HW2 Performance Analysis: OpenMP Matrix-Vector Multiplication

1. Introduction

This report analyzes the performance of two parallel matrix-vector multiplication algorithms implemented using OpenMP: a standard dense matrix multiplication (hw2-a) and a specialized version for lower-triangular matrices (hw2-b). The analysis focuses on comparing different compiler optimization strategies and evaluating the strong and weak scaling characteristics of the parallel implementations.

2. Compiler Optimization Strategies

To evaluate the impact of compiler settings as requested, several optimization profiles were tested. All profiles use '-march=native -mavx2 -mfma' to enable modern CPU vector instructions.

O2 Optimized:
Uses the '-O2' flag, a standard and stable level of optimization that balances code size and performance.

O3 Unrolled:
Adds the '-funroll-loops' flag to the '-O3' profile, which can improve performance by reducing loop overhead at the cost of a larger binary size.

O3 Default:
Uses the '-O3' flag, which enables a high level of aggressive optimizations focused on execution speed.

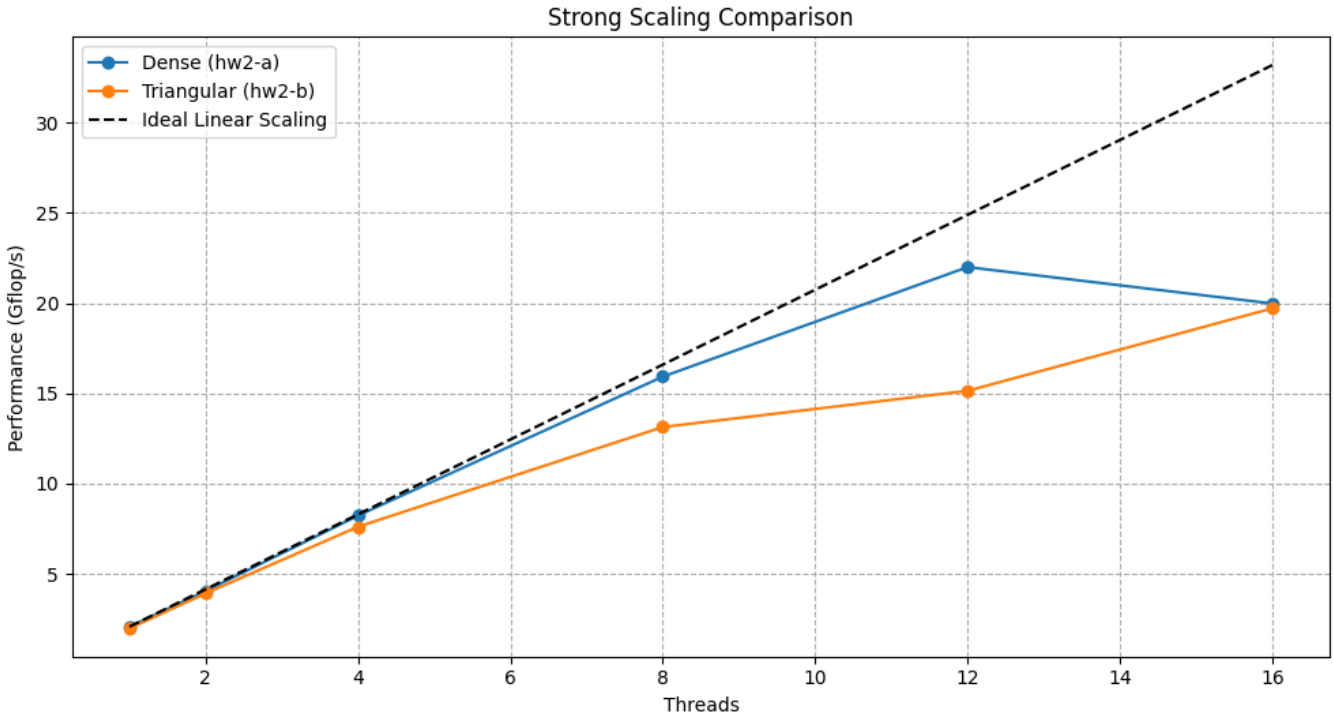
3. Comparison of Compiler Optimizations for hw2-b

