Comprehensive Optimization of CUDA Kernels: Achieving High GPU Performance Through Warp, Memory, and Stream Techniques in Canny Edge Detection

1st Jifeng Li
Department of Electronic Engineering
Columbia University
New York City, NY, US
jl6962@columbia.edu

2nd Rishita Yadav
Department of Electrical Engineering
Columbia University
New York City, NY, US
ry2501@columbia.edu

***Abstract*—This project explores the implementation and optimization of Canny Edge Detection using parallel computing techniques on CUDA. We systematically enhanced the baseline implementation by incorporating shared memory optimization, memory coalescing, and warp-level optimization strategies. Each optimization step was evaluated for its performance impact, leveraging NVIDIA Nsight Compute to analyze and compare efficiency gains. The results demonstrated significant speedup at every stage, underscoring the effectiveness of these optimization methods in accelerating the Canny edge detection algorithm. This study highlights the potential of advanced parallel computing techniques in improving the performance of computationally intensive image processing tasks.**

I. Introduction

Edge detection is a fundamental task in computer vision, serving as the foundation for object recognition, image segmentation, and motion analysis. Among various edge detection algorithms, the Canny edge detector [1] remains one of the most widely used due to its effectiveness in reducing noise and accurately detecting edges. However, the computational cost of Canny edge detection becomes a significant bottleneck when processing large-scale datasets such as ImageNet [2], especially in real-time applications or on high-resolution images.

To address this computational challenge, this work explores the acceleration of the Canny edge detection pipeline using NVIDIA CUDA, a parallel computing platform and programming model for GPUs [3]. The CUDA-based implementation of Canny edge detection leverages GPU-optimized techniques such as shared memory, efficient memory access patterns, and CUDA streams to exploit the massive parallelism inherent in modern GPUs. We compare the performance of our CUDA-based approach with an OpenCV-based implementation [4] to highlight the advantages of GPU acceleration.

The experimental evaluation is conducted on a subset of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2017 dataset [2]. To ensure a comprehensive analysis, the images were pre-processed and resized to six different resolutions: 128×128, 256×256, 320×320, 512×512, 640×640, and 1024×1024. The performance of both methods is evaluated in terms of execution time and accuracy.

Our results demonstrate that the CUDA-based implementation achieves significant speedups compared to the OpenCV implementation, particularly at higher resolutions where GPU parallelism is more effectively utilized. This study underscores the potential of GPU-accelerated edge detection for real-time applications and large-scale image processing tasks.

The contributions of this paper are as follows:

- We propose a GPU-accelerated implementation of the Canny edge detection pipeline using CUDA.
- We compare the CUDA implementation with the OpenCV-based approach on various image resolutions.
- We evaluate the performance of both methods using the ILSVRC 2017 dataset [2] to showcase scalability and computational efficiency.

The rest of this paper is organized as follows: Section II reviews related work, Section III describes the proposed GPU-accelerated Canny edge detection pipeline, Section IV presents the experimental setup and results, and Section V concludes the paper with future directions.

II. Background and Related work

Numerous researchers have explored the parallel implementation of the Canny edge detection algorithm, employing diverse hardware and optimization techniques. SHI Weizhong et al. [5] developed an FPGA-based optimization algorithm, enhancing processing speed for 512 × 512 gray-scale images. Jin et al. [6] utilized the ZC706 platform and SDSoC environment to accelerate edge detection, achieving a 16.97× speedup for the same resolution. Similarly, Keqiang et al. [7] optimized the Canny operator on the TI DSP TMS320C6678 processor, improving performance at an 800 × 600 resolution. Xiangjiao et al. [8] implemented a parallel Canny algorithm using TBB and C++, achieving a 3.673× acceleration on 22.89M gray-scale images using a quad-core CPU. Yue et al. [9] executed

the algorithm on GPUs with OpenGL, ensuring real-time performance for 256 × 256 images. Bin et al. [10] introduced a GPU+CPU-based approach, realizing a 5.39× speedup for 1024 × 1024 images.

Further, researchers like Jin et al. [11] proposed a Canny algorithm under the OpenCL architecture, reporting a 1.42× speedup for 2048 × 1536 images without factoring in data transfer. Mochurad [12] leveraged CUDA to achieve a 68× performance boost for 10240 × 10240 gray-scale images, while Horvath et al. [13] reported a 101× speedup for 1280 × 720 images on CUDA. Others explored batch processing improvements via Hadoop clusters [19, 20]. Enhanced algorithms for specific applications have also been proposed; for instance, FPGA implementations optimized for real-time 512 × 512 edge detection [21, 22] and FPGA-based methods for mobile vision systems. Suwen et al. [14] enhanced weak edge detection capabilities on FPGA, while Shengxiao et al. [15] achieved a 64× acceleration for 512 × 512 images on GPUs.

Advanced applications include Fuqiang et al.'s [16] low-error line segment detection using FPGA-based Canny algorithms, Sivakumar and Janakiraman's [17] MRI ROI segmentation on FPGA, and Hongye's [18] fingerprint acquisition system optimized with DSP. Other examples include systems integrating DSP and FPGA for verticality recognition, CUDA-based Gaussian mixture models for real-time video analysis, and satellite-based ship tracking using multi-feature Canny algorithms on embedded GPUs.

