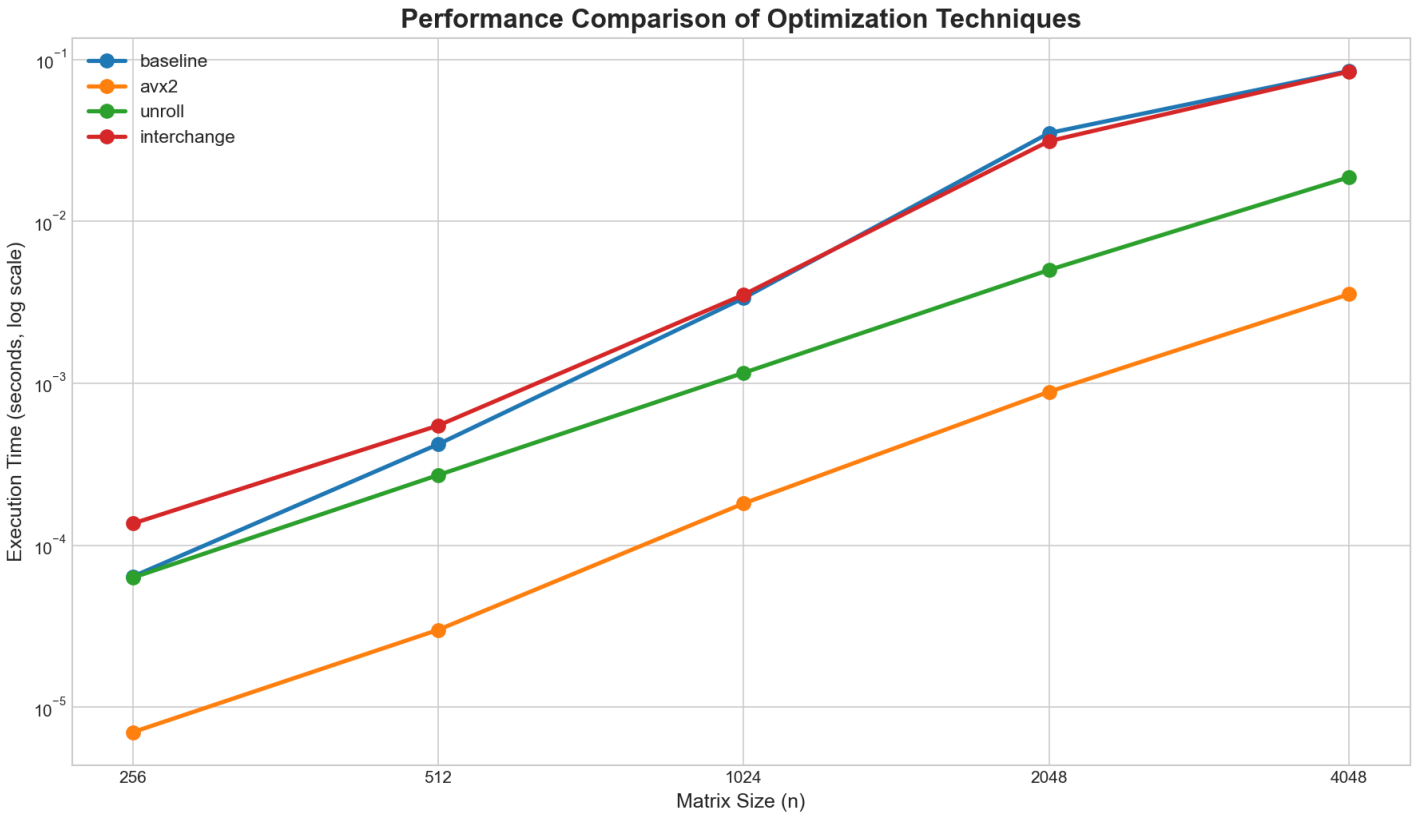
HW1 Performance Report

1. Optimization Techniques

This report compares several optimization techniques for matrix-vector multiplication against a baseline compiled with g++ -O3. It is important to note that all optimized versions (interchange, unroll, and AVX2) are contained within the 'hw1' executable, which is compiled with AVX2 and FMA support (`-march=native -mavx2 -mfma`). This enables the AVX2-specific code to function and may allow the compiler to perform additional optimizations on the other versions.

