18645 Project: Optimizing Canny Edge Detection

First 2 Stages: Noise Reduction (Gaussian Blur), Gradient Calculation (Sobel Filter + Gradient Intensity + Gradient Direction)

Ander Zhu
Department of Electrical and Computer Engineering
Carnegie Mellon University Silicon Valley
Mountain View, USA
anderz@andrew.cmu.edu

Yifan Zhao
Department of Electrical and Computer Engineering
Carnegie Mellon University Silicon Valley
Mountain View, USA
yzhao9@andrew.cmu.edu

***Abstract*—This project investigates the design and optimization of the first two stages of the Canny Edge Detection algorithm: Gaussian smoothing and gradient computation. Our objective was to implement a hand-optimized kernel that approaches the architectural performance limits of the target machine. To maximize SIMD utilization, we developed a 10-chain fused multiply-add (FMA) microkernel capable of saturating the processor's FMA pipelines. The resulting implementation achieves substantial speedups over the naive scalar version and delivers partial but meaningful improvements over OpenCV's highly optimized baseline, while still producing numerically correct output. Overall, this work demonstrates the importance of combining algorithmic insight with microarchitectural awareness when designing high-performance image-processing kernels.**

I. Introduction

Edge detection is a foundational operation in computer vision, image analysis, and robotics. The Canny Edge Detection algorithm remains one of the most widely adopted techniques due to its robustness, accuracy, and strong theoretical grounding. The first two stages of the Canny pipeline—Gaussian smoothing and gradient computation—are especially important because they determine the quality of all subsequent steps, including non-maximum suppression and hysteresis thresholding. These stages are also computationally expensive, particularly for large images or real-time requirements, making them prime candidates for SIMD-based acceleration.

Our project focuses on optimizing these two stages using AVX2 SIMD instructions and hand-tuned microkernels designed to approach the architectural limits of the target CPU. For both the Gaussian and Sobel filters, we leverage the separability of the 2D convolution to decompose the computation into independent horizontal and vertical 1D passes. To maximize the use of hardware, we designed a 10-chain fused multiply-add (FMA) microkernel that hides FMA latency and saturates the pipelines. Because both the horizontal and vertical passes operate on rows, the kernel benefits from favorable memory-access patterns in a row-major layout.

The second stage—gradient magnitude and direction calculation—uses the results of the Sobel filters to compute spatial derivatives in the x and y directions. This step is also vectorized using SIMD intrinsics. Our implementation computes magnitude and orientation efficiently while avoiding redundant memory accesses and minimizing pipeline stalls.

Across both stages, we developed a flexible benchmarking harness based on the rdtsc instruction to obtain cycle-accurate performance measurements and compared our optimized kernels against both a naive scalar baseline and OpenCV's highly optimized implementation. The results demonstrate strong performance improvements from our vectorized approach, highlighting the effectiveness of combining algorithmic simplification, SIMD parallelism, and microarchitectural tuning.

II. Design

A. 1D Horizontal Convolution Kernel

The horizontal convolution kernel performs a 1×3 convolution between each row of the input image and the filter kernels. In our implementation, we use both the Gaussian and Sobel 1×3 filters:

$$G_{1\times3} = \frac{1}{4}[1 \quad 2 \quad 1]$$
$$S_{1\times3} = \frac{1}{4}[-1 \quad 0 \quad 1]$$

For each output pixel, the convolution consists of three multiply–accumulate operations. Each neighboring pixel is multiplied by a corresponding filter coefficient, and the results are accumulated to produce the final value. These operations map naturally onto the SIMD fused multiply–add (FMA) unit.

Our horizontal kernel computes a 1×80 output tile (80 pixels) per iteration. This is achieved using 10 vector accumulators, where each AVX2 `YMM` register holds 8 single-precision floating-point values. The kernel requires 14 SIMD registers in total:

- 10 registers for the output accumulators (`acc0`–`acc9`),
- 3 registers for the filter coefficients (`k0`, `k1`, `k2`),
- 1 temporary register (`reg`) for loading input pixels.

