Assignment Report #1
Matrix–Matrix Multiplication: Performance and Scaling

Student Name: Adilet Akimshe (ID: 202194337)
Course: PHYS 421 Parallel Computing

Submitted: September 7, 2025

Software packages used

GCC 13.3.0	C++ compilation (Linux/WSL)
OpenBLAS (Ubuntu 24.04 libopenblas-dev)	BLAS library (DGEMM)
Python 3.12 + NumPy 1.x, pandas, matplotlib	Python benchmarks and plotting

AI tools used

ChatGPT-5	Code review and organization, latex editing

Abstract

This report compares six approaches to dense $n \times n$ matrix multiplication in double precision: three C++ variants (naive triple loop, transpose-aided, and blocked), the BLAS DGEMM routine (OpenBLAS), and two Python versions (explicit loops and NumPy). Benchmarks were performed on Ubuntu (WSL) in a controlled single-thread setting. The blocked kernel improves cache locality substantially over naive methods but still falls short of DGEMM. NumPy mirrors DGEMM performance, since it delegates the computation to BLAS. Overall timings follow the expected $O(n^3)$ scaling, with DGEMM sustaining about 30–37 GFLOPS on this system.

1 Introduction and Problem Statement

We study performance characteristics of dense matrix–matrix multiplication $C = AB$ on a single CPU core. The operation costs asymptotically $2n^3 - n^2$ floating-point operations (FLOPs). Despite identical arithmetic counts, layout and access patterns determine whether runtime is compute-bound or memory-bound. We compare:

1. **Method 1**: textbook triple loop (i–j–k) in C++,
2. **Method 2**: C++ triple loop with precomputed B^T to improve locality,
3. **Method 3**: blocked (tiled) C++ kernel with block size n_b,
4. **Method 4**: BLAS DGEMM (OpenBLAS),
5. **Method 5**: pure-Python loops,

6. **Method 6**: NumPy (`A @ B`).

We measure *seconds per multiply* and compute GFLOPS as GFLOPS $= (2n^3 - n^2)/t/10^9$. The total number of floating-point operations (FLOPs) required for multiplying two $n \times n$ matrices is

$$\text{FLOPs} \;=\; n^3 \text{ (multiplications)} \;+\; n^2(n-1) \text{ (additions)} \;\approx\; 2n^3 \quad \text{for large } n. \tag{1}$$

Given a measured runtime t (in seconds) for one multiplication, the achieved throughput in double precision is computed as

$$\text{GFLOPS} \;=\; \frac{\text{FLOPs}}{t \times 10^9} \;\approx\; \frac{2n^3}{t \times 10^9}. \tag{2}$$

This expression reports the performance in billions of floating-point operations per second.

2 Hardware and Software Environment

Experiments were performed under Ubuntu 24.04 running inside WSL. To ensure a fair single-thread comparison across all methods, BLAS multithreading was explicitly disabled by sourcing a small shell script `env_single_thread.sh` containing:

```
#!/bin/bash
export OMP_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
```

Listing 1: Single-thread environment setup (env_single_thread.sh).

This script was invoked before each benchmark run via

```
source ./env_single_thread.sh
```

Compiler flags for C++: `-O3 -march=native -DNDEBUG`. Python environment used a virtualenv with NumPy, pandas, and matplotlib.

3 Methods and Implementation

3.1 C++ implementations (Methods 1–4)

Method 1 uses row-major arrays; $B[k, j]$ is accessed with stride n in the innermost loop, hurting cache locality. **Method 2** precomputes B^T and multiplies rows of A with rows of B^T so both operands in the inner loop are contiguous. **Method 3** partitions matrices into $n_b \times n_b$ tiles and accumulates contributions into the corresponding C tile. This improves temporal locality by reusing A and B tile data while the C tile remains resident in cache. The nominal live working set per micro-phase is roughly $3n_b^2$ doubles (A, B, C). **Method 4** calls `cblas_dgemm` from OpenBLAS.

3.2 Python implementations (Methods 5–6)

Method 5 is a direct translation of the textbook triple loop into Python, serving as a reference for interpreter overhead. **Method 6** uses NumPy's `@` operator (BLAS-backed).

