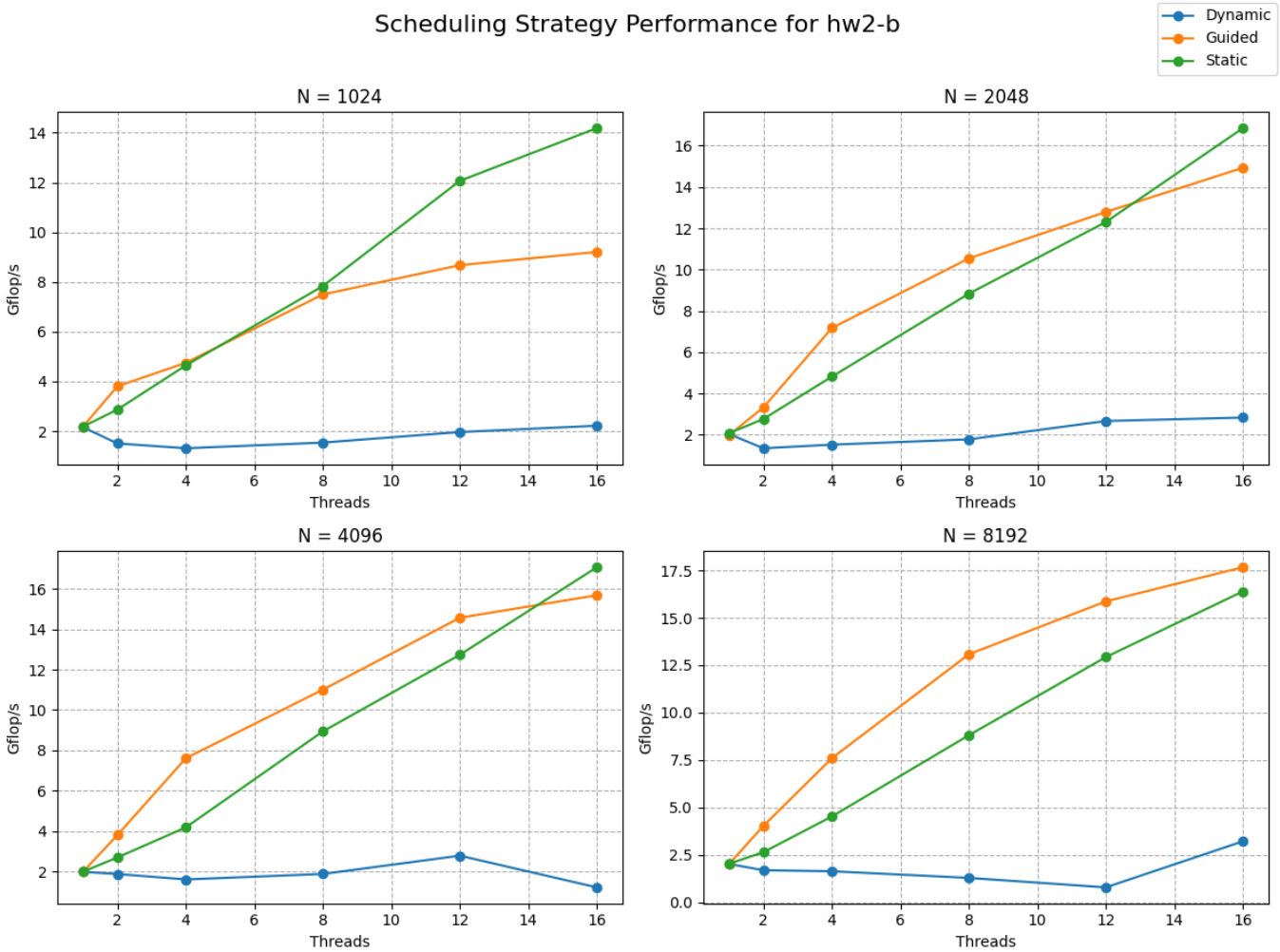
HW2 Performance Report

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, evaluating OpenMP scheduling policies, and analyzing the scaling characteristics of the parallel implementations.

2. Analysis of OpenMP Scheduling Strategies for hw2-b



The chart above visually compares the performance of the three main OpenMP scheduling strategies across different matrix sizes. A brief summary of each strategy:

- static: Iterations are divided into fixed-size chunks. Best for balanced workloads but can be inefficient here.
- dynamic: Threads request work as they become free. Adapts to imbalanced loads but has higher overhead.
- guided: A hybrid approach where chunk sizes decrease over time, balancing overhead and load adaptability.