In summary, three main research avenues are evident: parallelizing the Canny operator on various architectures (CPU, FPGA, DSP, GPU, and Hadoop clusters), enhancing the operator's computational efficiency, and applying it to practical systems for acceleration. While FPGA and CUDA exhibit high acceleration potential, they face challenges like cost, debugging complexity, and limited portability. Most existing studies focus on a single parallel technology, lacking comparative analysis across different models. Considering the heterogeneity of modern computing platforms, efficiently utilizing diverse processors, such as CPUs and GPUs, is critical.

This paper addresses these gaps by exploiting GPU storage through PyCUDA and PyOpenCL for high-speed Canny edge detection. The study analyzes and optimizes GPU memory access modes (global, local, and constant) for parallelized Gaussian filtering and gradient computation. Enhancements include optimizing template operations and extending images to align with GPU architectures, reducing storage demands, and improving vectorized computation. Experimental results validate the effectiveness of these optimizations in achieving efficient parallel execution and high-speed edge detection.

III. Methodology

The implementation of the Canny Edge Detection algorithm was carried out through four different approaches, namely:

- Naive Implementation
- Shared Memory Optimization
- Memory Coalescing
- Warp-Level Optimization

Each implementation followed a structured pipeline consisting of preprocessing, edge detection, and optimization steps. The key phases of the methodology are described below:

A. Preprocessing Steps

Before applying the edge detection algorithm, the input image underwent the following preprocessing steps to prepare it for gradient calculations:

1) Grayscale Conversion: The input color image was first converted to a grayscale image to reduce computational complexity. This was achieved using a weighted sum of the RGB channels, as per the standard formula:

$$I_{gray} = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

where are the red, green, and blue pixel intensities, respectively, as can be seen from figure 2.

2) Gaussian Blurring: To reduce noise and smooth the image, a Gaussian blur was applied using a convolution filter. A 2D Gaussian kernel was used, defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where is the standard deviation of the Gaussian distribution. The kernel size and were chosen to balance between noise reduction and edge preservation.

B. Edge Detection Steps

The Canny Edge Detection algorithm was implemented through the following stages:

1) Gradient Calculation (Sobel Filtering): The Sobel operator was used to calculate the gradients of the smoothed image in the x and y directions. The Sobel filters are defined as:

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

The gradients and were convolved with the image to compute the gradient magnitude and direction :

$$G = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

2) Non-Maximum Suppression: To thin the edges and retain only the strongest ones, non-maximum suppression was applied. For each pixel, the gradient magnitude was compared to its neighbors along the direction of the gradient. If the pixel value was not the local maximum, it was suppressed (set to zero).

3) Thresholding (Hysteresis): To finalize the edges, a double-thresholding technique was applied:

- High threshold: Retain strong edge pixels.
- Low threshold: Include weak edge pixels if they are connected to strong edges. This step ensured the detection of meaningful edges while reducing noise-induced artifacts.