3.3 Build and Run Instructions

C++:

```
make              # builds mm_bench, linked with -lopenblas
./mm_bench --m 2000 --step 100 --nb 128 --base_reps 8 \
--csv results/cpp_bench.csv
```

Python:

```
python3 -m venv .venv
source .venv/bin/activate
pip install numpy pandas matplotlib
python3 scripts/mm_bench.py --m5 300 --step 20 --base_reps5 60 \
--base_reps6 8 --csv results/python_bench.csv
python3 scripts/plot_results.py
```

Block size. A practical n_b heuristic targets L2 capacity: ensure roughly $3n_b^2 \cdot 8$ bytes (A, B, C tiles) fit comfortably below L2. So we have 3 matrices each with a block of $3n_b^2$ of 8 bytes for double precision. For L1 it would be fast but the overhead is huge because we would need to move small matrices many times in the memory. For L3, the memory is shared and slow, so the whole point of fast cache is lost. For L2 sizes of 256–1024 KB, candidates like 96–224 often perform well; the final choice is empirical.

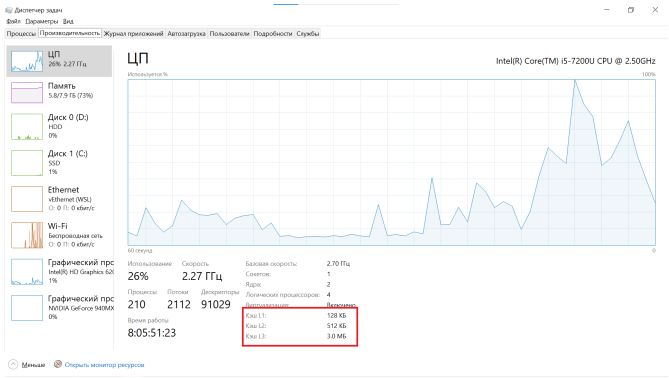

Figure 1: CPU cache hierarchy on the test machine (Windows system info). Block size $n_b = 128$ was chosen to fit $3n_b^2 \cdot 8 \approx 0.39$ MB comfortably within the 512 KB L2 cache per core

4 Results

Figures 2–4 summarize the measurements. Matrices are filled with i.i.d. uniform random FP64 entries in $[0, 1]$. Timing excludes random generation and transpose. The experiment with python took too long even for small sizes, so I decided that it was clear enough to not continue further with it.

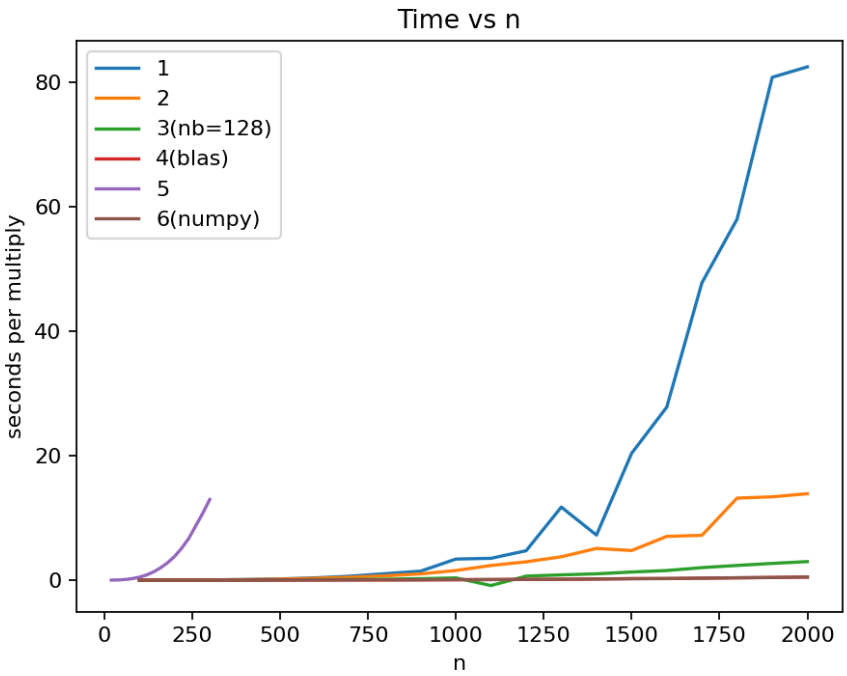

Figure 2: Time per multiplication vs matrix size n.

