Scalability Analysis: Loop Unrolling

Instruction Level Optimization vs Memory Wall

Progetto AMSC

December 2, 2025

Performance Overview: Loop Unrolling (Float)

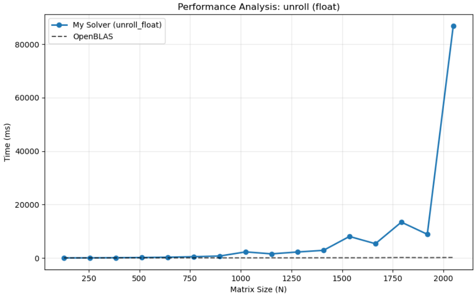

Figure: Execution Time (ms)

Curve remains cubic ($O(N^3)$). No visible improvement over Naive.

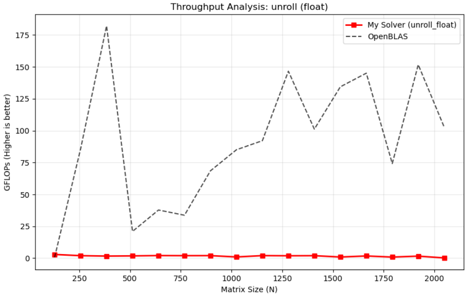

Figure: Throughput (GFLOPS)

Throughput is negligible (< 1 GFLOP), constrained by latency.

Quantitative Analysis: Float vs Double

Impact of Unrolling on Execution Time

Size (N)	Float	Double	vs Naive (Float)
128^3	2.15 ms	3.42 ms	-16% (Faster)
1024^3	4.79 s	4.31 s	+31% (Slower)
1536^3	13.44 s	19.02 s	High Instability
2048^3	49.68 s	53.70 s	**+4% (Slower)**

The "Optimization Theatre"

Loop Unrolling aims to reduce loop overhead (incrementing counters, branching).

Result: It failed to improve performance. At $N = 2048$, it is essentially identical to (or slightly slower than) the Naive implementation.

Reason: Saving a few CPU cycles on loop control is irrelevant when the CPU is stalled for hundreds of cycles waiting for Memory (Cache Misses).

Why did Unrolling fail?

Manually unrolling the loop (processing 4 or 8 elements explicitly) changes the instruction stream but not the data access pattern.

- **Same Memory Access Pattern:** We are still accessing matrix B by columns ($stride = N$). The L1 Cache Miss rate remains near 100% for large N.
- **Compiler Optimization:** Modern compilers (GCC/Clang) at `-O2` or `-O3` often auto-unroll loops better than manual attempts. Manual unrolling might confuse the compiler's heuristic.
- **Instruction Cache Pressure:** Unrolling increases the binary code size, potentially causing Instruction Cache misses.

Visual Concept

Loop Control (Saved):
`i++, if i < N jump`
(Cost: $\approx 1 - 2$ cycles)

Memory Fetch (Unchanged):
`Load B[k][j]`
(Cost: $\approx 100 - 300$ cycles)

Conclusion: The bottleneck is the Fetch, not the Loop Control.