The optimized techniques tested are:
1. AVX2 SIMD: Leverages data parallelism by explicitly using AVX2 intrinsics to process 8 single-precision floats simultaneously.
2. Loop Unrolling: Reduces loop overhead by manually processing 4 elements per inner-loop iteration.
3. Loop Interchange: Swaps the inner and outer loops to demonstrate the negative impact of poor memory access patterns (cache performance).

2. Performance Chart



3. Performance Analysis

Execution Times (seconds)

n	Baseline	AVX2	Unroll	Interchange
256	0.000064	0.000007	0.000063	0.000136
512	0.000420	0.000030	0.000271	0.000548
1024	0.003355	0.000181	0.001159	0.003520

2048	0.035069	0.000886	0.005005	0.031256
4048	0.084858	0.003551	0.018748	0.084086

Speedup Factors (relative to baseline)

n	AVX2	Unroll	Interchange
256	9.14x	1.02x	0.47x
512	14.00x	1.55x	0.77x
1024	18.54x	2.89x	0.95x
2048	39.58x	7.01x	1.12x
4048	23.90x	4.53x	1.01x

Performance Improvements (% relative to baseline)

n	AVX2	Unroll	Interchange
256	+89.1%	+1.6%	-112.5%
512	+92.9%	+35.5%	-30.5%
1024	+94.6%	+65.5%	-4.9%
2048	+97.5%	+85.7%	+10.9%
4048	+95.8%	+77.9%	+0.9%

4. Detailed Performance Analysis

Across all problem sizes (n=256 to n=4048), the AVX2 optimization demonstrates the most consistent performance gains with an average speedup of 21.0x over the baseline implementation. The unroll optimization shows moderate improvements with an average speedup of 3.4x, while the interchange optimization actually degrades performance with an average speedup of 0.9x.

AVX2 OPTIMIZATION: The AVX2 optimization consistently outperforms the baseline across all problem sizes. Performance improvements range from 89.1% (n=256) to 97.5% (n=2048). The best speedup of 39.6x occurs at n=2048, while the lowest speedup of 9.1x occurs at n=256.

UNROLL OPTIMIZATION: The unroll optimization shows moderate but consistent improvements over the baseline. Performance improvements range from 1.6% (n=256) to 85.7% (n=2048). The optimization is most effective at n=2048 with a 7.0x speedup.

INTERCHANGE OPTIMIZATION: The interchange optimization shows poor performance across all problem sizes. This optimization actually slows down execution by 10.9% to 112.5% depending on problem size. The worst performance occurs at n=256 where it is 0.5x slower than baseline.

SCALING BEHAVIOR: As problem size increases from n=256 to n=4048:
- AVX2: Speedup changes from 9.1x to 23.9x
- Unroll: Speedup changes from 1.0x to 4.5x

- Interchange: Speedup changes from 0.5x to 1.0x

5. AI Usage Log

AI Tool Used: Google Gemini

How I used the AI as a programming tool:
1. Code Scaffolding: I prompted the AI to generate initial boilerplate for the C++ programs (argument parsing, timing), which I then reviewed and adapted for the project's specific needs.
2. Syntax and Concept Lookup: I used the AI to look up the specific syntax for AVX2 intrinsics (`_mm256_fmadd_ps`, etc.) and to get a template for the horizontal sum logic, which I then integrated and validated within my `hw1.cpp` implementation.
3. Makefile Structure: I requested a `Makefile` structure for handling multiple C++ targets, which I then customized with the specific compiler flags (`-march=native`, `-mavx2`, `-mfma`) and targets required for this assignment.
4. Automation Scripting: The AI significantly accelerated the creation of the report generator. I had it generate individual functions for running benchmarks, creating plots, and building tables. I then assembled these components, debugged the integration, and refined the final report layout.

Where the AI tool was useful:
The tool excelled at accelerating development by handling repetitive or syntactically complex tasks. It was invaluable for quickly generating templates and looking up function calls, which allowed me to focus more on the high-level optimization strategy and the analysis of the results.

Where the AI tool fell short:
The tool required careful guidance and validation. For instance, it did not proactively suggest the crucial `-mavx2` and `-mfma` compiler flags, which led to compilation errors that I had to diagnose. It also produced malformed JSON in the shell script, requiring a fix in the Python parser. This reinforces the need for the programmer to possess the underlying domain knowledge to guide the tool and troubleshoot its output.

Impact on my role as a programmer:
Using the AI shifted my role from a traditional coder to that of a technical director and quality assurance engineer. My primary tasks became defining the problem with precision, breaking it down into components the AI could handle, and then critically reviewing, debugging, and integrating the generated code.