Improving the performance of Dense Matrix-Matrix Multiplication (GEMM)

Sai Kaushik S (2025CSZ8470)
Yosep Ro (2025ANZ8223)
Indian Institute of Technology Delhi

November 26, 2025

1 Introduction

The goal of this lab is to explore the performance impact of various hardware-level optimizations for Dense Matrix–Matrix Multiplication (GEMM) and to analyze their practical effects in terms of GFLOPs, speedup, and scalability. The provided baseline is a GEMM implementation written in Python, which we reimplemented and optimized in C++ to achieve high-performance execution. The original Matrix Multiplication (MM) approach is illustrated in Figure 1.

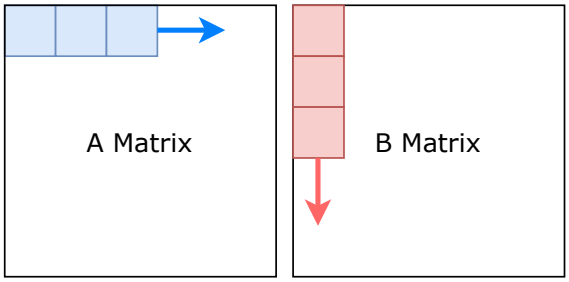

Figure 1: Baseline Matrix Multiplication

To enhance overall system performance, we applied six key optimizations that target different aspects of the processor and memory hierarchy. The complete source code, simulation scripts, and data used for generating the plots in this report are publicly available on GitHub at https://github.com/sai-kaushik-s/Enhanced-GEMM.

The host system specifications are as follows:

Table 1: System configurations.

Component	Specification
CPU	AMD Ryzen 7 5800H (8C/16T)
RAM	16 GB DDR4-3200 MHz (Dual Channel)
ISA	x86 64-bit
OS	Ubuntu Desktop 24.04 LTS

2 Architectural Design

This section presents seven architecture-level optimization techniques applied to the baseline GEMM implementation. Each subsection describes the technique, the architectural reasoning behind it, and its expected performance impact.

2.1 Row and Column Major Storage of the Matrics (Spacial Locality)

Since the multiplier matrix (the B Matrix) is accessed column-wise in all the multiplication steps, we have the matrix stored in a column major order to improve access to the required contiguous elements, enhancing the locality of the data when fetching it to the caches.

2.2 Multi-level Tiling of Matrix (Cache Blocking)

We employed a **tiling (blocking)** technique that divides the matrices into multiple smaller tiles. The tile size was carefully chosen to fit within the cache capacity, ensuring that each block of data could be reused multiple times before being evicted from the cache. Furthermore, we implemented a **two-level tiling technique** aligned with the L1 and L2 cache levels so that the innermost tiles fit in the L1 cache while the outer tiles fit in the L2 cache. This hierarchical blocking reduces memory traffic and improves cache reuse, leading to a substantial decrease in cache miss rates. The graphical illustration of two-level tiling is described in Figure 2.

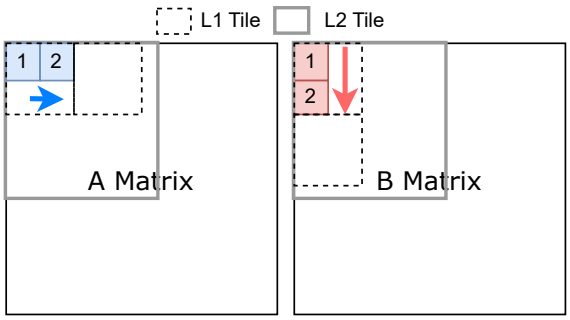

Figure 2: Two-level cache-based tiling

We maximized the tile size to fit within each cache level (L1 and L2), ensuring that the working set of the computation remains in the cache. The tile size is

determined according to the following constraint:

$$(\text{Tile size of A}) + (\text{Tile size of B}) \leq \alpha \cdot \frac{\text{Cache size}}{\text{sizeof(double)}}$$

where $\alpha \in [0.6, 0.8]$ is a safety factor that accounts for code and metadata overhead.

At the beginning of each iteration, one tile of matrix A and one tile of matrix B are loaded from memory into the L2 cache. Within that region, the algorithm iterates over smaller L1-sized sub-tiles to perform block multiplications. Each L1 tile is processed by the vectorized micro-kernel, which performs fused multiply-add (FMA) operations across packed data. After completing all sub-tiles for a given L2 tile, the resulting partial sums are accumulated into the corresponding output block in C and written back to memory once. The hierarchical loop structure can be described as Algorithm 1.