Fig. 1. Comparision between outputs of CUDA and OpenCL outputs

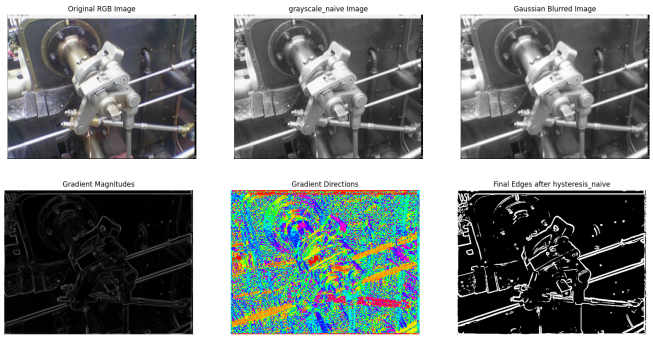

Fig. 2. Outputs from different steps of processing

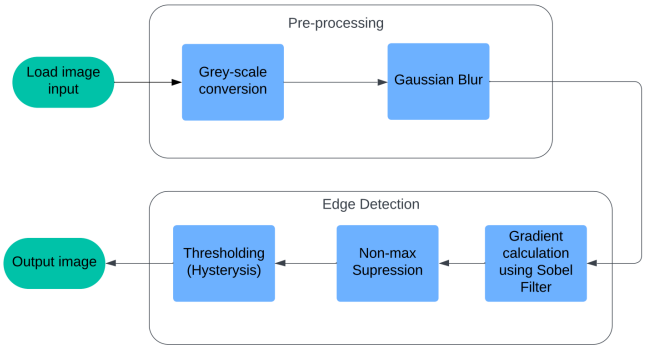

Fig. 3. Block diagram of the methodolgy pipeline

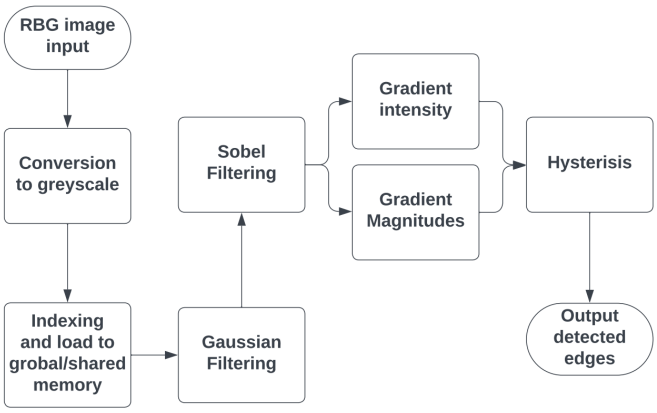

Fig. 4. Flowchart of the implementation steps

C. CUDA Implementation Optimization Strategies

The four implementations were designed to progressively optimize the performance of the Canny Edge Detection algorithm on GPU hardware. The optimizations are summarized as follows:

1) Naive Implementation: In the naive implementation, each pixel computation was handled independently without any optimization. The image data was read directly from global memory, and computations were performed sequentially on the GPU.

2) Tiling and Warp Optimizations: To further optimize the algorithm, warp-level operations were incorporated to reduce thread divergence and improve parallel execution efficiency. The following strategies were employed:

- Warp intrinsics: Built-in functions were used to share data within a warp, minimizing shared memory usage for certain calculations.
- Gradient comparison: Non-maximum suppression was optimized using warp-level synchronization to process neighboring pixels efficiently.

3) Shared/Private Memory Optimizations: To reduce global memory access latency, shared memory was utilized to load image tiles into faster on-chip memory. Threads within a block cooperatively loaded a portion of the image into shared memory, allowing for faster access during convolution and gradient calculations.

- Tile-based processing: Image data was divided into small tiles that fit into shared memory.
- Boundary handling: Extra rows and columns were loaded to handle boundary pixels during filtering.

D. Kernel Configurations

The kernel configurations for each implementation were carefully selected to maximize GPU utilization. A block size of threads was used for most implementations, as it strikes a balance between parallelism and shared memory usage. The grid size was dynamically calculated based on the input image dimensions to ensure full coverage of the image. For warp-level optimization, the block size was aligned to warp boundaries (multiples of 32) to take advantage of warp-level intrinsics and minimize thread divergence. These configurations were experimentally validated to achieve optimal performance across all implementations.

E. OpenCL implementation

The same methodology was implemented in OpenCL as well. The output of both methods were compared. The edge detection results of both can be seen in figure 1.

F. Flowchart and Pseudocode

The implemented algorithm focuses on edge detection through a multi-stage pipeline consisting of *Pre-processing* and *Edge Detection*, as illustrated in Figures 3 and 4. The input RGB image is first converted to greyscale to reduce computational complexity while retaining critical edge information. Gaussian filtering is then applied to smooth the image and minimize noise, ensuring more robust edge detection.

In the *Edge Detection* phase, the Sobel filter calculates gradient magnitudes and intensities to identify significant intensity changes corresponding to edges. Non-maximum suppression is applied to eliminate non-local maxima, refining the gradient response. Finally, double-thresholding with hysteresis links weak edges to strong edges, ensuring continuity while discarding false positives. The resulting output is the detected edges in the input image.

The implementation efficiently manages data flow by loading greyscale image data into global/shared memory, followed by sequential application of Gaussian filtering, Sobel filtering, and edge refinement techniques. This structured pipeline optimizes memory access and computational load while achieving accurate edge detection.

Algorithm 1 CUDA-Based Canny Edge Detection (Overview)

Require: Input image I with dimensions $\texttt{width} \times \texttt{height}$, Gaussian kernel K, thresholds $T_{\text{high}}, T_{\text{low}}$
Ensure: Edge-detected image M'

$G \leftarrow$ **Algorithm 2** (Grayscale Conversion)
$B \leftarrow$ **Algorithm 3** (Gaussian Blur on G)
$M, \theta \leftarrow$ **Algorithm 4** (Gradient Magnitude and Direction)
$M \leftarrow$ **Algorithm 5** (Non-Maximum Suppression on M, θ)
$M' \leftarrow$ **Algorithm 6** (Double Thresholding on M)
$M' \leftarrow$ **Algorithm 7** (Hysteresis Thresholding on M')
return M'

Algorithm 2 Grayscale Conversion

Require: Input image I of size $\texttt{width} \times \texttt{height}$
Ensure: Grayscale image G

for all pixels (x, y) in the image grid **do**
    $G(x, y) \leftarrow 0.299 \cdot R(x, y) + 0.587 \cdot G(x, y) + 0.114 \cdot B(x, y)$
end for
Synchronize device
return G

Algorithm 3 Gaussian Blur

Require: Grayscale image G, Gaussian kernel K
Ensure: Blurred image B

for all pixels (x, y) where $2 \leq x < \texttt{width} - 2$ and $2 \leq y < \texttt{height} - 2$ **do**
    $B(x, y) \leftarrow \sum_{i=-2}^{2} \sum_{j=-2}^{2} G(x+i, y+j) \cdot K(i+2, j+2)$
end for
Synchronize device
return B