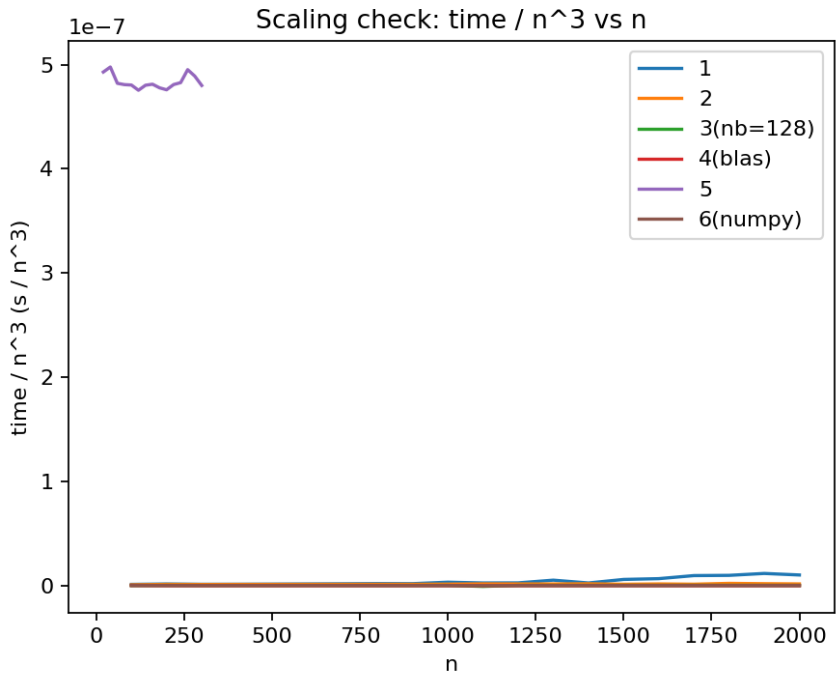

Figure 3: Scaling check: t/n^3 vs n (approximately flat indicates $O(n^3)$).

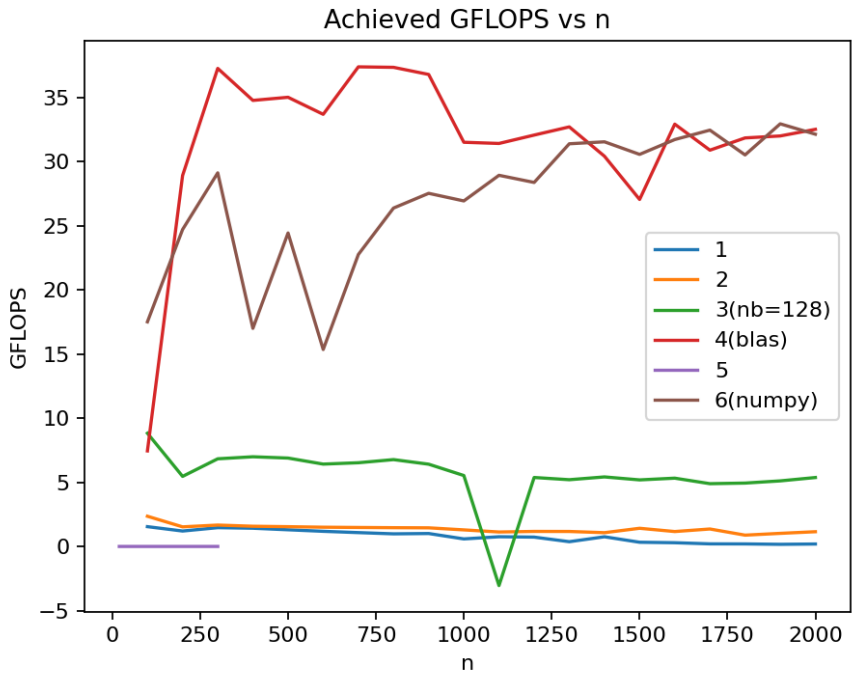

Figure 4: Achieved GFLOPS vs n.

Table 1: Sample timings and throughput for selected matrix sizes.