Algorithm 1 Multi-level Tiling Matrix Multiplication

1: **for** each L2-sized tile of A and B **do**
2: Load $A_{\text{L2-tile}}$ and $B_{\text{L2-tile}}$ into L2 cache
3: **for** each L1-sized subtile inside the current L2 block **do**
4: Perform micro-kernel multiplication on $A_{\text{L1-subtile}}$ and $B_{\text{L1-subtile}}$
5: Accumulate results into C_{subtile}
6: **end for**
7: Write back C_{subtile} to main memory
8: **end for**

2.3 AVX2 / AVX-512 Vectorization

To further increase the computational throughput of the inner kernel, we explicitly vectorize the innermost loop using AVX2 and AVX-512 intrinsics. While the naive implementation performs scalar multiply–accumulate operations on a single element of C at a time, the optimized kernel operates on multiple columns of C in parallel by exploiting wide SIMD registers and fused multiply-add (FMA) instructions.

For the AVX2 case, we use 256-bit vectors (`_m256d`), which can hold four double-precision elements. For AVX-512, we follow the same structure but widen the kernel to 512-bit vectors (`_m512d`), which hold eight double-precision elements. By mapping the inner-most computation onto AVX2 or AVX-512 FMA instructions, the kernel can perform 4–8 floating-point operations per instruction and keep the floating-point pipelines saturated. Combined with loop unrolling across output columns, the vectorized micro-kernel substantially increases the achieved per-core GFLOPs and raises the measured IPC compared to the scalar baseline. Figure 3 illustrates how a single AVX vector loads multiple consecutive elements from the packed A and B matrices into SIMD registers, enabling wide FMA operations within the inner kernel.

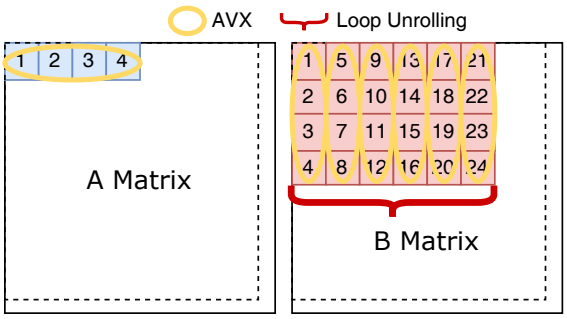

Figure 3: Illustration of the inner micro-kernel. A single SIMD vector loads four consecutive elements from the A panel (left), while loop unrolling applies multiple SIMD operations across several consecutive columns of the B panel (right).

```
#pragma omp parallel for collapse(2) schedule(static)
for (size_t i = 0; i < aRows; i += MC) {
    for (size_t j = 0; j < bCols; j += NC) {
        // Each thread computes one output tile
        compute_tile(i, j);
    }
}
```

Figure 4: OpenMP parallelization of the outer tile loops.

2.4 Loop Unrolling

To further increase instruction-level parallelism (ILP) and reduce loop-control overhead, we apply manual loop unrolling in the innermost kernel over the output columns. Instead of updating a single column of C per iteration, the unrolled kernel updates six adjacent columns in parallel, each tracked by its own accumulator register. As shown in Figure 3, loop unrolling enables the micro-kernel to process multiple contiguous columns of the B matrix simultaneously, allowing the hardware scheduler to overlap FMAs, loads, and prefetch operations more effectively.

2.5 Cache Prefetching

To hide memory latency in the innermost micro-kernel, we issue prefetch instructions that bring future elements of the packed A and B panels into the L1 cache before they are consumed. By prefetching data at a fixed offset ahead of the current iteration, the kernel overlaps memory access with computation and reduces long-latency cache misses.

2.6 OpenMP Parallelization

We parallelize the outer-most tile loops using OpenMP with static scheduling. Each thread receives an independent (i, j) tile of the output matrix, avoiding false sharing and minimizing synchronization overhead. Because each tile fits within the target cache level, the per-thread working set remains cache-resident. The parallelization strategy is shown in Figure 4.

