Performance Analysis of Auto, Guided, and Explicit Vectorization on an Edge-Detection Kernel Using Intel Advisor

Author One[1], Author Two[2]

Resumen— **Edge detection is a fundamental numerical kernel in computer vision and image processing. This work implements the gradient-magnitude edge detector $G[i] = \sqrt{G_x[i]^2 + G_y[i]^2}$ in modern C++ in four variants: scalar baseline, compiler auto-vectorization, guided (OpenMP SIMD), and explicit (Intel AVX2) vectorization. We analyse performance and vectorization efficiency with Intel Advisor and report execution times and vectorization percentages. [Placeholder: fill with one-sentence summary of main result once data available.]**

Palabras clave—**Edge detection, gradient magnitude, Sobel kernel, SIMD, auto-vectorization, OpenMP SIMD, Intel AVX2, Intel Advisor, performance analysis.**

I. Introduction

EDGE detection is a fundamental operation in computer vision, medical image processing, robotic perception, and as a preprocessing stage for convolutional neural networks. Edges mark sharp changes in intensity that typically correspond to object boundaries, and allow an image to be reduced to structural information: shapes, contours, and regions. From a computational standpoint, edges are characterised by a high gradient magnitude, so standard approaches use operators such as Sobel, Prewitt, or Canny [?].

The numerical kernel we focus on is the gradient-magnitude computation: at each pixel i, the edge strength is given by

$$G[i] = \sqrt{G_x[i]^2 + G_y[i]^2}, \qquad (1)$$

where G_x and G_y are the horizontal and vertical gradients obtained by convolving the image with small kernels (e.g. Sobel 3×3). This kernel is compute-bound and highly regular, making it a good candidate for SIMD vectorization on modern CPUs.

The goal of this work is to compare three vectorization strategies—automatic (compiler-only), guided (directives such as OpenMP SIMD), and explicit (hand-coded SIMD intrinsics)—on a C++ implementation of this kernel, and to analyse their performance and vectorization efficiency using Intel Advisor. We implement a scalar baseline plus the three vectorized versions, run them under Intel Advisor to obtain execution times and vectorization metrics, and present the results in this article. The remainder of the document is organised as follows: Section II reviews the kernel and vectorization concepts; Section III describes the implementation and the Advisor-based methodology; Section IV reports the results with placeholders for the measured data; Section V summarises the findings and outlines future work.

[1]Department, University, e-mail: `author1@univ.edu`.
[2]Department, University, e-mail: `author2@univ.edu`.

II. Background and related work

A. Gradient-magnitude edge detection

An edge is defined as a strong local change in image intensity. For a discrete 2D image, the gradient is approximated by convolving the image with derivative kernels. The Sobel operator uses two 3×3 kernels to compute G_x (horizontal gradient) and G_y (vertical gradient). The magnitude G in Eq. (1) is then a scalar measure of edge strength at each pixel. Convolution is implemented as a double loop over the image with a 3×3 neighbourhood; the magnitude step is a single pass over the pixels. Both phases are data-parallel and exhibit regular memory access, which is favourable for SIMD and for compiler auto-vectorization [?].

B. Vectorization strategies

Auto-vectorization: The compiler (e.g. Intel oneAPI `icpx` or GCC with `-O3 -march=native`) attempts to map scalar operations to SIMD instructions without source changes. Effectiveness depends on loop structure, alignment, and absence of dependencies that prevent vectorization.

Guided (implicit) vectorization: The programmer adds directives such as `#pragma omp simd` to mark loops as vectorisable, helping the compiler and optionally enabling OpenMP SIMD-specific optimisations [?].

Explicit vectorization: The programmer uses SIMD intrinsics (e.g. Intel AVX/AVX2 `_mm256_*` for 256-bit vectors) to implement the kernel by hand [?]. This gives full control and can achieve high utilisation when the compiler or guided approach falls short, at the cost of portability and maintainability.

Intel Advisor [?] provides roofline-style analysis, vectorisation advice, and metrics such as vectorisation coverage and loop body efficiency, which we use to interpret the performance of each variant.

III. Methodology

A. Scalar implementation

We implemented the edge-detection kernel in modern C++ (C++17). The scalar version consists of:

1. Loading a grayscale image (PGM format) into a contiguous buffer of `double`.
2. Convolving the image with the Sobel G_x and G_y 3×3 kernels with same-size output (border pixels are skipped or replicated). Two nested loops over rows and columns compute each output pixel from a 9-tap stencil.
3. Computing the gradient magnitude $G[i] = \sqrt{G_x[i]^2 + G_y[i]^2}$ in a single loop over all pixels.

No SIMD directives or intrinsics are used; this serves as the baseline for comparison and for Intel Advisor's "no vectorization" reference.

B. Auto-vectorized version

The same source code is compiled with aggressive optimisation and architecture-specific flags: `-O3 -march=native` (and with Intel oneAPI, `icpx` is used). No pragmas or intrinsics are added. The compiler is expected to auto-vectorise the convolution and magnitude loops where legal. We refer to this variant as "auto" in the results.

C. Guided (implicit) vectorization

A second variant is built from the same algorithmic structure but with `#pragma omp simd` applied to the inner loop of the convolution and to the magnitude loop. The project is linked with OpenMP (`-fiopenmp` for Intel oneAPI). This guides the compiler to generate SIMD code for those loops and allows Advisor to report on guided vectorisation.

D. Explicit vectorization

A third variant implements the convolution and magnitude computation using Intel AVX2 intrinsics (`_mm256_*`). Four `double` values are processed per iteration (256-bit registers). The convolution inner loop is unrolled to compute four consecutive pixels at a time; the remainder is handled with a scalar epilogue. The magnitude step uses `_mm256_loadu_pd`, `_mm256_mul_pd`, `_mm256_add_pd`, and `_mm256_sqrt_pd`, with a scalar tail. This variant is compiled with `-O3 -march=native` so that FMA and other instructions are available.