Algorithm 4 Gradient Magnitude and Direction

Require: Blurred image B
Ensure: Gradient magnitude M and direction θ

for all pixels (x, y) where $1 \leq x < \texttt{width} - 1$ and $1 \leq y < \texttt{height} - 1$ **do**
    $G_x(x, y) \leftarrow \sum_{u=-1}^{1} \sum_{v=-1}^{1} B(x+u, y+v) \cdot S_x(u, v)$
    $G_y(x, y) \leftarrow \sum_{u=-1}^{1} \sum_{v=-1}^{1} B(x+u, y+v) \cdot S_y(u, v)$
    $M(x, y) \leftarrow \sqrt{G_x(x, y)^2 + G_y(x, y)^2}$
    $\theta(x, y) \leftarrow \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right)$
end for
Synchronize device
return M, θ

Algorithm 5 Non-Maximum Suppression

Require: Gradient magnitude M, gradient direction θ
Ensure: Thinned gradient magnitude M

for all pixels (x, y) where $1 \leq x < \texttt{width} - 1$ and $1 \leq y < \texttt{height} - 1$ **do**
    Compare $M(x, y)$ with neighbors in the direction $\theta(x, y)$
    if $M(x, y)$ is not a local maximum **then**
        $M(x, y) \leftarrow 0$
    end if
end for
Synchronize device
return M

Algorithm 6 Double Thresholding

Require: Gradient magnitude M, thresholds $T_{\text{high}}, T_{\text{low}}$
Ensure: Edge map M'

for all pixels (x, y) in the image grid **do**
    if $M(x, y) \geq T_{\text{high}}$ **then**
        $M'(x, y) \leftarrow 255$ ▷ Strong edge
    else if $T_{\text{low}} \leq M(x, y) < T_{\text{high}}$ **then**
        $M'(x, y) \leftarrow 128$ ▷ Weak edge
    else
        $M'(x, y) \leftarrow 0$ ▷ No edge
    end if
end for
return M'

Algorithm 7 Hysteresis Thresholding

Require: Thresholded image M'
Ensure: Refined edge map M'
for all pixels (x, y) where $M'(x, y) = 128$ (weak edge) **do**
  if any neighbor $M'(x + \Delta x, y + \Delta y) = 255$ (strong edge) **then**
    $M'(x, y) \leftarrow 255$ ▷ Promote to strong edge
  else
    $M'(x, y) \leftarrow 0$ ▷ Suppress weak edge
  end if
end for
Synchronize device
return M'

IV. Experimental Setup

A. Dataset

In this work, a subset of the ILSVRC2017 dataset, consisting of 80 images, was utilized for experimental purposes. These images were preprocessed to generate a structured dataset with multiple image resolutions, enabling evaluation across varying scales. The preprocessing pipeline involved systematic cropping and resizing of images to six predefined resolutions. The following steps outline the procedure:

1) **Input and Output Setup:** The original images, located in the input directory (`./images/`), were processed and stored in a designated output directory (`./cropped_datasets/`).
2) **Resolutions:** Each image was processed to produce six different resolutions:
   - 128×128
   - 256×256
   - 320×320
   - 512×512
   - 640×640
   - 1024×1024
3) **Cropping and Resizing:** For each resolution:
   - If the target size was smaller than or equal to the original image size, the image was center-cropped.
   - If the target size exceeded the dimensions of the original image, the image was resized using the Lanczos resampling filter.

   The cropping procedure was performed by calculating the center of each image. Specifically, given an image of dimensions $W \times H$ (width and height) and a target size $w \times h$, the cropping bounds were calculated as follows:

$$\text{left} = \frac{W - w}{2}, \quad \text{top} = \frac{H - h}{2}, \tag{1}$$

$$\text{right} = \text{left} + w, \quad \text{bottom} = \text{top} + h. \tag{2}$$

   These bounds ensured that the central region of each image was preserved.
4) **Directory Structure:** The processed images were organized into subdirectories based on their resolution. For each resolution $w \times h$, a separate folder (`w_xh/`) was created under the output directory. For example:
   - `./cropped_datasets/128x128/`
   - `./cropped_datasets/256x256/`, and so on.
5) **File Format:** The original file names were preserved in the output directories, and the images were saved in their original format (`.jpg`, `.JPEG`, or `.png`).

The resulting dataset consists of 80 images per resolution, yielding a total of 480 processed images organized into six folders. This structured dataset allows for systematic evaluation and analysis across multiple scales, facilitating experiments that require input images of varying dimensions.

B. The computational platform

The computational platform is a virtual machine running on Google Cloud, equipped with an NVIDIA Tesla T4 GPU with 15.36 GB of memory and CUDA version 12.6. The system utilizes an Intel Xeon(R) CPU E7-2300 at 2.30 GHz with 2 cores and 6 threads. The platform operates on an x86_64 architecture and supports 64-bit processing. Memory usage for the running Python process is 98 MiB. The system is optimized for heterogeneous computing tasks, leveraging both CPU and GPU capabilities for high-performance parallel computations. The environment includes NVIDIA's Night Compute Command Line Profiler (v2024.3.1.0).

