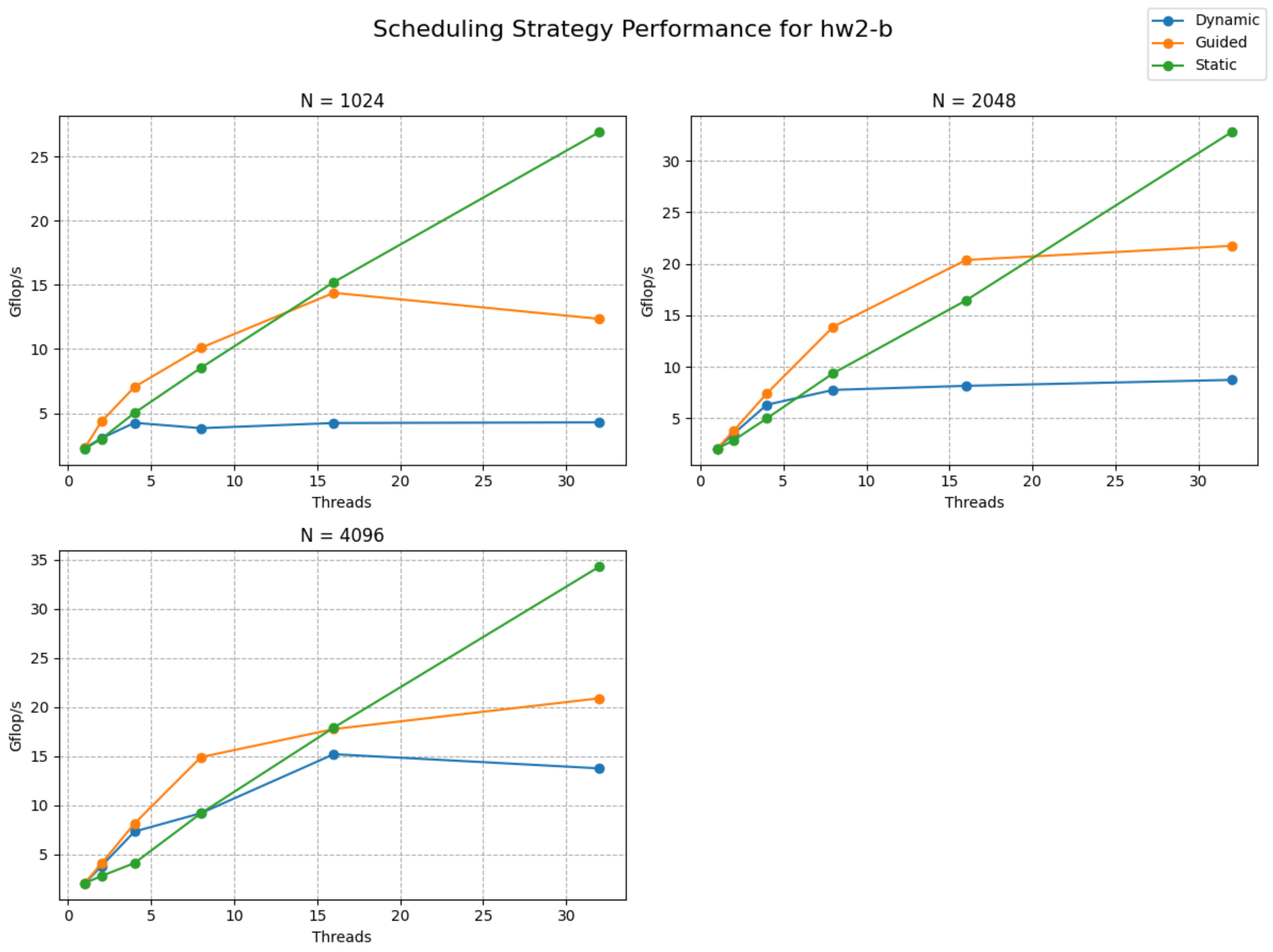
HW2 Performance Report

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 for the triangular matrix multiplication (hw2-b), which has an imbalanced workload. The workload is heaviest for the last rows of the matrix and lightest for the first rows.

Based on the cumulative performance across all tests, the 'static' scheduling strategy was determined to be the most effective. This result is somewhat unexpected for an imbalanced workload. A possible explanation is that for the matrix sizes and thread counts tested, the overhead associated with dynamic scheduling outweighed its load-balancing benefits. The simplicity and low overhead of the static scheduler, which pre-allocates an equal number of iterations to each thread, proved to be more efficient overall. This can happen if the imbalance is not severe enough to leave threads idle for long periods.

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.

