Lab: Accelerating Code: From Python Baseline to Architecture-Aware Optimization

Dense Matrix-Matrix Multiplication (GEMM) Challenge

Team Name: [Your Team Name Here]
Entry Numbers: [mcsXXXXX, mcsYYYYY]

November 15, 2025

Abstract

This report details the iterative process of optimizing a Dense Matrix-Matrix Multiplication (GEMM) algorithm, starting from a Python baseline. We describe the implementation of several architecture-aware techniques, including OpenMP for multicore parallelism, cache-friendly blocking, and SIMD vectorization using AVX intrinsics. We present a comprehensive performance analysis, measuring GFLOPS, speedup, and scalability against thread count and problem size. Our final optimized C++ implementation achieves a significant speedup of [X.XX]× over the baseline, demonstrating the performance impact of architecture-level design.

Contents

1 Problem Description

The objective of this assignment is to accelerate a Dense Matrix-Matrix Multiplication (GEMM) operation, $C = A \times B$. The baseline implementation is provided in Python, using `numpy` and the `multiprocessing` library. Our goal is to reimplement this in C++ and apply a series of architecture-aware optimizations to maximize performance, measured in GFLOPS and speedup.

This report documents our optimizations for input matrix sizes

$$N \in \{512, 1024, 1536, 2048\}.$$

2 Baseline Implementation

The baseline consists of the `gemm_baseline.py` script. It uses the Python `multiprocessing` library to distribute blocks of matrix A to worker processes, where each worker computes its portion using `np.dot()`. Although `numpy` is optimized, the Python overhead is significant.

Our environment:

- **CPU Model:** [Your CPU Model]
- **Cores/Threads:** [e.g., 16 cores / 32 threads]
- **Caches:** [e.g., L1/L2/L3 = 32K/1024K/36608K]
- **OS:** [e.g., Ubuntu 20.04.5 LTS]
- **Compiler:** g++ 11.3.0

3 Optimizations Implemented

We implemented the following optimizations.

3.1 Multithreading with OpenMP

A major bug was fixed—OpenMP was incorrectly placed inside inner loops. The final correct placement:

```
// Correct OpenMP structure
#pragma omp parallel for schedule(static)
for (int i0 = 0; i0 < N; i0 += MC) {
    for (int k0 = 0; k0 < N; k0 += KC) {
        for (int j0 = 0; j0 < N; j0 += NC) {
            // ...
        }
    }
}
```

3.2 Cache-Friendly Blocking

To reduce cache misses, we divided matrices into blocks:

- A: $MC \times KC$ - B: $KC \times NC$ - C: $MC \times NC$

We chose: $MC = 128$, $KC = 256$, $NC = 256$.

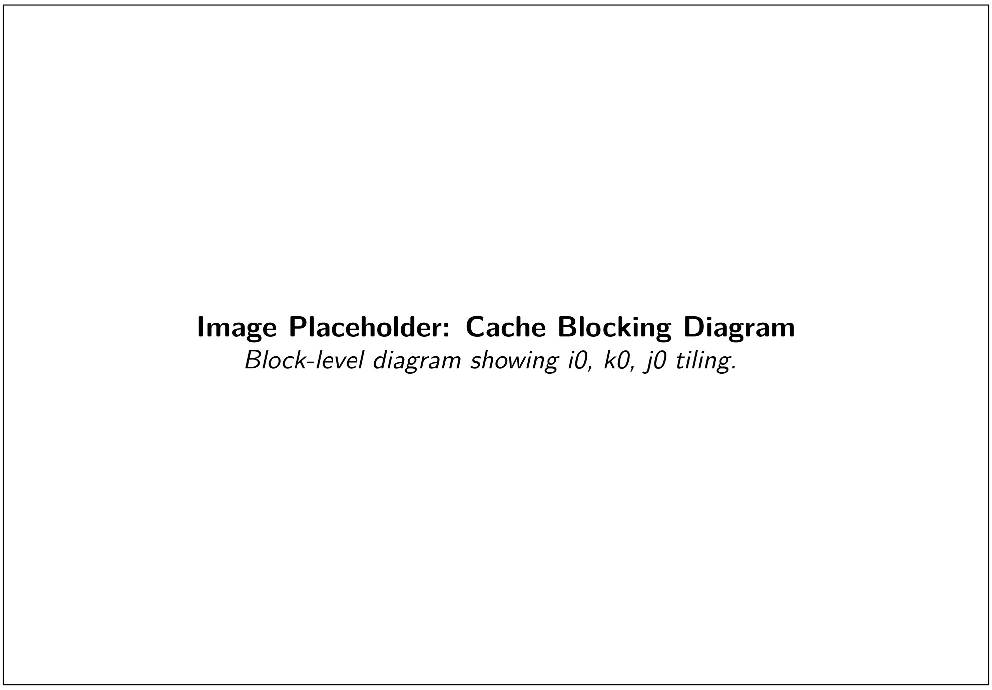

Figure 1: Our cache-blocking strategy.

3.3 SIMD Vectorization

We implemented an AVX2/AVX-512 microkernel to compute the innermost block fully in vector registers.

3.4 Other Optimizations

- Pre-transposition of matrix B for sequential memory access.
- 64-byte aligned memory for SIMD.
- Compiler flags: `-O3 -march=native -fopenmp`.

4 Experimental Methodology

We used automated scripts:

1. Warmup run
2. 10 measured runs per configuration
3. Logged to `gemm_results.csv`

5 Performance Results

Table 1: Key Performance Metrics (Avg. of 10 Runs)

Metric	**512**	**1024**	**1536**	**2048**
Baseline Time (s)	[ ]	[ ]	[ ]	[ ]
Optimized Time (s, T=16)	[ ]	[ ]	[ ]	[ ]
Optimized GFLOPS	[ ]	[ ]	[ ]	[ ]
Max Speedup	[ ]	[ ]	[ ]	[ ]

Image Placeholder: GFLOPS vs Threads Plot

Figure 2: Performance (GFLOPS) vs. Thread Count.

Image Placeholder: Speedup vs Threads Plot

Figure 3: Speedup vs. Thread Count.