V. Results and Discussion

A. CUDA vs OpenCV Execution Time Comparison

This study compares the performance of different CUDA implementations (**Naive**, **Warp Optimized**, **Memory Lock**, **Shared Memory**) with OpenCV across six image resolutions: 128×128, 256×256, 320×320, 512×512, 640×640, and 1024×1024. The results are summarized in Figure 5.

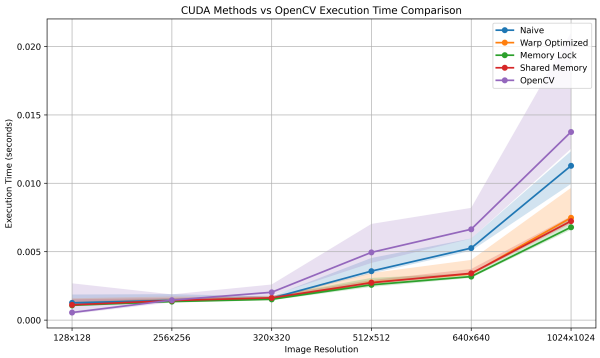

Fig. 5. Execution time comparison of CUDA methods vs OpenCV for various image resolutions.

At lower resolutions (128×128 and 256×256), all methods, including CUDA and OpenCV, show similar execution times due to the minimal computational load. However, as the resolution increases beyond 512×512, performance differences become evident. The OpenCV implementation exhibits a steep rise in execution time, particularly at 1024×1024, indicating a

lack of GPU optimizations. In contrast, CUDA-based methods scale better with increasing image size.

Among the CUDA implementations, the **Naive method** shows moderate improvement but remains less efficient compared to the optimized approaches. The **Warp Optimized** and **Memory Lock** methods achieve the best performance, demonstrating the benefits of warp-level parallelism and memory access optimizations. The **Shared Memory** implementation also improves performance by reducing global memory access latency through data reuse, though it is slightly less efficient than the Warp Optimized and Memory Lock methods.

At the largest resolution (1024×1024), the execution time for OpenCV is significantly higher than all CUDA methods, further underscoring the advantages of GPU-based parallelism and memory optimizations employed in CUDA implementations.

B. Shared Memory Version Analysis

The shared memory version of the *gaussianBlur* kernel demonstrates improved performance compared to the baseline (*gaussianBlur_naive*) by utilizing tiled memory access and reducing redundant global memory reads. Table I summarizes the key results from the profiling data for GPU activities and API calls.

The profiling results reveal that the `gaussianBlur` kernel accounts for **30.32%** of the total GPU execution time, with an average runtime of **31.11** μ**s** per call. However, Host-to-Device (HtoD) and Device-to-Host (DtoH) memory copies together contribute to over 60% of GPU activity time, indicating significant overhead. This suggests opportunities for optimization, such as using pinned memory or implementing asynchronous memory copies [3].

The analysis of API calls shows that the most time-consuming operations are `cuCtxCreate` (64.21%) and `cuCtxDetach` (25.50%), which are associated with context initialization and cleanup. These calls are one-time costs and do not directly impact kernel execution performance. In contrast, the kernel launch overhead from `cuLaunchKernel` is minimal, contributing only **0.36%** of the total API call time.

The shared memory optimization in the *gaussianBlur_shared* kernel significantly reduces global memory access latency by tiling data into shared memory. This optimization results in improved kernel execution time compared to the baseline. However, the substantial proportion of memory transfer time remains a bottleneck, highlighting the need for further optimizations, such as asynchronous memory transfers or double-buffering, to overlap computation with data movement and minimize latency [19].

The *gaussianBlur_shared* kernel effectively improves performance by leveraging shared memory to reduce global memory latency. Nevertheless, memory transfers still dominate GPU activity time, limiting the overall performance gains. Addressing these data transfer inefficiencies through advanced memory management techniques will be essential to achieving further improvements.

C. NCU Report Analysis

In this section, we analyze the performance of various CUDA kernels with the *gaussianBlur_naive* implementation serving as the baseline. The focus is on throughput metrics, Roofline analysis, and key performance indicators to compare the optimized versions: *gaussianBlur_warp*, *gaussianBlur_shared*, and *gaussianBlur_memlock*.

The baseline kernel, *gaussianBlur_naive*, demonstrates significant runtime improvement potential but suffers from inefficient memory access patterns and limited compute resource utilization. As shown in the Roofline chart (Figure 6), its low arithmetic intensity reveals that the kernel is memory-bound. GPU throughput analysis indicates that the kernel achieves **68.69%** of theoretical compute and memory throughput, with moderate utilization of Streaming Multiprocessors (SMs) and registers. The main bottlenecks stem from uncoalesced global memory accesses and excessive DRAM latency, presenting opportunities for further optimization.

The optimized kernels address these limitations with varying degrees of success. The *gaussianBlur_warp* kernel employs warp-level parallelism to minimize thread divergence and coalesce memory accesses. As a result, it achieves compute and memory throughput of **79.29%**, which demonstrates a considerable improvement over the baseline. The Roofline analysis confirms that this version has increased arithmetic intensity and better utilization of memory bandwidth.

