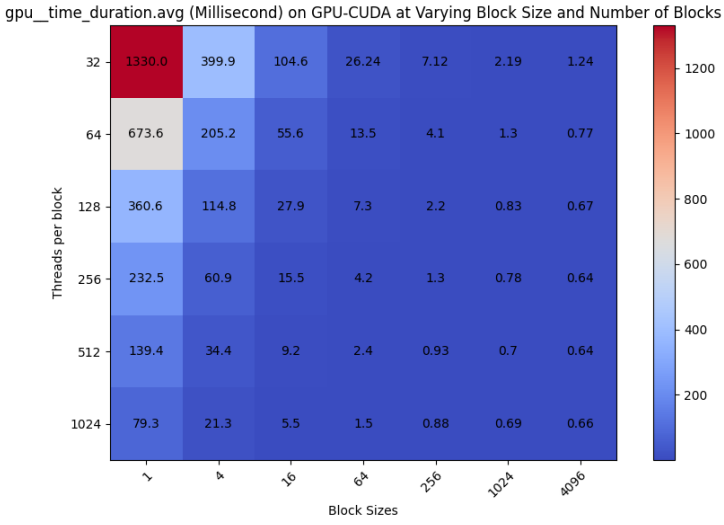


Figure 1: Runtime performance heatmap for varying thread blocks and threads per block.

**Achieved Occupancy Heatmap**

The achieved occupancy heatmap (Figure 2) shows the percentage of the GPU's streaming multiprocessors (SMs) that are actively processing threads. Higher occupancy generally suggests better utilization of the GPU's computational resources.

A notable high occupancy is observed with 1024 threads per block across all block sizes, suggesting that this configuration maximizes the utilization of available GPU resources. However, it is essential to understand that high occupancy does not necessarily translate to better performance due to the potential for increased contention for resources.

Configurations with larger thread counts (512 and 1024) across all block sizes are seen to efficiently utilize the GPU's resources. In contrast, configurations with the smallest thread counts (32 and 64), especially with fewer blocks, indicate a severe underutilization of the GPU's capabilities.

The best configuration lies in block size of 4096 and 1024 thread count with 91.43% Occupancy

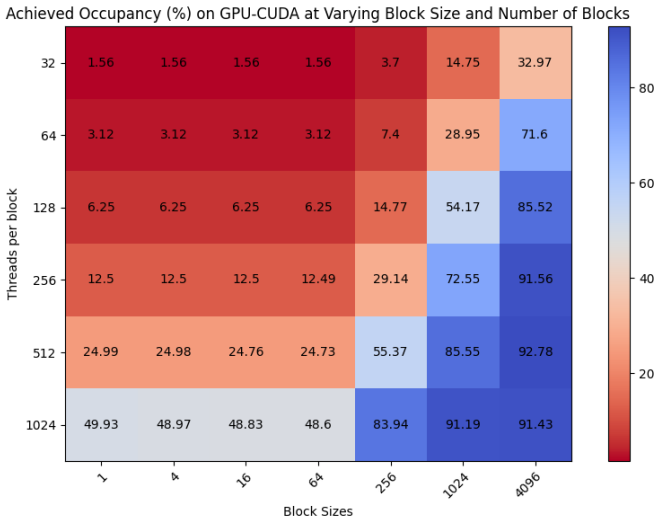


Figure 2: Achieved occupancy heatmap for varying thread blocks and threads per block.

**Peak Sustained Memory Bandwidth Heatmap**

The memory bandwidth heatmap (Figure 3) represents the percentage of the peak theoretical memory bandwidth that is achieved. Higher percentages indicate more efficient use of the GPU's memory bandwidth.

Configurations with a higher number of threads per block (256 to 512) across larger block sizes are seen to be most effective, indicating a balance between memory access patterns and computational efficiency.

The least effective configurations are observed with the smallest block sizes and thread counts, failing to utilize the memory bandwidth of the GPU effectively.