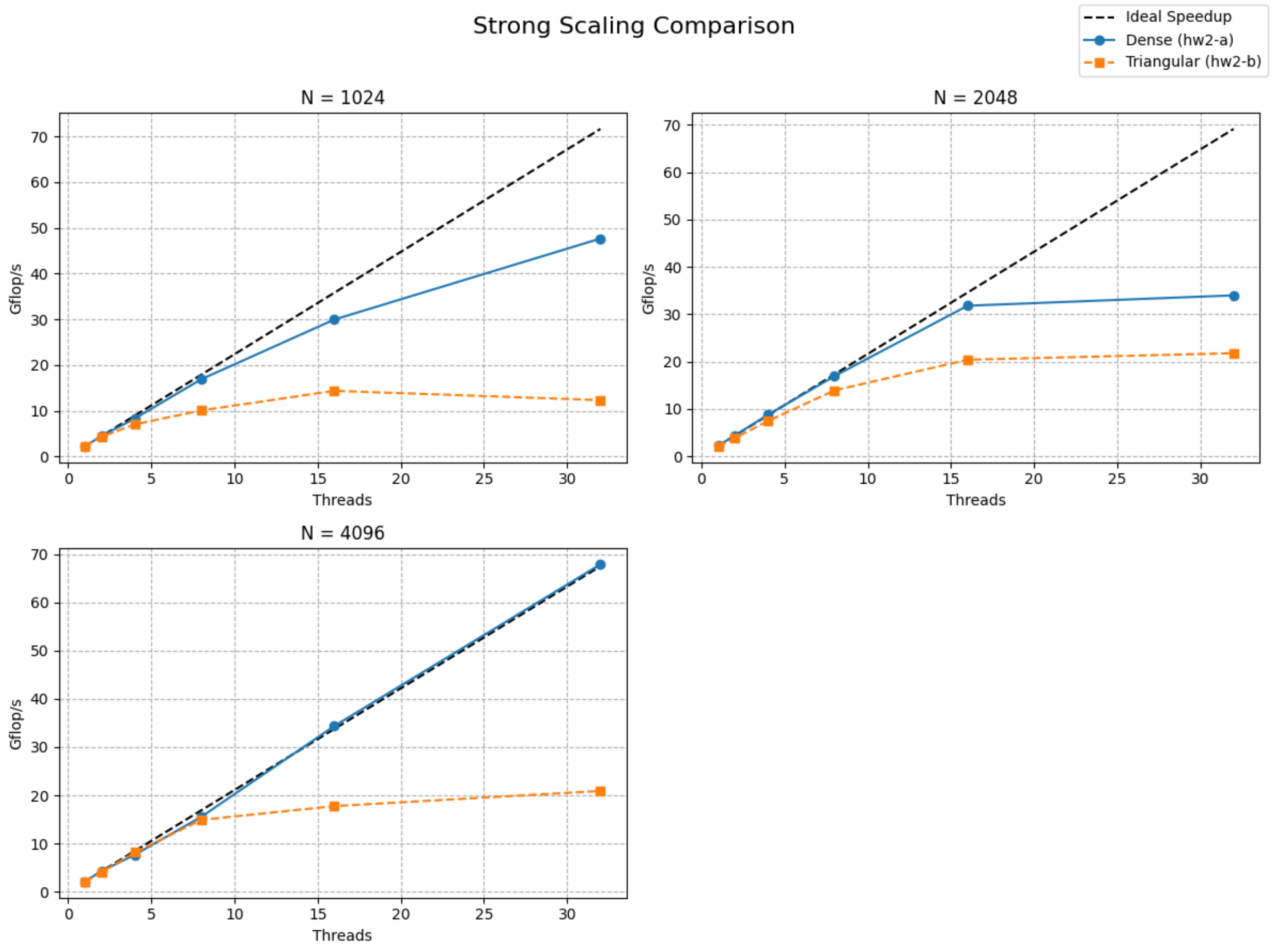
4. Comparison of Compiler Optimizations for hw2-b