The *gaussianBlur_shared* kernel further optimizes performance by leveraging shared memory tiling to reduce global memory access latency. The GPU throughput analysis reveals that this implementation achieves the highest throughput at **83.33%**, effectively balancing memory and compute resource utilization. Shared memory improves data locality within thread blocks, minimizing DRAM accesses and reducing memory bottlenecks.

The *gaussianBlur_memlock* kernel focuses on memory locking to minimize uncoalesced memory accesses. This version achieves **83.17%** compute and memory throughput, slightly lower than the shared memory version but still demonstrating balanced and efficient GPU resource utilization. Notably, L2 cache dependency is reduced to **4.41%**, and DRAM throughput improves to **6.62%**, highlighting the effectiveness of memory locking for mitigating global memory inefficiencies.

Overall, the optimized kernels (*gaussianBlur_warp*, *gaussianBlur_shared*, and *gaussianBlur_memlock*) significantly outperform the baseline. The Roofline analysis and throughput metrics indicate that these versions address both memory and compute-bound limitations. The shared memory optimization achieves the highest throughput, while the memory locking technique closely follows, highlighting the benefits of reducing latency and improving memory coalescence.

a) Summary Analysis: The overall performance summary, as depicted in Figure 11, reveals substantial improvements across the optimized kernels when compared to the baseline implementations. The naive versions, such as *gaussianBlur_naive*, exhibit limited efficiency due to uncoalesced

TABLE I
PERFORMANCE ANALYSIS OF SHARED MEMORY VERSION

Type	Time (%)	Total Time (ms)	Calls	Avg (us)	Min – Max (us)
GPU Activities					
[CUDA memcpy HtoD]	45.17%	3.7072	81	45.768	0.672 – 63.424
gaussianBlur	30.32%	2.4888	80	31.110	17.247 – 44.831
[CUDA memcpy DtoH]	15.17%	1.2449	80	15.561	8.5130 – 20.864
rgb2gray	9.34%	0.7669	80	9.586	6.048 – 13.216
API Calls					
cuCtxCreate	64.21%	297.07	1	297070	297070 – 297070
cuCtxDetach	25.50%	117.96	1	117960	117960 – 117960
cuMemFree	3.02%	13.972	240	58.215	4.697 – 197.06
cuMemAlloc	2.24%	10.341	240	43.087	2.148 – 190.07
cuMemcpyHtoD	2.05%	9.4755	81	116.98	50.542 – 185.36
cuMemcpyDtoH	1.07%	4.9293	80	61.616	41.948 – 87.190
cuCtxSynchronize	0.81%	3.7375	80	46.718	25.984 – 64.373
cuStreamSynchronize	0.58%	2.6814	80	33.517	20.409 – 47.098
cuLaunchKernel	0.36%	1.6621	160	10.388	4.462 – 58.547

memory access patterns, low arithmetic intensity, and excessive memory latency. These issues are reflected in their lower compute and memory throughput, as well as higher execution times.

The optimized kernels address these limitations through targeted memory and computational optimizations. The *gaussianBlur_warp* kernel achieves a throughput of **79.29%** by leveraging warp-level parallelism to reduce divergence and coalesce global memory accesses. The introduction of shared memory tiling in *gaussianBlur_shared* further improves performance, enabling data reuse and reducing global memory latency. As a result, *gaussianBlur_shared* achieves the highest compute and memory throughput of **83.33%**, demonstrating the effectiveness of shared memory for reducing bandwidth dependency. The *gaussianBlur_memlock* kernel also shows significant gains, reaching **83.17%** throughput, primarily due to its ability to minimize uncoalesced memory accesses and improve data transfer efficiency.

A broader analysis of all kernels in Figure 11 highlights three key insights. First, optimized kernels consistently outperform the naive implementations, with up to a **56.19x** estimated speedup in compute performance. Second, memory optimization techniques such as shared memory tiling and memory locking play a central role in mitigating global memory bottlenecks and improving GPU resource utilization. Third, the NCU summary identifies uncoalesced memory accesses and long scoreboard stalls as critical optimization opportunities, reinforcing the need for fine-tuned memory access strategies [3].

In conclusion, the analysis validates the importance of adopting advanced CUDA optimization techniques, such as warp-level execution, shared memory utilization, and memory access coalescence. By effectively addressing both memory-bound and compute-bound limitations, the optimized kernels achieve significant performance improvements, providing a robust foundation for scalable and efficient GPU kernel execution.

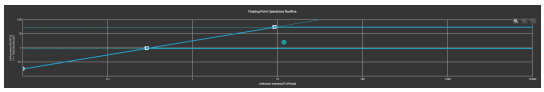

Fig. 6. Roofline analysis for *gaussianBlur_naive* kernel.

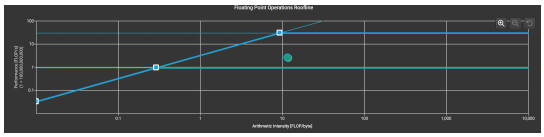

Fig. 7. Roofline analysis for *gaussianBlur_warp* kernel.