N	Kernel	L1	L2	L3
1024	Baseline	16.84%	30.70%	0.98%
	Optimized	8.45%	0.81%	37.84%
2048	Baseline	17.84%	44.56%	53.59%
	Optimized	8.43%	0.79%	41.27%
4096	Baseline	22.80%	77.92%	98.72%
	Optimized	8.73%	0.86%	50.69%

Table 2: Cache miss rates (%) for baseline and optimized GEMM kernels across different matrix sizes, measured using `perf stat` (`mem_load_retired{hit,miss}`).

N	Kernel	Instr	IPC	GFLOPs
1024	Baseline	1.93×10^{9}	2.12	14.9
	Optimized	1.55×10^{9}	1.42	82.2
2048	Baseline	1.42×10^{10}	1.20	8.00
	Optimized	1.05×10^{10}	1.00	138.4
4096	Baseline	1.08×10^{11}	0.84	5.60
	Optimized	7.82×10^{10}	1.62	201.1

Table 3: Retired instructions, IPC, and achieved GFLOPs for baseline and optimized GEMM kernels across matrix sizes, measured using `perf stat`.

2.7 NUMA-aware Memory Placement

On a dual-socket NUMA system, remote memory accesses incur significantly higher latency. To minimize cross-socket traffic, we ensure that thread placement and memory allocation occur on the same NUMA node. Specifically, the program allocates its working buffers after the OpenMP threads are pinned, enabling a first-touch allocation policy that aligns memory pages with the thread's NUMA domain.

3 Result

3.1 Cache Miss Analysis

Table 2 reports the L1, L2, and L3 cache miss rates for both the baseline and optimized kernels across problem sizes $N = \{1024, 2048, 4096\}$. The baseline suffers from consistently high L2 and L3 miss rates, which grow rapidly with matrix size due to the poor spatial locality of the naive triple-nested loops. For instance, at $N = 4096$, the L2 and L3 miss rates reach 77.92% and 98.72%, indicating that most accesses fall through all cache levels and incur expensive main-memory fetches.

In contrast, the optimized kernel dramatically reduces L2 misses to below 1% for all matrix sizes, demonstrating the effectiveness of multi-level tiling, data packing, and software prefetching. The L1 miss rate is also reduced by approximately half compared to the baseline (e.g., 22.80% $\rightarrow$ 8.73% for $N = 4096$), owing to the use of unit-stride accesses within the packed panels and the predictable access pattern of the micro-kernel.

We observe that the optimized kernel exhibits higher L3 miss rates than L2 misses, but this is expected: once the packed panels are loaded into L2, the kernel repeatedly reuses them within the tile, causing most demand loads to hit in L2 and bypass the LLC entirely. Therefore, the elevated L3 miss rate is not a performance bottleneck; rather, it reflects that the working set is effectively retained in the private L1 and L2 caches during the inner kernel. Overall, these results confirm that cache blocking and prefetching drastically reduce cache stall penalties and enable sustained high throughput.

3.2 IPC, Instruction Count, and GFLOPs Analysis

To complement the cache miss analysis in Section 3.1, we further analyze the retired instruction count, instructions-per-cycle (IPC), and achieved floating-point throughput (GFLOPs) for both the baseline and optimized GEMM kernels. Table 3 summarizes the results for three matrix sizes.

Across all problem sizes, the optimized kernel retires a comparable or even smaller number of instructions than the baseline, yet achieves one to two orders of magnitude higher floating-point throughput. This behavior reflects the effect of architectural optimizations such as AVX2/AVX-512 vectorization, manual loop unrolling, and multi-level tiling, which allow each retired instruction to perform substantially more useful work.

For example, at $N = 4096$, the baseline kernel achieves only 5.6 GFLOPs, despite retiring 1.08×10^{11} instructions. In contrast, the optimized kernel retires fewer instructions (7.82×10^{10}) while achieving 201.1 GFLOPs, corresponding to over a 36$\times$ speedup. Although the IPC of the optimized kernel does not necessarily increase proportionally, its throughput is significantly higher because each vector FMA instruction operates on 4–8 double-precision values simultaneously.

These results demonstrate that the optimized kernel shifts execution from a scalar, memory-bound regime toward a throughput-oriented, vectorized compute-bound regime, confirming the effectiveness of the architectural techniques described in Section 2.

3.3 Performance Analysis

In this section, we evaluate the end-to-end performance impact of the architectural optimizations described in Section 2. We analyze the runtime scaling behavior with respect to thread count, and quantify the resulting speedup of the optimized kernel over the baseline implementation. The results are obtained using the standardized `scorer.py` script provided with the assignment framework, which executes each kernel three times and reports the median runtime.