Since the target AVX2 machine provides 16 `YMM` registers (`YMM0`–`YMM15`), the design fits comfortably within the hardware constraints. The filter registers are initialized once before entering the main loop, while the accumulators are reset for each 1×80 tile. Ignoring boundary handling for simplicity, each output pixel is computed as

$$\begin{aligned}\text{Output}[i][j] = &\text{Input}[i][j-1] &&\times\text{Filter}[0]\\ &+\text{Input}[i][j] &&\times\text{Filter}[1]\\ &+\text{Input}[i][j+1] &&\times\text{Filter}[2]\end{aligned}$$

Since each output depends only on three neighboring input pixels in the same row, different outputs are independent and can be computed in parallel using SIMD FMA instructions. On our selected machine, a single SIMD FMA instruction has a latency of 5 cycles and a throughput of 2 FMAs per cycle. By maintaining 10 independent accumulation chains (`acc0`–`acc9`), we ensure that the processor always has enough independent work to issue FMAs at maximum throughput. This will ideally saturates the FMA pipelines and prevents pipeline bubbles, enabling the kernel to run near the architectural peak.

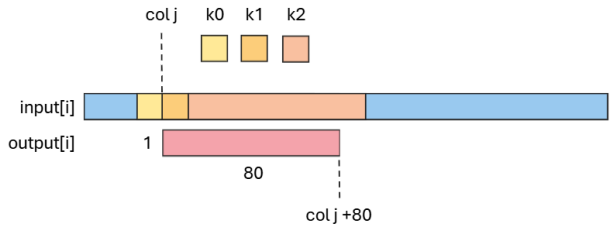

Fig. 1. 1D Horizontal Convolution Kernel.

B. 1D Vertical Convolution Kernel

The vertical convolution kernel performs a 3×1 convolution between the input image matrix and the vertical filter kernels:

$$G_{3\times1} = \frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} \qquad S_{3\times1} = \begin{bmatrix}-1\\0\\1\end{bmatrix}$$

Each output pixel is computed independently as a weighted sum of its vertical neighbors. Since each output depends only on three pixels from rows $(i-1)$, i, and $(i+1)$, the computation of different output columns is independent and can be parallelized across SIMD lanes.

For each output pixel, the convolution involves three multiply–accumulate operations. Each vertically adjacent input pixel is multiplied by a corresponding filter coefficient, and the results are accumulated to produce the final output value. These operations map directly onto SIMD FMA instructions.

The main operations in this kernel are multiplication and addition, both of which are efficiently executed using the SIMD FMA unit. Like the horizontal kernel, the vertical kernel computes an output tile of 80 pixels per iteration. This is achieved using 10 SIMD accumulators, each holding 8 single-precision floating-point values. The register usage is identical to the horizontal convolution kernel:

- 10 registers for output accumulators (`acc0`–`acc9`),
- 3 registers for filter taps (`k0`, `k1`, `k2`),
- 1 temporary load register (`reg`).

Thus, the vertical kernel also requires a total of 14 SIMD registers. The selected AVX2 architecture provides 16 `YMM` registers, so the design fits comfortably. The filter registers are initialized once, and the accumulators are reinitialized for each 1×80 tile. Ignoring boundary handling, each output pixel in the vertical pass is computed as:

$$\begin{aligned}\text{Output}[i][j] = &\text{Input}[i-1][j] &&\times\text{Filter}[0]\\ &+\text{Input}[i][j] &&\times\text{Filter}[1]\\ &+\text{Input}[i+1][j] &&\times\text{Filter}[2]\end{aligned}$$

Even though the kernel is vertical, we still vectorize across columns. Thus, each iteration processes 80 horizontally adjacent pixels, but the three input vectors come from three different rows: $(i-1)$, i, and $(i+1)$.

On the selected AVX2 machine, each SIMD FMA instruction has a latency of 5 cycles and a throughput of 2 FMAs per cycle. By maintaining 10 independent accumulation chains (`acc0`–`acc9`), the vertical kernel, like the horizontal one, issues 10 independent FMAs per tap. This ideally keeps both FMA pipelines fully utilized, minimizing pipeline bubbles and achieving near-peak throughput.

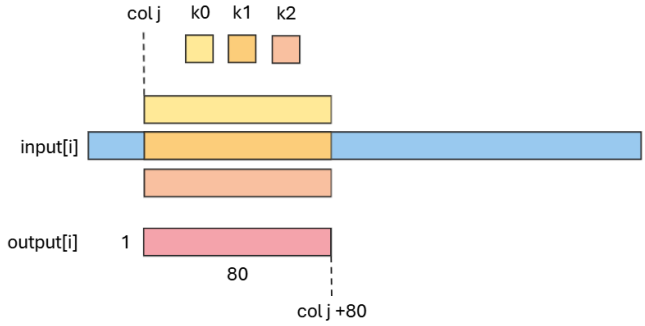

Fig. 2. 1D Vertical Convolution Kernel.