D. Discussion

The performance analysis of the *gaussianBlur_naive* kernel and its optimized counterparts demonstrates the effectiveness of various CUDA optimizations. The results show that memory access patterns and compute efficiency are critical factors in achieving high GPU performance. As observed in the Roofline model analysis, the *gaussianBlur_naive* kernel is memory-bound, exhibiting low arithmetic intensity. This limitation highlights the need for optimization strategies that address memory access latency, memory coalescing, and data reuse.

The warp-level optimizations (*gaussianBlur_warp*) improve execution efficiency by reducing divergence and ensuring coalesced memory accesses. Similarly, shared memory tiling (*gaussianBlur_shared*) minimizes redundant global memory accesses, leading to better memory throughput. The *gaussianBlur_memlock* kernel achieves the highest performance through the effective use of memory locking, further optimizing global memory access.

The balance between compute and memory throughput observed in the optimized kernels aligns with established studies on GPU performance optimization [19], [20]. Achieving higher arithmetic intensity and reducing memory bottlenecks are widely recognized as essential steps for leveraging GPU resources efficiently [21]. These observations reinforce the importance of considering both compute-bound and memory-bound aspects when optimizing CUDA kernels.

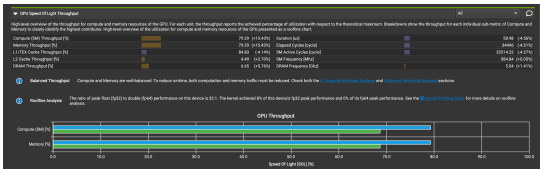

Fig. 8. GPU throughput for *gaussianBlur_warp* kernel.

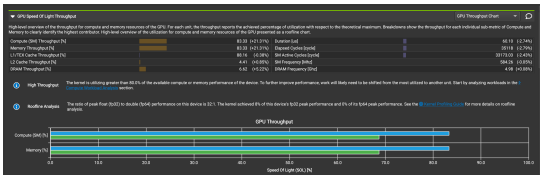

Fig. 9. GPU throughput for *gaussianBlur_shared* kernel.

Despite the improvements, there remain areas for further investigation, particularly in exploring the trade-offs between shared memory usage, occupancy, and thread-level parallelism. Future work can explore more advanced optimization techniques, as discussed in the next section.

E. Future Work

While significant performance gains have been achieved through warp-level optimizations, shared memory tiling, and memory coalescing, further improvements can be pursued through the following approaches:

1) **Kernel Fusion:** Combining multiple kernels into a single kernel to reduce kernel launch overhead and improve data locality [22].
2) **Asynchronous Memory Transfers:** Utilizing CUDA streams and overlapping computation with memory transfers to hide memory latency and improve overall throughput [3].
3) **Dynamic Parallelism:** Implementing dynamic kernel launches to enable finer-grained parallelism and reduce workload imbalance for large-scale data processing [20].
4) **Occupancy Tuning:** Further tuning block size and shared memory usage to maximize SM occupancy while balancing resource utilization [23].
5) **Hybrid Approaches:** Integrating mixed-precision computations or utilizing tensor cores (where applicable) to accelerate operations while maintaining numerical accuracy [24].

These techniques can be systematically applied and analyzed using tools such as NVIDIA Nsight Compute and Nsight Systems to identify new performance bottlenecks and measure improvements. A more comprehensive exploration of advanced memory hierarchies and compute scheduling will provide additional opportunities for achieving peak GPU performance.

VI. Conclusion

In this project, we optimized all CUDA kernels by applying a series of advanced optimization techniques, including warp-level control flow optimization, page-locked memory transfers,

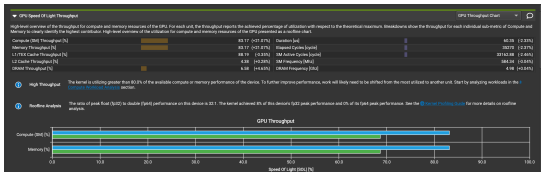

Fig. 10. GPU throughput for *gaussianBlur_memlock* kernel.

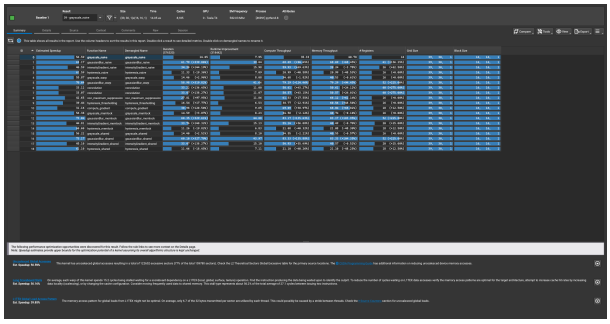

Fig. 11. NCU summary results showing estimated speedups, compute throughput, and memory throughput for various kernels.

shared memory tiling, and stream-based concurrency. The baseline implementations, such as *gaussianBlur_naive*, were found to be predominantly memory-bound with uncoalesced global memory accesses and low arithmetic intensity, limiting their performance. By implementing warp-level optimizations, we reduced divergence and improved execution efficiency. Shared memory tiling further minimized global memory access latency through data reuse within thread blocks, while memory locking techniques enhanced transfer bandwidth between host and device. Additionally, stream optimizations enabled overlapping memory transfers with kernel execution, improving overall throughput.