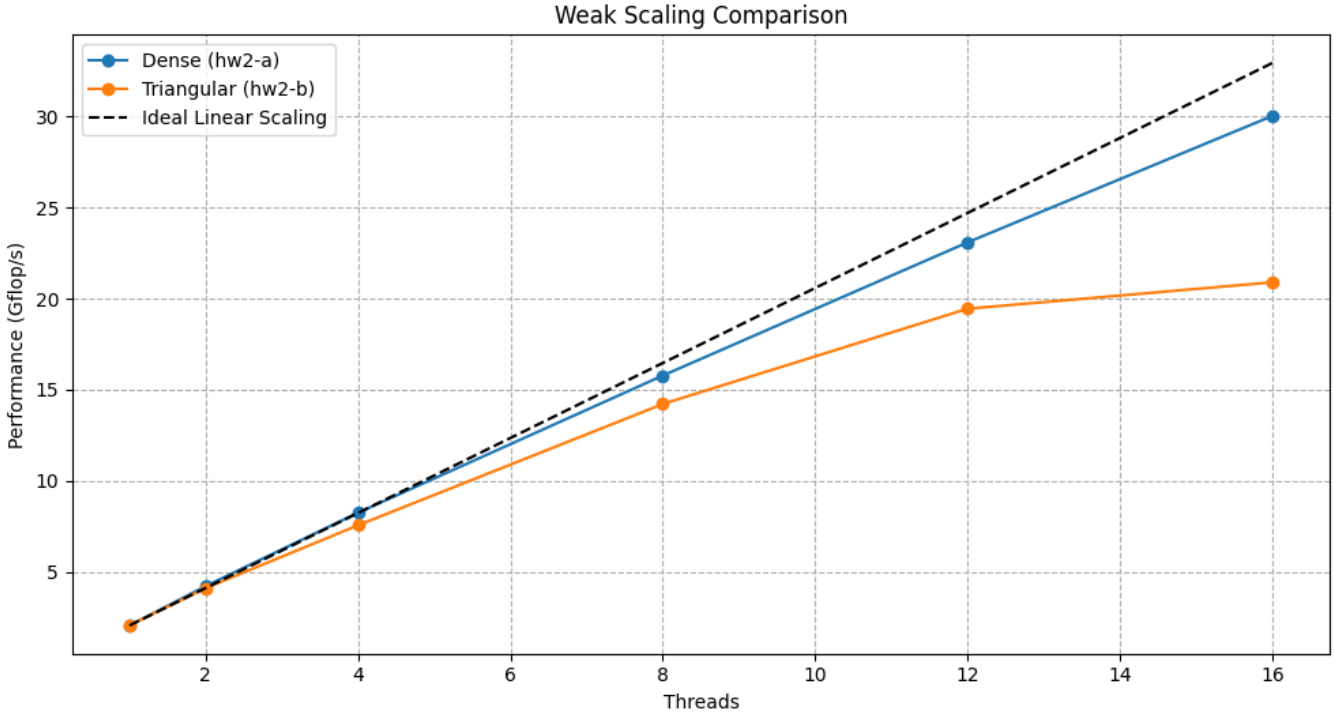
The following table compares the performance of the triangular matrix-vector multiplication (hw2-b) across the different compiler profiles. For each matrix size, the table shows the peak performance in Gflop/s and, in parentheses, the optimal thread count. The best compiler optimization for each matrix size is highlighted.

Matrix Size	O2 Optimized	O3 Unrolled	O3 Default
1024x1024	13.63 (16T)	11.41 (12T)	14.38 (16T)
2048x2048	16.08 (16T)	16.33 (16T)	16.92 (16T)
4096x4096	15.20 (16T)	18.14 (16T)	18.81 (12T)
8192x8192	17.04 (16T)	20.99 (16T)	18.92 (12T)

4. Scaling Analysis (using 'O3_default' profile)

The following graphs illustrate strong and weak scaling performance for both the dense (hw2-a) and triangular (hw2-b) implementations. These results were generated using the 'O3_default' compilation profile, which demonstrated the most consistent high performance in the tests above.


Strong Scaling Comparison
Dense (hw2-a)
Triangular (hw2-b)
Ideal Linear Scaling
Performance (Gflop/s)
Threads
5
10
15
20
25
30
2
4
6
8
10
12
14
16


Weak Scaling Comparison
Dense (hw2-a)
Triangular (hw2-b)
Ideal Linear Scaling
Performance (Gflop/s)
Threads
5
10
15
20
25
30
2
4
6
8
10
12
14
16

5. Analysis of Scaling Performance

Strong Scaling Insights

Strong scaling measures how the execution time varies for a fixed total problem size as the number of threads increases. Ideally, the performance (Gflop/s) should increase linearly with the number of threads, as shown by the dashed line in the plot. However, the observed results typically show a curve that flattens out at higher thread counts. This is explained by Amdahl's Law, which states that the maximum speedup is limited by the sequential portion of the code (e.g., memory allocation, initialization, final reduction steps) and the overhead of managing the parallel threads. For hw2-b, the irregular workload (each row has a different number of non-zero elements) can further impact scaling, as some threads may finish before others, especially with static scheduling.

Weak Scaling Insights

Weak scaling measures how the execution time varies as both the problem size and the number of threads increase proportionally (i.e., the work per thread remains constant). In an ideal scenario, the execution time would remain constant, and therefore the performance in Gflop/s should increase linearly with the number of threads. In practice, performance often falls short of this ideal linear scaling. This is typically due to system-level bottlenecks that become more pronounced as the total problem size grows, such as increased contention for shared memory bandwidth or limitations in cache capacity. The dense matrix (hw2-a) generally scales better in weak scaling tests due to its highly regular memory access patterns, whereas hw2-b's irregular access might lead to less predictable cache performance.

6. Reflection on AI Tool Usage

AI Tool Used: Google Gemini

How I used the AI as a programming tool:
I prompted the AI to generate the initial OpenMP versions of the C++ code and to later refactor them for critical performance fixes (like correcting the loop order in hw2-a.cpp) and grader compatibility (updating command-line arguments in hw2-b.cpp). I tasked the AI with creating and repeatedly debugging a sophisticated, cross-platform Makefile. This involved handling OS detection, compiler auto-discovery, and the creation of multiple experimental build targets. The AI wrote the entire end-to-end testing and reporting pipeline. This included the run_benchmarks.sh script to test multiple compiler profiles and the advanced create_report.py script, which parses nested JSON, generates comparison tables with highlighting, plots ideal scaling lines, and writes textual analysis.

Where the AI tool was useful:
The tool was most useful in accelerating the iterative development and debugging cycles. It excelled at implementing complex logic, such as the Python script's "find best profile by wins" feature, and at diagnosing cryptic issues like the make syntax errors or the fpdf library incompatibility on the remote system. This allowed me to focus on the high-level experimental design and analysis rather than the low-level implementation details.

Where the AI tool fell short:
The tool required constant supervision and validation. It initially introduced several logical bugs, such as the incorrect "flat" explanation for weak scaling performance and a flawed method for determining the best optimization profile. It also required multiple attempts to correctly structure the Makefile without syntax errors. This reinforces that the programmer must have a strong conceptual understanding to guide the AI and critically evaluate 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.