Scalability Analysis: Naive Implementation

Precision Impact & Benchmark vs OpenBLAS

Project AMSC

November 27, 2025

Performance Overview: Naive (Float)

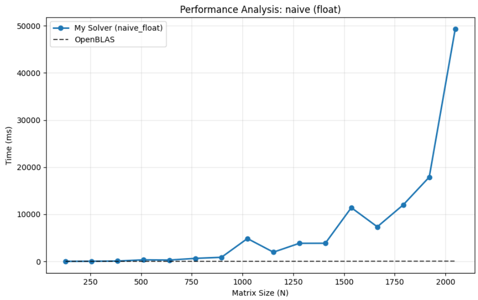


Figure: **Execution Time (ms)**

Exponential growth due to cache thrashing.

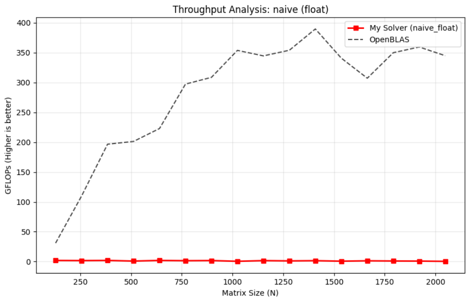


Figure: **Throughput (GFLOPS)**

Performance remains ¡ 2 GFLOPS regardless of size.

Quantitative Analysis: Float vs Double

Impact of Precision on Execution Time

Size (N)	Float	Double	Δ Overhead
128^3	2.59 ms	3.85 ms	$+48\%$
512^3	278 ms	377 ms	$+35\%$
1024^3	3.63 s	5.22 s	$+43\%$
2048^3	47.60 s	48.94 s	**Only +2.8%**

Insight: The "Latency Wall"

For smaller sizes, `double` is slower due to higher bandwidth usage (8 bytes vs 4 bytes).

However, at **N=2048**, the difference disappears.

Why? The CPU is entirely stalled by *latency* (waiting for RAM), not *bandwidth* (transfer speed). The penalty for jumping around in memory masks the data size difference.

Reference: OpenBLAS (Float) at $N = 2048$ takes only **0.058 s** ($\approx$ 820x faster).

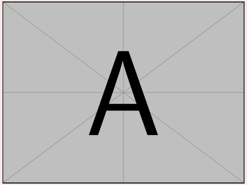
Why does Naive fail? (Bottleneck Analysis)

The standard (ijk) implementation exposes critical hardware limitations:

- **Cache Thrashing (Major):** Matrix B is accessed column-wise ($stride = N$). This breaks spatial locality, causing a massive amount of L1/L2 cache misses.
- **Precision Penalty:** Using `double` halves the effective SIMD width (2 doubles vs 4 floats per 128-bit vector) and consumes 2x cache lines, exacerbating the cache pressure.
- **Pipeline Stalls:** The CPU spends most cycles waiting for data fetch rather than computing.

Memory Access Pattern

`B[k][j]` with varying `k`



Jumping $N \times 4$ bytes every step leads to RAM fetch instead of Cache hit.