The results demonstrate significant performance improvements across all optimized kernels. The *gaussianBlur_shared* kernel achieved the highest compute and memory throughput (**83.33%**), demonstrating balanced utilization of GPU resources. Page-locked memory and stream concurrency were effective in addressing Host-to-Device and Device-to-Host transfer bottlenecks, while shared memory optimizations significantly reduced global memory latency. These improvements validate the importance of addressing both compute and memory-bound bottlenecks to achieve efficient GPU performance. Future work can focus on further integrating techniques such as kernel fusion, asynchronous transfers, and mixed-precision computation to enhance scalability and performance for larger and more complex workloads.

VII. Acknowledgements

We sincerely appreciate Professor Kostic for his exceptional guidance and mentorship throughout the course. We also extend our heartfelt thanks to our teaching assistant, Pranav Kota, for his unwavering support and dedication.

REFERENCES

[1] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," in *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3. Springer, 2015, pp. 211–252.

[3] NVIDIA Corporation, *CUDA C Programming Guide*, 2024, version 12.6. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] G. Bradski, "The opencv library," in *Dr. Dobb's Journal of Software Tools*, 2000.

[5] W. SHI, W. CHEN, Y. FAN, J. DONG, S. CAO, and H. XU, "Fpga-based real-time edge detection and its implementation for deep-space images," *Electronic Science and Technology*, vol. 33, no. 5, pp. 45–49, 2020.

[6] W. Jin, Z. Jun, L. Cong, and W. Hanning, "Implementation of sdsoc acceleration algorithm for edge detection algorithm in machine vision," *Computer Engineering and Applications*, vol. 55, no. 12, pp. 208–214, 2019.

[7] X. Keqiang, L. Guangming, L. Renren, W. Zhijun, and X. Jun, "Implementation and optimization of canny operator on dsp," *Modern Electronics Technique*, vol. 37, no. 6, pp. 8–11, 2014.

[8] L. Xiangjiao, L. Guangliang, Z. Xuewu, and G. Jinliang, "The parallel canny algorithm based on tbb," *Journal of Nanyang Institute of Technology*, vol. 6, no. 3, pp. 47–50, 2014.

[9] Z. Yue, W. Xiaohong, and H. Xiaohai, "Real-time image edge detection based on gpu," *Electronic Measurement Technology*, vol. 31, no. 2, pp. 140–142, 2009.

[10] T. Bin and L. Wen, "Fast canny algorithm based on gpu+cpu," *Chinese Journal of Liquid Crystals and Displays*, vol. 31, no. 7, pp. 714–720, 2016.

[11] W. Jin, L. Ying, L. Zhentao, and L. Qiaoshen, "Gpu implementation of machine vision algorithm based on opencl," *Computer Engineering and Design*, vol. 40, no. 2, pp. 346–351, 2019.

[12] L. Mochurad, "Canny edge detection analysis based on parallel algorithm, constructed complexity scale and cuda," *Computing and Informatics*, vol. 41, no. 6, pp. 957–980, 2022.

[13] M. Horvath, B. Michael, and A. Shadi, "Canny edge detection on gpu using cuda," in *IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, 2023, pp. 419–425.

[14] Z. Suwen, C. Zhixing, and S. Yixin, "Improved canny edge detection algorithm and implementation in fpga," *Infrared Technology*, vol. 32, no. 2, pp. 93–96, 2010.

[15] N. Shengxiao, W. Sheng, and Y. Jingjing, "A fast image segmentation algorithm fully based on edge information," *Journal of Computer-Aided Design and Computer Graphics*, vol. 24, no. 11, pp. 1410–1419, 2012.

[16] Z. Fuqiang, Y. Cao, and X. Wang, "Fast and resource-efficient hardware implementation of modified line segment detector," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 11, pp. 3262–3273, 2018.

[17] V. Sivakumar and N. Janakiraman, "A novel method for segmenting brain tumor using modified watershed algorithm in mri image with fpga," *Biosystems*, vol. 198, no. S1, pp. 1–13, 2020.

[18] Z. Hongye, "Optimization identification and simulation about household registration management personal fingerprint image," *Heilongjiang Science*, vol. 11, no. 12, pp. 1–3, 2020.

[19] N. Corporation, *CUDA C Programming Guide*. NVIDIA Corporation, 2021.

[20] M. Harris, "Optimizing parallel reduction in cuda," *NVIDIA Developer Blog*, 2013.

[21] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016.

[22] Z. Jia and H. Zhao, "Optimized parallel reduction using cuda," in *Proceedings of the International Conference on High Performance Computing*, 2012.

[23] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference*, 2010.

[24] N. Corporation, "Accelerating mixed precision training with nvidia tensor cores," *NVIDIA Developer Blog*, 2020.

APPENDIX

TABLE II
INDIVIDUAL STUDENT CONTRIBUTIONS (IN %)

Task	**% Jifeng Li**	**% Rishita Yadav**
Overall	50	50
Coding	60	40
Report	50	50
Presentation	40	60