Figure 5 shows the runtime of both kernels for $N = 1024$ while varying the thread count from 1 to 128. The baseline kernel exhibits almost no performance variation across threads, reflecting the fact

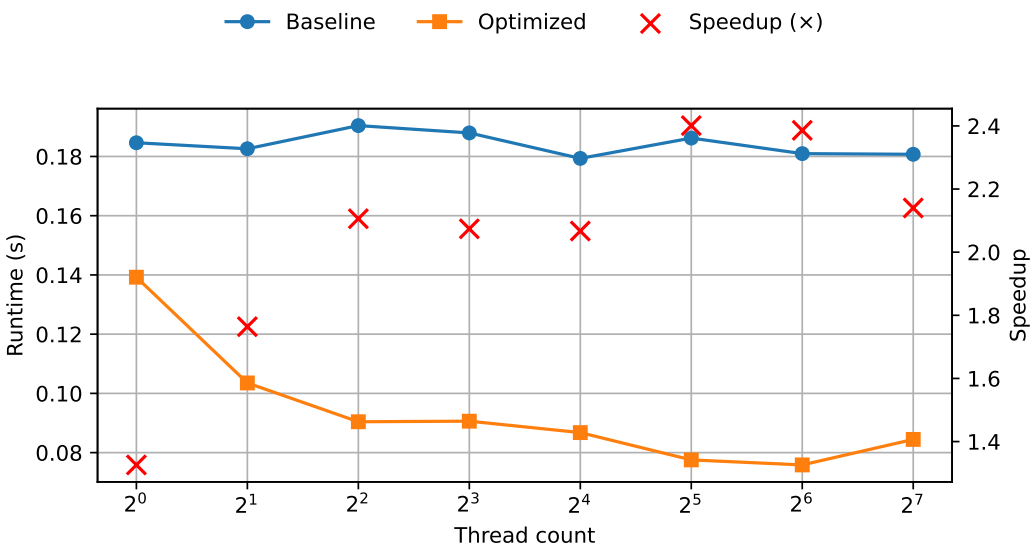

Figure 5: Runtime and speedup of the baseline and optimized GEMM kernels. The left plot shows runtime and speedup as a function of the thread count, and the right plot shows runtime and speedup across different matrix sizes.

that its naive memory access pattern limits the benefit of parallel execution. In contrast, the optimized kernel shows substantially reduced runtimes as the number of threads increases, demonstrating effective use of OpenMP parallelism and improved cache locality within each thread.

Additionally, the red "X" markers in Figure 5 represent the observed speedup Speedup $= T_{\text{baseline}}/T_{\text{optimized}}$ at each thread count. Speedups range from 1.33× at one thread to peaks above 2.4× beyond 32 threads, indicating that the optimized kernel benefits not only from vectorization and tiling, but also from improved inter-thread load balance and reduced memory stall cycles.

4 Conclusion

Our results show that the dominant architectural factors behind the performance improvements are (1) increased computational parallelism through SIMD and multithreading, and (2) improved memory-system efficiency through parallelism-aware tiling and prefetching.

First, the combination of AVX2/AVX-512 vectorization and OpenMP parallelization provides the largest contribution to overall speedup. By widening the inner micro-kernel to operate on 4–8 double-precision elements per instruction, the optimized kernel substantially increases the effective floating-point throughput. This leads to a significant improvement in achieved GFLOPs compared to the scalar baseline, especially for large matrix sizes where the available parallelism is abundant. OpenMP further amplifies this effect by distributing independent tile computations across multiple cores, allowing the overall GFLOP rate to scale with thread count. These results confirm that SIMD width and core count fundamentally determine the upper bound of compute throughput.

Second, memory-system behavior plays a crucial role. The baseline kernel suffers from high L2 and L3 miss rates due to repeated accesses to non-contiguous matrix regions. In contrast, our multi-level tiling scheme was deliberately designed with parallelism in mind: each tile is sized to fit the L1/L2 cache capacity, enabling multiple threads to operate on cache-resident blocks without interference. Additionally, the loop-unrolling strategy was aligned with the vector width so that the inner kernel issues a balanced mix of loads and FMAs, maximizing SIMD lane utilization. Software prefetching further overlaps memory latency with computation. Together, these mechanisms reduce L2 miss rates to below 1% and greatly diminish memory stall cycles, allowing the compute units to remain fully utilized and sustaining high GFLOPs.