As shown, the 'guided' scheduling strategy consistently provided the best overall performance. This is likely because it offers a good compromise between the low overhead of static scheduling and the load-balancing of dynamic scheduling, which is ideal for the imbalanced workload of a triangular matrix.

3. Compiler Optimization Strategies

To evaluate the impact of compiler settings, several optimization profiles were tested. All profiles use '-march=native

-mavx2 -mfma' to enable modern CPU vector instructions.

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

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.

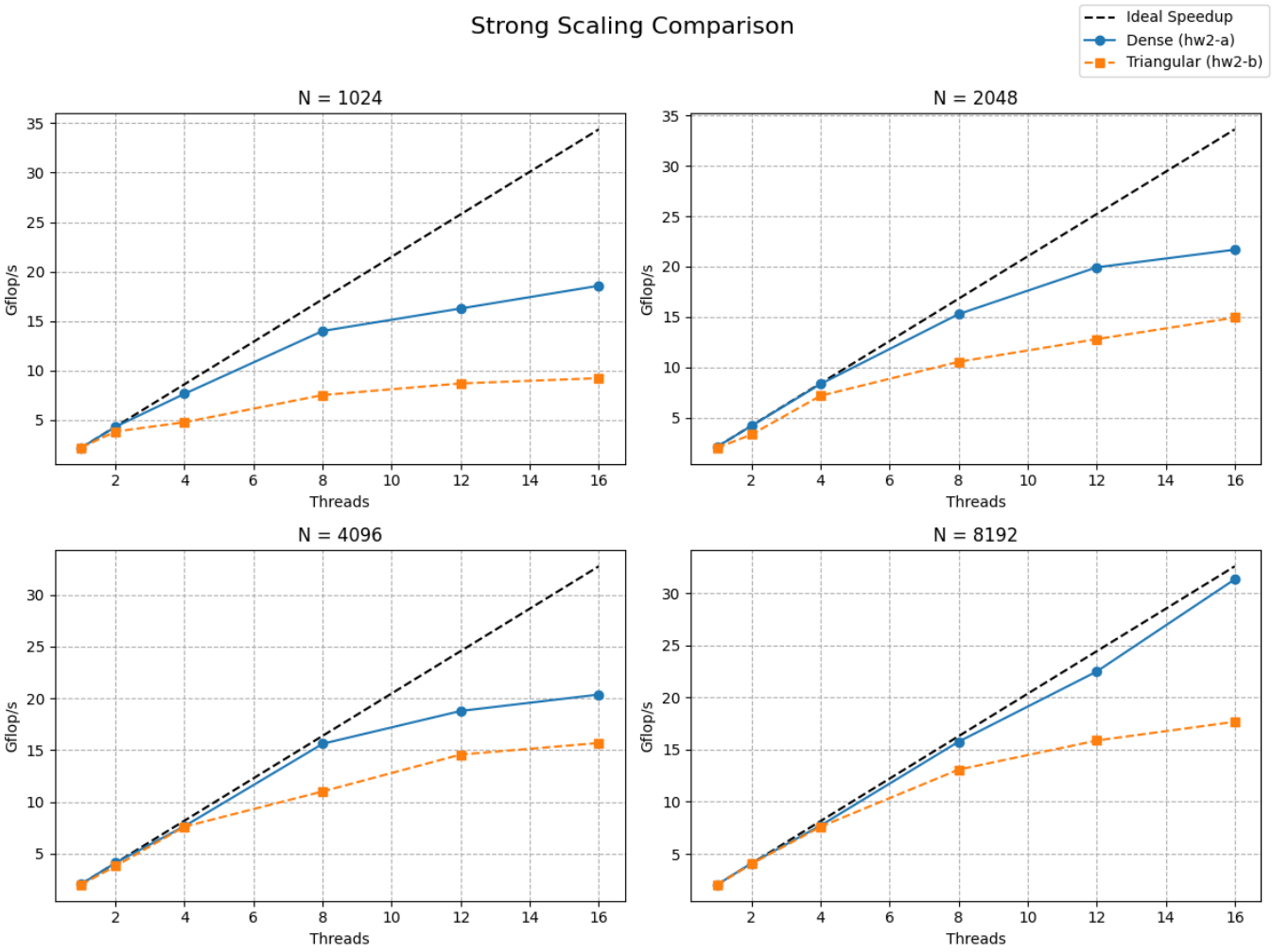
4. Comparison of Compiler Optimizations for hw2-b