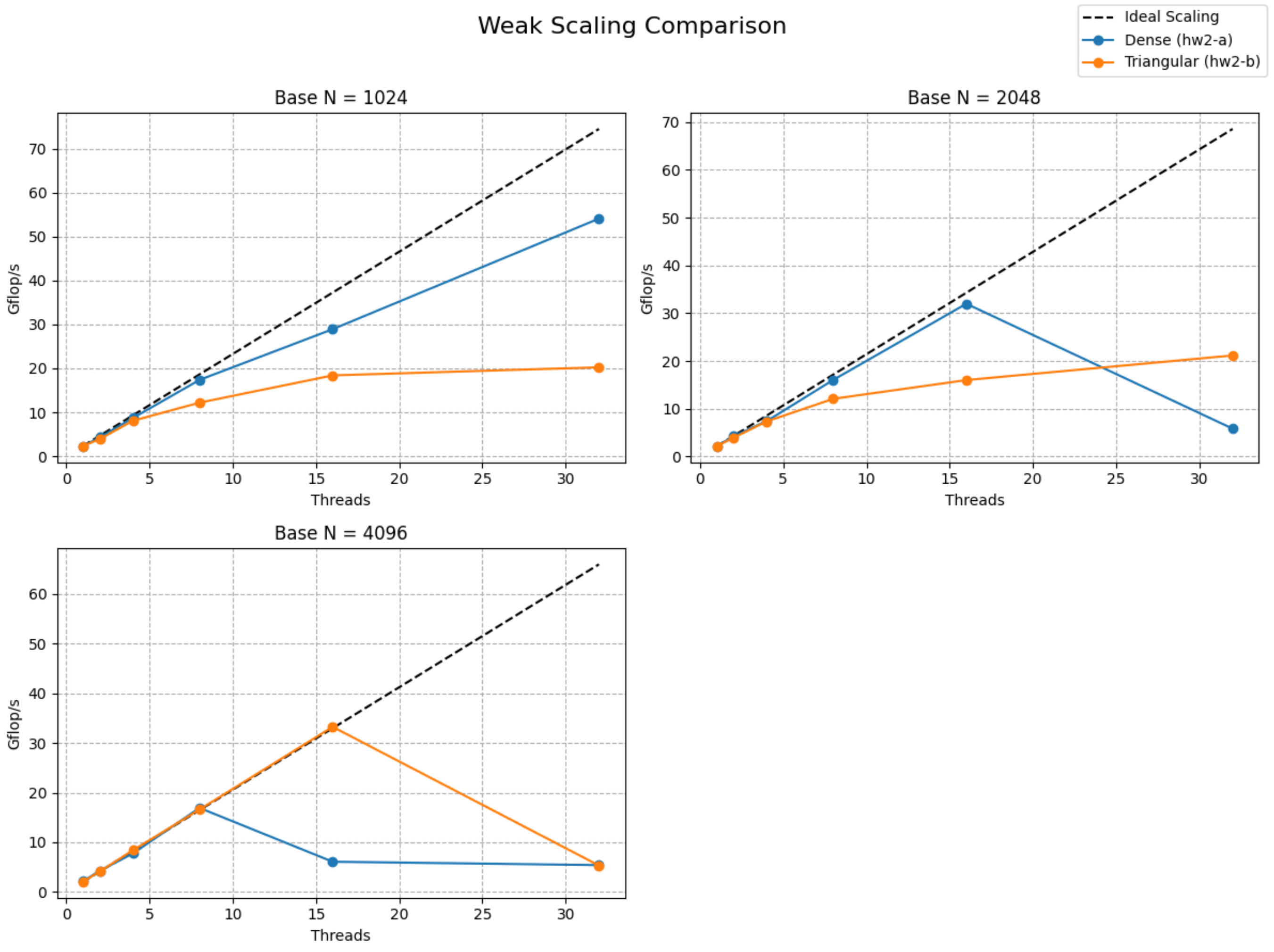
The following table compares the performance of the triangular matrix-vector multiplication (hw2-b) across the different compiler profiles. For each data point, the results from the best-performing schedule ('static') were used. The table shows the peak performance in Gflop/s and, in parentheses, the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

Matrix Size	O3 Default	O2 Optimized	O3 Unrolled
1024x1024	19.08 (32T)	24.99 (32T)	26.91 (32T)
2048x2048	32.53 (32T)	29.76 (32T)	32.78 (32T)
4096x4096	25.01 (32T)	34.11 (32T)	34.32 (32T)

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
Gflop/s
Threads
0
5
10
15
20
25
30
40
50
60
70


Weak Scaling Comparison
Ideal Scaling
Dense (hw2-a)
Triangular (hw2-b)
Base N = 1024
Base N = 2048
Base N = 4096
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.

The Impact of Thread Count vs. Available Cores

An important observation from manual testing is the significant performance difference between running the benchmark with a specific, limited number of threads (e.g., OMP_NUM_THREADS=16) versus allowing OpenMP to use all available cores on the system (e.g., 256 cores on Bridges-2). While it seems intuitive that more cores should equal more performance, this is often not the case for memory-bound problems like matrix-vector multiplication.

The primary bottleneck is not the CPU's computational power, but the speed at which it can fetch data from main memory (RAM). This is known as the memory bandwidth limit. When an excessive number of threads are launched simultaneously, they all contend for access to this limited memory bandwidth. This creates a "traffic jam" on the memory bus, causing most threads to spend their time waiting for data rather than performing calculations.

The benchmark script carefully tests a range of thread counts (1 to 32) to find the optimal point where the system's memory bandwidth can effectively service the active cores. This 'sweet spot' delivers the peak performance seen in the graphs. Launching threads beyond this point leads to diminishing returns and eventually a sharp drop in performance due to memory contention and parallel overhead, demonstrating a critical concept in high-performance computing: scaling is limited by the most constrained resource, which in this case is memory bandwidth.

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.