C. Gradient Magnitude Kernel

1) Kernel 3: Squared Magnitude: This kernel computes the squared gradient magnitude for each pixel using the horizontal (G_x) and vertical (G_y) gradient components.

- **Core Computation:** Calculates the squared magnitude, $M^2 = G_x^2 + G_y^2$. The square-root operation is intentionally omitted to avoid an expensive floating-point operation, as only relative magnitudes are compared in later stages.
- **Independent Operations:** Each output pixel, $out[i]$, is computed independently based on its local $G_x[i]$ and $G_y[i]$ values. There is no data dependency between pixels.
- **Dependent Instructions:** For each pixel, two multiply operations (for squaring) and one add operation (for summation) are required. This is optimally implemented as a single Fused Multiply-Add (FMA) instruction using the relationship $out = G_y \times G_y + G_x^2$.

- **Functional Unit(s):** Utilizes **SIMD FMA units** (AVX2 `vfmadd` instruction) for element-wise squaring and summation in a single, high-throughput operation.

a) SIMD Implementation Details (AVX2):

Feature	Detail
Ideal Kernel Size	10 SIMD ymm256 registers (80 singles), limited by register file.
Actual Kernel Size	5 SIMD ymm256 registers (40 singles) due to the 16 available SIMD registers.
Register Allocation	ymm0–ymm4: Output (5 vectors)
	ymm5–ymm9: G_x Input (5 vectors)
	ymm10–ymm14: G_y Input (5 vectors)

D. Gradient Direction Kernel

1) Kernel 4: Compare Direction: This kernel determines the quantized gradient direction of each pixel based on the horizontal (G_x) and vertical (G_y) gradient components.

- **Core Computation:** Classifies the gradient into one of four principal directions ($0°, 45°, 90°, 135°$) by comparing the absolute magnitudes of G_x and G_y against constant tangent thresholds. This approximation avoids the computationally expensive arctan function.
- **Approximation:** The classification relies on simple geometric ratios. The boundaries for the four directions are defined by the slope ratios $\frac{|G_y|}{|G_x|}$ being compared against $\tan(22.5°)$ and $\tan(67.5°)$.
- **Independent Operations:** Each pixel's direction code is determined independently, making the operation perfectly parallel.
- **Functional Unit(s):** Primarily relies on **SIMD Comparison (CMP) and Blending (BLENDF) instructions**. Due to the dependent chain of comparisons and blends, the vectorization benefits are limited by the instruction's reciprocal throughput. Performance relies mainly on the intra-vector parallelism (8 lanes per `__m256`).

a) SIMD Implementation Details (AVX2):

Feature	Detail
Kernel Size	1 SIMD ymm256 register (8 singles) per iteration.
Inputs/Outputs	2 inputs (G_x, G_y), 1 output (dir code).
Key Constants	$\tan(22.5°) \approx 0.414$, $\tan(67.5°) \approx 2.414$, and direction codes (0.0f, 1.0f, 2.0f, 3.0f).

III. Parallelization

A. OpenMP Optimization for Kernel 3: Squared Magnitude

The independent nature of the computation (each output pixel depends only on local inputs) makes this an ideal candidate for OpenMP's data parallelism.

The `#pragma omp parallel for schedule(static)` directive is used on the vectorized loop. `schedule(static)` is optimal because the workload (40 elements) is uniform for every loop iteration, which ensures efficient load balancing across all threads.

B. OpenMP Optimization for Kernel 4: Compare Direction

Each pixel's output is determined independently by its two local inputs (G_x and G_y).

The optimal strategy is to use the `#pragma omp parallel for schedule(static)` directive on the main SIMD loop to distribute the uniform, step-by-step workload across multiple cores for maximum efficiency.

IV. Performance

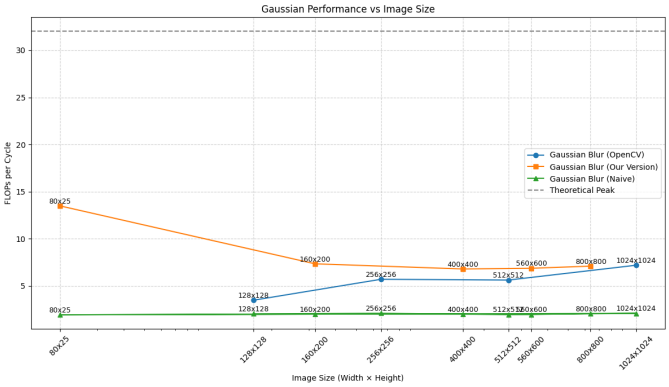

Fig. 3. Gaussian Kernel Performance.