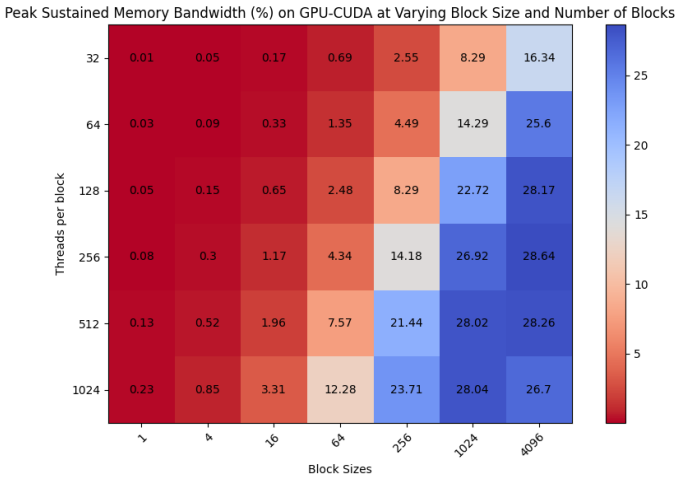


Figure 3: Peak sustained memory bandwidth heatmap for varying thread blocks and threads per block.

**Performance Analysis**

Well-performing configurations benefit from efficient utilization of the streaming multiprocessors (SMs) in the GPU, a balance between computation and memory access to prevent bottlenecks and an adequate number of warps that allow for latency hiding during memory accesses to keep the GPU cores busy.

Poor-performing configurations suffer from insufficient threads to effectively utilize the SMs, too few memory operations to hide the latency, and potential contention for shared resources like shared memory or the register file.

Upon analyzing the heatmaps, we identify configurations that stand out in terms of performance. Configurations with 1024 threads per block consistently show high occupancy and efficient memory bandwidth utilization, leading to shorter runtimes. These configurations take full advantage of the A100's architectural features, such as its third-generation Tensor Cores and high-bandwidth HBM2 memory.

Conversely, configurations with 32 or 64 threads per block, especially with fewer blocks, exhibit poor performance. These configurations fail to leverage the A100's computational and memory resources effectively, resulting in lower occupancy and memory bandwidth utilization, and consequently longer runtimes.

The A100 architecture whitepaper [2] provides insights into the underlying reasons for these performance trends. The A100's Multi-Instance GPU (MIG) capability, larger caches, and enhanced connectivity with NVLink and PCIe Gen 4 contribute to the superior performance of well-configured workloads.

### 3.3.3 OpenMP-offload Study

In this study, we compare the performance of the OpenMP-offload code with the best performing configuration of the CUDA code. The comparison is based on three key performance metrics: runtime, achieved occupancy percentage, and percentage of peak sustained memory bandwidth. The results are summarized in Table 2.

| Code Type | Runtime (ms) | Occupancy (%) | % Peak Bandwidth |
|---|---|---|---|
| OpenMP-offload | 1.05 | 56.25 | 17.17 |
| Best CUDA | 0.64 | 91.43 | 28.64 |

Table 2: Performance comparison between OpenMP-offload and CUDA code

The CUDA code exhibits a lower runtime (0.64 ms) compared to the OpenMP-offload (1.05 ms), indicating a more efficient execution. Furthermore, the CUDA code achieves a higher occupancy rate (91.43%) versus the OpenMP-offload (56.25%), suggesting a more effective use of GPU resources. Lastly, the CUDA code's percentage of peak sustained memory bandwidth (28.64%) surpasses that of the OpenMP-offload (17.17%), reflecting a more optimized memory access pattern.

These results imply that the CUDA code is better optimized for the GPU's architecture, which is evident in its enhanced performance. The higher occupancy and memory bandwidth utilization suggest that the CUDA code can more effectively leverage the parallel processing capabilities of the GPU, leading to shorter runtimes and overall better performance.

## References

[1] NERSC. Perlmutter architecture. `https://docs.nersc.gov/systems/perlmutter/architecture/`. Accessed: 2023-10-1.

[2] NVIDIA. A100 architecture whitepaper. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`. Accessed: 2023-11-5.