Language	Method	n	Time (s)	GFLOPS
C++	1 (naive)	200	0.00947	1.69
	2 (transpose)	200	0.00804	1.99
	3 (blocked,128)	200	0.00193	8.28
	4 (BLAS)	200	0.00063	25.45
C++	1 (naive)	1000	2.689	0.74
	2 (transpose)	1000	1.308	1.53
	3 (blocked,128)	1000	0.268	7.45
	4 (BLAS)	1000	0.04799	41.66
C++	1 (naive)	2000	81.22	0.20
	2 (transpose)	2000	13.54	1.18
	3 (blocked,128)	2000	3.356	4.77
	4 (BLAS)	2000	0.5006	31.96
Python	5 (loops)	200	3.807	0.004
	6 (NumPy)	200	0.000646	24.70
Python	5 (loops)	300	12.96	0.004
	6 (NumPy)	1000	0.0742	26.93
	6 (NumPy)	2000	0.498	32.12

4.1 Observations

Scaling. Dividing runtime by n^3 gives nearly flat curves for C++ Methods 1–4 and NumPy, which is exactly what cubic complexity predicts. Small sizes wobble because fixed overheads, timer noise, and low arithmetic intensity dominate.

Absolute time. The plain triple loop (Method 1) explodes in cost and is unusable past $n \approx 1800$. Adding a transpose (Method 2) helps, but not enough to make it competitive. Blocking (Method 3) cuts times dramatically, and the BLAS routine (Method 4) wins outright across the board.

Throughput. DGEMM sustains about 30–37 GFLOPS on this CPU, with NumPy trailing just behind since it calls BLAS under the hood. The blocked kernel lands in the 5–9 GFLOPS range, though it stumbles around $n \approx 1100$ (likely cache or TLB oddities). Methods 1–2 limp along at 0.5–2.5 GFLOPS, bottlenecked by memory access.

Cache and tiling. The jump from Method 2 to Method 3 shows why tiling matters. Cutting the working set to $n_b \times n_b$ tiles keeps data in cache and lets each load do more useful work. With 512 KB L2, $n_b = 128$ is a sensible choice; testing nearby sizes could squeeze out a bit more speed.

Block size Picking the block size n_b was basically a game of fitting three blocks (A, B, and C) into L2 cache without blowing it up. With 512 KB available, $n_b = 128$ made sense because the working set is about 0.39 MB, which sits comfortably in cache. The performance dip around $n \approx 1100$ shows that hardware is never perfectly predictable, but overall the choice was reasonable. Achieved performance of 30–37 GFLOPS for BLAS is about 70–85% of the theoretical single-core peak ($\sim$ 43 GFLOPS at 2.7 GHz with AVX2). For a laptop CPU running under WSL, that's pretty respectable.

5 Discussion

The experiments highlight how strongly memory access patterns shape performance in dense AB. Adding a transpose (Method 2) improves locality compared to the naive loop, but the working set is still too large to be efficient. Tiling (Method 3) reduces the active footprint so that A, B, and C blocks are reused in cache, boosting arithmetic intensity and closing much of the gap to BLAS. The optimized library routine (Method 4) goes further with data packing, prefetching, vector-aware micro-kernels, and loop unrolling, which is why it consistently beats my handwritten code. On the Python side, raw loops (Method 5) are hopelessly slow, while NumPy (Method 6) essentially matches BLAS since it delegates everything to the same routines under the hood.

6 Reproducibility

To reproduce:

1. Build with `make` (OpenBLAS linked).

2. `source env_single_thread.sh`
3. Run C++ and Python commands as listed above.
4. Generate plots via `scripts/plot_results.py`.

7 Conclusion

Raw python performed horribly. Standard textbook algorithm on C++ works faster than python simply by the virtue of being compiled and optimized. On my system, the block method works reasonably well, almost as well as Numpy or BLAS. NumPy achieves slightly worse performance by invoking BLAS under the hood. The data follow the expected $O(n^3)$ trend with overhead-driven deviations at small sizes. The best result is of course from BLAS because some underpaid Chinese PhD figured all out and optimized everything in assembly a long time ago.

Acknowledgments

I credit Youtube and Chatgpt for the help.

Honor Statement

I affirm that this work complies with Nazarbayev University academic integrity policies and the policies regarding the use of AI tools outlined in the course syllabus.