Optimization Strategies: Loop Unrolling & Register Blocking

SIMD 1D vs SIMD 2D (Register Blocking)

Progetto AMSC

November 27, 2025

Part 1: SIMD 1D Unrolling Overview

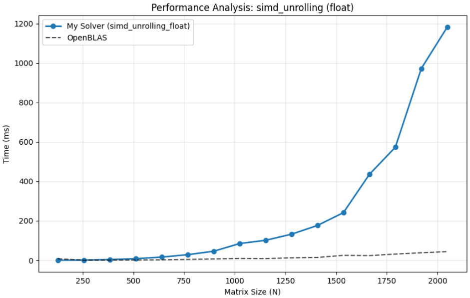

Figure: **Execution Time (1D)**

Linear scaling, but limited by memory bandwidth.

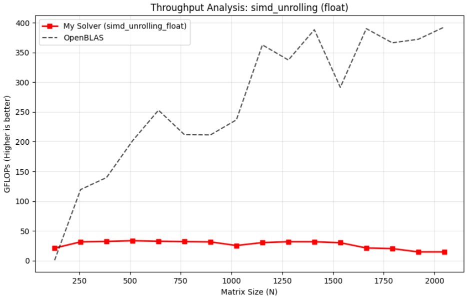

Figure: **Throughput (1D)**

Performance caps at $\approx$ 15 GFLOPS for large N.

1D Unrolling: Quantitative Analysis

1D Unrolling Performance (Float vs Double)

Size (N)	Float	Double	Gap
128^3	0.16 ms	0.30 ms	$+80\%$
1024^3	75.87 ms	160.20 ms	$+111\%$
1536^3	255.82 ms	1.07 s	$+317\%$
2048^3	**1.10 s**	**3.00 s**	**2.7x Slower**

Observation

1D Unrolling reduces loop overhead compared to basic SIMD, but it hits the **Memory Wall** early.

At $N = 2048$, the CPU spends more time fetching matrix B from RAM than computing, limiting the Float throughput to ≈ 15.6 GFLOPS.

Part 2: SIMD 2D Unrolling (Register Blocking)

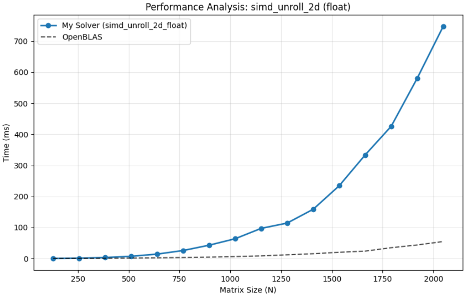

Figure: Execution Time (2D)

Significant speedup observed at large N.

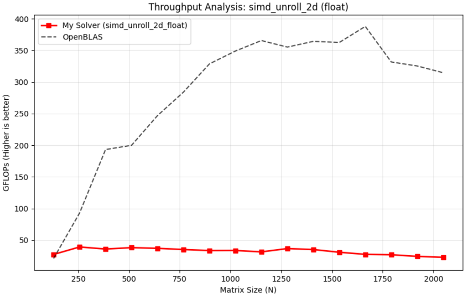

Figure: Throughput (2D)

Higher sustained throughput ($\approx$ 23.5 GFLOPS).

2D Unrolling: Quantitative Analysis

2D Unrolling Performance (Float vs Double)

Size (N)	**Float**	**Double**	**Gap**
128^3	0.13 ms	0.26 ms	+100%
1024^3	68.68 ms	137.71 ms	+100%
2048^3	**0.73 s**	**1.86 s**	**2.5x Slower**

Speedup vs 1D (Float): $1.10s \rightarrow 0.73s$ (**+50% Faster**)

The Impact of Register Blocking

By processing blocks of data, we improve **Arithmetic Intensity**.

We achieve $\approx$ 23.5 GFLOPS (Float) vs 15.6 GFLOPS (1D). The CPU is utilized much more efficiently because it waits less for memory.

Optimization Strategy: 1D Unrolling (Horizontal)

Mechanism: For each scalar element $A_{i,k}$, we load a "horizontal strip" of B and update a corresponding strip of C.

- **Loads:** 1 Scalar (A) + N Vectors (B).
- **Compute:** Update N accumulators in C (same row).

Pros & Cons:

+ **Easy to implement:** Only requires unrolling the innermost loop (j).
+ **Low Register Pressure:** Uses few registers (1 for A, 4-8 for C).
- **Bandwidth Waste:** For every new row of C, we must reload the entire matrix B. The CPU spends cycles waiting for loads.

1D Pattern

$$C[i][j \ldots j+4] += A[i][k] \times B[k][j \ldots j+4]$$

Optimization Strategy: 2D Register Blocking (Pro Level)

Mechanism: We process a block of rows (e.g., 2 rows of A) simultaneously. We reuse the *same* vector of B to update multiple rows of C.

- **Loads:** 2 Scalars (A_i, A_{i+1}) + **1 Vector** $\mathbf{(}B\mathbf{)}$.
- **Compute:** 2 FMA operations simultaneously:
  $C_i += A_i \times B$
  $C_{i+1} += A_{i+1} \times B$

Why is it faster?

+ **Halved Memory Access:** We fetch B once but use it for 2 rows of C. This drastically reduces L1 Cache traffic.
+ **Higher Arithmetic Intensity:** More math per byte loaded.

Trade-offs

! **High Register Pressure:** Requires many active registers. Excessive blocking (e.g., 4x4) can cause *Register Spilling*, degrading performance.

! **Complexity:** Requires handling edge cases (odd matrix sizes) with specific cleanup loops.