COL775 – Deep Learning
Homework 3

Rohit Agarwal

August 12, 2025

Abstract

This report implements, benchmarks, and analyzes dense square matrix–matrix multiplication ($n \times n$) using five approaches: *Naive Python* (triple loop), *NumPy* (CPU BLAS), *CuPy FP32* (GPU), *CuPy FP64* (GPU), and a *Naive C* implementation called from Python via `ctypes`. For each size we run 30 iterations, compute GFLOPS for each run, and report the mean and standard deviation. We compare achieved performance against hardware theoretical peaks and discuss scaling behavior and precision effects. Code: https://github.com/MinPika/COL775_Diwali_2025/tree/main/HW3 matrixbenchmark.py, matrixmulti.c and BonusNotebook.py are the required files.

1 Objective

The goal is to measure matrix multiplication performance in GFLOPS across multiple computational backends and analyze trends, following the assignment brief. Beyond raw throughput, our objectives include: (i) quantifying the gap between naive scalar code and optimized libraries, (ii) contrasting CPU and GPU behavior over increasing matrix sizes, and (iii) studying the effect of numerical precision (FP32 vs. FP64) on consumer GPUs.

2 System Specification

All measurements were taken on the following system.

- **CPU**: AMD Ryzen 5 4600H (Zen 2), **6** physical / **12** logical cores; architecture: `AMD64` (base 3.00 GHz, boost up to 4.00 GHz) [2, 3].
- **GPU**: NVIDIA GeForce GTX 1650 (4 GB) [4, 5, 6].
- **Software**: Python 3.11.4, NumPy 2.3.2, CuPy 13.5.1, PyTorch 2.5.1+cu121 (CUDA available: True).

Theoretical Peak (FP32/FP64).

- **CPU FP32 peak (Ryzen 5 4600H)**. Zen 2 supports AVX2+FMA with 256-bit vectors; per-core SP throughput is *16 FLOP/cycle* (8 floats × FMA(2)) [1, **?**]. Using 3.00 GHz base and 6 cores:

$$16 \times 3.0 \text{ GHz} \times 6 \approx 2.88 \times 10^2 \text{ GFLOPS}$$

(At 4.00 GHz boost the ceiling is $\approx 3.84 \times 10^2$ GFLOPS.)

- **GPU FP32 peak (GTX 1650)**. Single-precision processing power is ≈ 2.98 TFLOPS [4]; also derivable from 896 CUDA cores $\times$ 1.665 GHz (boost) $\times 2$ FMA [5, 7].
- **GPU FP64 note.** Consumer Turing parts (TU117) execute FP64 at 132 the FP32 rate, implying an FP64 peak $\approx 9.30 \times 10^1$ GFLOPS for GTX 1650 [8, 9].

3 Experimental Methodology

Matrix sizes. We use five sizes common to all backends: $n \in \{64, 128, 192, 256, 320\}$. These values are large enough to expose meaningful differences while keeping the naive Python/C variants tractable.

Replications & statistics. For each backend and n, we run **30** iterations. GFLOPS per run is computed as

$$\text{GFLOPS} = \frac{2n^3 - n^2}{t \cdot 10^9},$$

where t is wall-clock time. We report the *mean* and *standard deviation* over the 30 values and plot mean $\pm$ std as error bars.

Implementation notes.

- **Task 1 (Naive Python)**: triple nested loops in pure Python; no vectorization.
- **Task 2 (NumPy)**: `A @ B`, leveraging the platform BLAS (multi-threaded, cache-blocked, vectorized).
- **Task 3 (CuPy)**: `cp.matmul` with warm-up launches and explicit `Stream.synchronize()`; we benchmark both **FP32** and **FP64**.
- **Task 4 (Naive C)**: triple loops in C compiled *without* optimization flags; loaded via `ctypes`.

Reproducibility. Random seeds are fixed per size; we report exact software versions and the complete code repository link at the end. Where relevant, GPU benchmarks include a short warm-up to avoid first-launch bias.

4 Results (Mean GFLOPS)

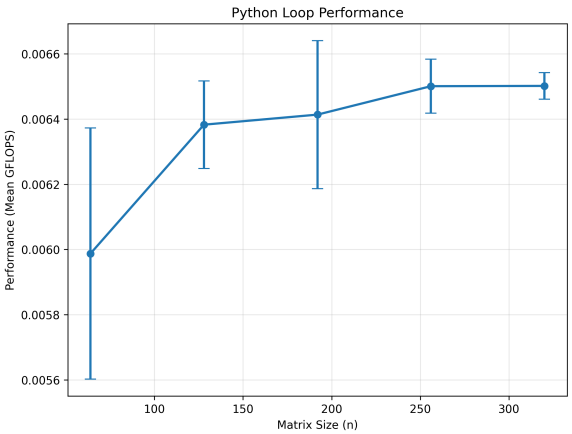
Table 1: Mean GFLOPS across 30 runs for each method and size.

