Scalability Analysis: Tiling (Blocking)

Optimizing Cache Reuse & Memory Locality

Progetto AMSC

December 2, 2025

Performance Overview: Tiling (Float)

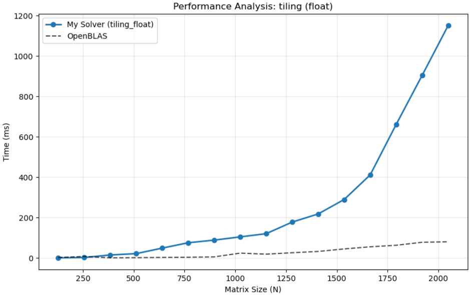


Figure: Execution Time (ms)

Consistent performance. No exponential explosion like Naive.

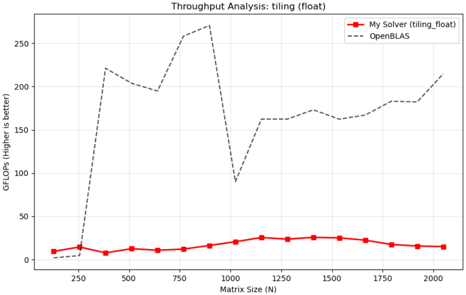


Figure: Throughput (GFLOPS)

Sustains $\approx 15 - 20$ *GFLOPS. Drops slightly at* $N = 2048$.

Quantitative Analysis: Float vs Double

Impact of Tiling on Execution Time

Size (N)	Float	Double	Δ Overhead
128^3	0.28 ms	0.31 ms	+10%
1024^3	94.75 ms	186.67 ms	+97%
1536^3	302.50 ms	1.03 s	+240%
2048^3	**1.32 s**	**3.05 s**	**+131% (≈ 2.3x)**

Insight: Cache Reuse Wins

Tiling drastically reduces Memory Access penalties compared to Naive ($47s \rightarrow 1.3s$).

Double Penalty: Since `double` takes 2x space, effective cache capacity is halved. The "Tile Size" (Block Size) effectively shrinks, leading to more cache misses compared to Float.

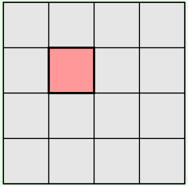
Comparison: Slower than optimized SIMD-2D ($0.73s$) due to higher loop overhead (6 nested loops).

Why Tiling works (The L1/L2 Sweet Spot)

Tiling (or Blocking) changes the order of operations to fit sub-matrices into the fast Cache memory.

- **Temporal Locality:** Instead of loading a row of A and scanning the *entire* matrix B (evicting data from cache), we load a small block of A and B, reuse them completely for calculations, and then move on.
- **Reduced Bandwidth Pressure:** Data is fetched from RAM once and used many times inside the CPU Cache.
- **The Trade-off:** It introduces complex loop logic (6 nested loops). Without efficient SIMD inside the inner loops, the instruction overhead limits the maximum GFLOPS.

Visual Concept



Processing Block-by-Block

Keeps active data in L1 Cache ($<$ 32KB).