Figure 3 compares the FLOPs-per-cycle performance of three Gaussian blur implementations across a range of image sizes. As expected, the naïve baseline remains flat at around 2 FLOPs/cycle, showing minimal SIMD utilization. The OpenCV implementation performs noticeably better, achieving between 3.5–7 FLOPs/cycle, but still falls well below the hardware's theoretical limit due to its more general-purpose design.

Our optimized Gaussian kernel consistently outperforms both baselines. On a very small input (80×25), which fits entirely within the L1 cache, our kernel reaches approximately 14 FLOPs/cycle, demonstrating excellent utilization of vector units and effective pipeline saturation in steady state. However, this value remains below the theoretical peak throughput of 32 FLOPs/cycle. A likely explanation is the unavoidable repeated loading and storing of data within each 1D pass, which

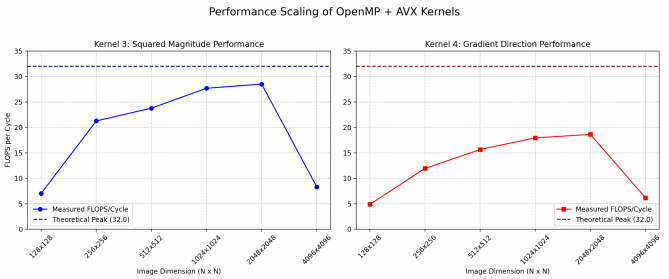

Fig. 4. Enter Caption

introduces memory traffic that prevents the FMA pipelines from being fully saturated.

As image sizes increase, performance gradually converges toward about 7 FLOPs/cycle. This trend is expected because larger images exceed L1 capacity, causing more cache misses and reducing arithmetic intensity. Even so, our kernel maintains a clear margin over OpenCV across all tested resolutions, highlighting the benefit of specialized tiling, controlled register usage, and reduced overhead in our SIMD10 design.

Figure 4 shows the performance scaling of Kernel 3 (squared magnitude) and Kernel 4 (gradient direction). Kernel 3 achieves significantly higher FLOPs/cycle because its computation is dominated by FMAs, which map efficiently onto AVX2 and exhibit high arithmetic intensity. As image size increases, its performance rises toward 28–29 FLOPs/cycle before dropping at 4096×4096 due to cache pressure and memory-bandwidth limits. Kernel 4, in contrast, performs mostly comparisons and blends, which have lower throughput and limited instruction-level parallelism, resulting in a lower peak of about 18–19 FLOPs/cycle. Both kernels show reduced performance on the largest images as they transition from compute-bound to memory-bound behavior.

V. Future Directions

While the current implementation delivers strong performance for the first two stages of Canny Edge Detection, several promising directions remain for future exploration:

A. Full Canny Pipeline Optimization

A natural next step is extending the same microarchitectural optimization strategy to the remaining stages of the Canny algorithm, including non-maximum suppression, double thresholding, and edge hysteresis. These stages involve less regular computation patterns and more branching, so achieving high SIMD utilization will require careful restructuring (e.g., branch elimination, mask-based operations). Completing the full pipeline would allow end-to-end performance evaluation and enable comparison against state-of-the-art GPU and CPU implementations.

B. Kernel Fusion and Reduced Memory Traffic

Currently, each stage of the pipeline (Gaussian, Sobel X/Y, magnitude, direction) writes intermediate results back to memory. Fusing compatible stages—such as combining the vertical Sobel pass with the gradient magnitude computation—could significantly reduce memory bandwidth pressure. Exploring fusion opportunities and restructuring the algorithm to reduce load/store operations may close the gap between our achieved performance and the architectural peak.

C. Cache Blocking and Multi-Level Tiling

Although the current SIMD kernels already achieve good L1 locality for small tiles, larger images suffer reduced arithmetic intensity once data spills into lower cache levels. Future work could implement multi-level blocking strategies that explicitly target L1, L2, and L3 cache sizes. This could minimize memory stalls, particularly in the vertical convolution phase where stride access patterns are less favorable.

D. AVX-512 and Architecture-Specific Variants

While our design targets AVX2, modern CPUs increasingly support AVX-512 with more registers, wider vectors, and masking support. Porting the kernels to AVX-512 could:

- increase tile size from 80 to 160 pixels,
- expand the number of independent accumulation chains,
- reduce instruction overhead with native mask operations.

Similarly, designing separate kernels for ARM NEON or Apple M-series architectures would broaden usability and allow cross-architecture performance comparisons.