E. Performance analysis with Intel Advisor

For each variant we:

1. Build the executable with the appropriate flags (scalar: no SIMD; auto: `-O3 -march=native`; guided: plus `-fiopenmp`; explicit: `-O3 -march=native`).
2. Run the binary under Intel Advisor (Survey and/or Roofline) on a fixed input image and record the execution time reported.
3. Use Advisor's vectorisation and roofline reports to obtain metrics such as percentage of code vectorised, loop efficiency, and memory vs compute bounds.

Input images are PGM grayscale; image size and path are kept consistent across runs so that times and metrics are comparable. **Placeholder:** The exact Advisor workflow (e.g. `advixe-cl -collect survey`, then `advixe-cl -report survey`) and command-line options will be filled in once the experiments are run.

IV. Experimental results

Experiments were run on [**PLACEHOLDER: machine description, e.g. CPU model, number of cores, RAM**]. The input image used for all runs is [**PLACEHOLDER: image size, e.g. 1024×1024 or 4096×4096**] pixels. Each configuration was executed multiple times; the values below correspond to [**PLACEHOLDER: e.g. median of 5 runs / average of 10 runs**].

A. Execution time

Table I reports the execution time (in [**PLACEHOLDER: units, e.g. milliseconds (ms) or microseconds (μs)**]) for the scalar baseline and the three vectorized versions. **Placeholder values are used; replace with actual measurements.**

Tabla I: Execution time per variant. [**REPLACE T1–T4 with measured values.**]

Variant	Time	Unit
Scalar	T1	[**e.g. ms**]
Auto	T2	[**e.g. ms**]
Guided (OpenMP SIMD)	T3	[**e.g. ms**]
Explicit (AVX2)	T4	[**e.g. ms**]

Summary: execution time for the scalar version is T1; for the auto-vectorized version, T2; for the guided (implicit) version, T3; and for the explicit (AVX2) version, T4. [**PLACEHOLDER: Once data are available, add one or two sentences comparing T1–T4, e.g. "The explicit version is the fastest, with a X.X× speedup over scalar."**.]

B. Intel Advisor vectorization metrics

Intel Advisor was used to collect survey and, where applicable, roofline data for each build. Table II summarises the vectorization-related metrics. **Replace placeholder labels with the actual metrics reported by Advisor (e.g. "Vectorization %", "Loop body efficiency", "Estimated potential gain").**

Tabla II: Intel Advisor vectorization metrics. [**REPLACE P1–P4 and any other columns with real metric names and values.**]

Variant	Vectorization percent	[**Other metric**]
Scalar	P1	[–]
Auto	P2	[–]
Guided	P3	[–]
Explicit	P4	[–]

Vectorization percent for the scalar version is P1 (expected 0% or N/A); for the auto-vectorized version, P2; for the guided version, P3; and for the explicit version, P4. [**PLACEHOLDER: Add a

short interpretation, e.g. "Advisor reports that the magnitude loop is fully vectorised in the explicit build, whereas the auto build only vectorises it partially.".]

C. Speedup and comparison

[**PLACEHOLDER: Fill with computed speedups once T1–T4 are known.**]

- Speedup of auto vs scalar: [**SPEEDUP_AUTO = T1/T2**].
- Speedup of guided vs scalar: [**SPEEDUP_GUIDED = T1/T3**].
- Speedup of explicit vs scalar: [**SPEEDUP_EXPLICIT = T1/T4**].
- Relative improvement of explicit with respect to guided (implicit): **"...a improvement of X.X of the explicit version with respect to the implicit (guided) version..."** (e.g. 1.2× faster, or "X.X% reduction in time").

[**PLACEHOLDER: If you have roofline or other Advisor figures, add them here and reference as Fig. ?? or similar.**]

V. Conclusions

This work implemented the gradient-magnitude edge-detection kernel $G[i] = \sqrt{G_x[i]^2 + G_y[i]^2}$ in modern C++ in four variants: scalar baseline, compiler auto-vectorization, guided (OpenMP SIMD), and explicit (Intel AVX2) vectorization. Performance and vectorization efficiency were analysed with Intel Advisor.

[**PLACEHOLDER: Summarise the main quantitative result once data are available.**] For instance: the measured execution times are T1 (scalar), T2 (auto), T3 (guided), and T4 (explicit) in the chosen units. The explicit version achieves the best time (T4), and it is observed an improvement of X.X of the explicit version with respect to the implicit (guided) version [**e.g. "1.25× faster" or "about 20% faster".**], and an improvement of X.X with respect to the auto-vectorized version. [**Adjust these sentences with the actual T1–T4 and computed X.X values.**]

[**PLACEHOLDER: Interpretation of Advisor results.**] The vectorization percent reported by Intel Advisor is P1 for scalar, P2 for auto, P3 for guided, and P4 for explicit. [**Add one or two sentences interpreting what this means: e.g. that the compiler auto-vectorised the magnitude loop but not fully the convolution; that the guided pragmas increased vectorisation; that the explicit version shows high (e.g. 100%) vectorisation in the hot loops.**]

[**PLACEHOLDER: Brief takeaway.**] In summary, guided and explicit vectorization both improve over the scalar and auto versions for this kernel; the explicit AVX2 implementation gives the highest control and typically the best performance at the cost of portability. Future work could include larger images, other kernels (e.g. full Canny), or comparison with GPU implementations.

Acknowledgements

To be completed if applicable.