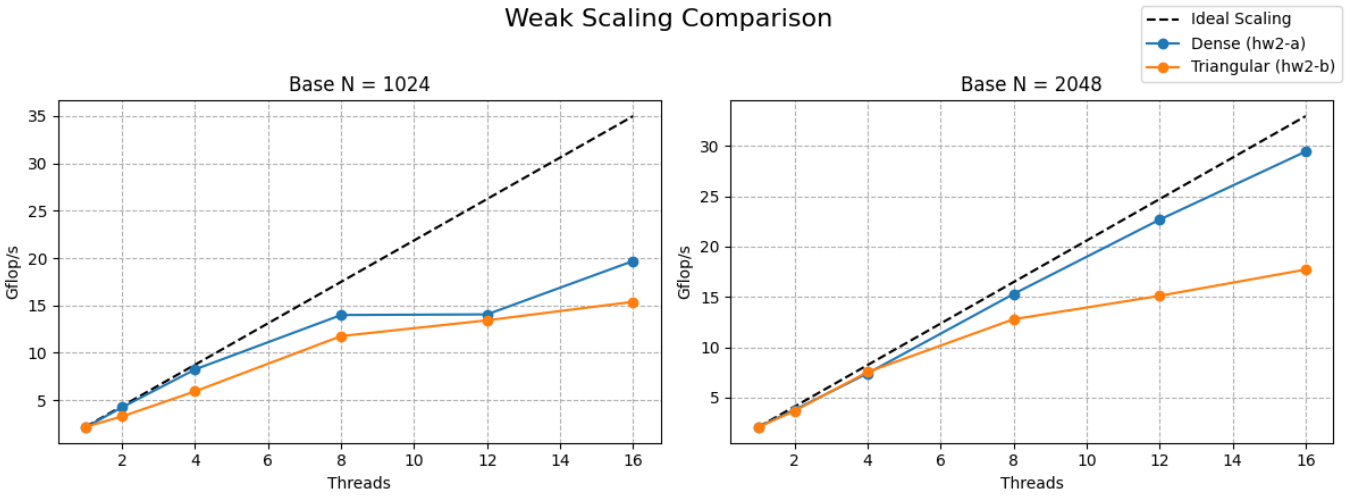
The following table compares the performance of the triangular matrix-vector multiplication (hw2-b) across the different compiler profiles. For each data point, the results from the best-performing schedule ('guided') were used. The table shows the peak performance in Gflop/s and, in parentheses, the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

Matrix Size	O3 Default	O2 Optimized	O3 Unrolled
1024x1024	8.97 (16T)	8.46 (16T)	9.21 (16T)
2048x2048	14.52 (16T)	13.94 (12T)	14.93 (16T)
4096x4096	17.97 (16T)	17.21 (12T)	15.68 (16T)
8192x8192	16.75 (16T)	15.27 (16T)	17.66 (16T)

5. Scaling Analysis (using 'O3_unrolled' profile)

The following graphs illustrate strong and weak scaling performance. These results were generated using the 'O3_unrolled' compilation profile. Both charts show side-by-side comparisons for all tested matrix size configurations.


Strong Scaling Comparison
Ideal Speedup
Dense (hw2-a)
Triangular (hw2-b)
N = 1024
N = 2048
N = 4096
N = 8192
Gflop/s
Threads


Weak Scaling Comparison
Ideal Scaling
Dense (hw2-a)
Triangular (hw2-b)
Base N = 1024
Base N = 2048
Gflop/s
Threads

6. 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, performance (Gflop/s) should increase linearly with the number of threads (the 'ideal speedup' line). The charts show this ideal case as a dashed line. The observed results typically show a curve that achieves good speedup initially but then flattens out at higher thread counts. This is explained by Amdahl's Law, where speedup is limited by sequential code portions and parallel overhead. This effect is more pronounced at smaller matrix sizes, where the amount of parallel work is not large enough to overcome the overhead of managing many threads.

Weak Scaling Insights

Weak scaling measures performance as both the problem size and the number of threads increase proportionally (i.e., work per thread is constant). Ideally, performance in Gflop/s should increase linearly with the number of threads. The charts demonstrate this by starting separate weak scaling experiments from different base matrix sizes. In practice, performance often falls short of this ideal. 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. By comparing the plots, we can see that experiments starting with a larger base N tend to achieve higher absolute Gflop/s, likely due to a better computation-to-communication ratio.

7. 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.