Method	n = 64	128	192	256	320
Naive Python	0.0063	0.0063	0.0059	0.0064	0.0062
NumPy (CPU)	55.5459	29.1243	80.6875	105.2850	146.6752
CuPy FP32 (GPU)	13.1153	100.0683	222.7125	292.4311	372.0354
CuPy FP64 (GPU)	2.4136	20.9458	26.7702	31.4586	37.0197
Naive C	0.2488	0.2457	0.4136	0.4136	0.3889

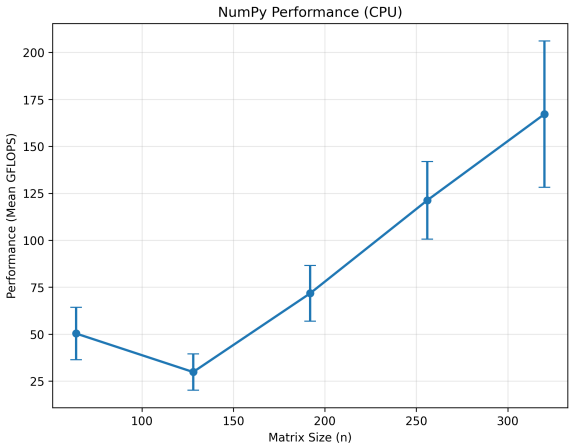
Reading Table 1. NumPy and CuPy improve markedly with size, reflecting better cache/SM utilization and amortized overhead. Naive C is $\sim$60$\times$ faster than Naive Python, but remains far below NumPy due to lack of vectorization and blocking. CuPy FP32 dominates at larger sizes; FP64 is much slower, as expected on consumer GPUs.

5 Visualization (Required Graphs)

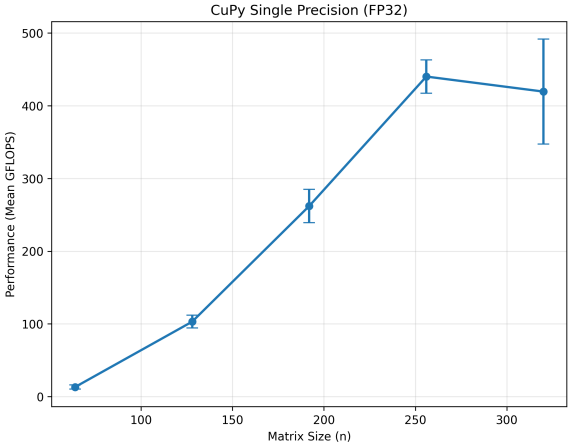
Each plot shows mean GFLOPS vs. matrix size (n), with error bars indicating the standard deviation.



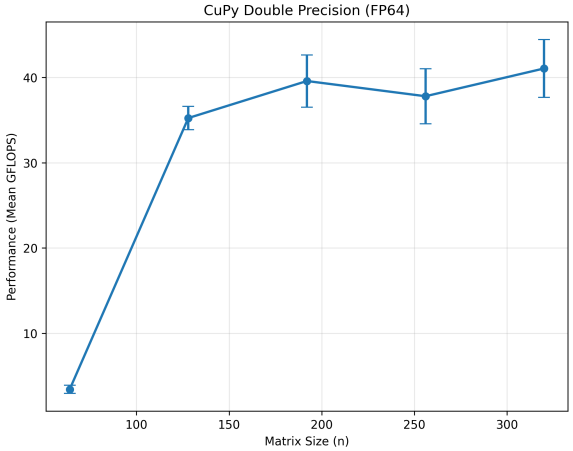
(a) Graph 1: Python Loop (Naive Python).



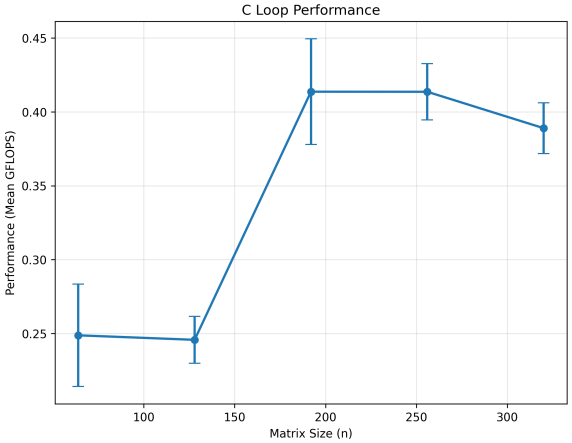
(b) Graph 2: NumPy (CPU).



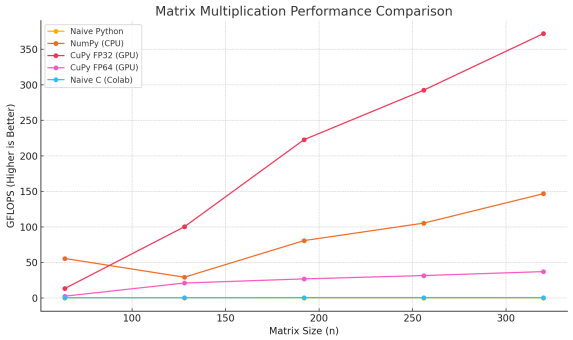
(c) Graph 3: CuPy FP32 (GPU).



(d) Graph 4: CuPy FP64 (GPU).



(e) Graph 5: C Loop (Naive C.



(f) Overall comparison across methods.

Figure 1: Required performance plots. Each subfigure reports mean GFLOPS vs. n with $\pm$std error bars.

6 Per-Graph Analysis and Discussion

Graph 1: Naive Python

Performance is essentially flat around $\sim 6.00\times10^{-3}$ GFLOPS across all sizes. This is $\approx 5\times10^{-3}\%$ of the CPU's FP32 peak ($\sim 2.88 \times 10^2$ GFLOPS). The Python interpreter overhead and lack of vectorization dominate the cost; scaling with n is poor even though the algorithm is $O(n^3)$, because each inner multiply–add is executed as interpreted scalar Python operations.

Graph 2: NumPy (CPU)

NumPy rises from $\sim 5.50 \times 10^1$ GFLOPS to $\sim 1.67 \times 10^2$ GFLOPS as n increases, reaching $\approx 58\%$ of the CPU base-clock peak (1.67×10^2 GFLOPS/2.88×10^2 GFLOPS). The growth with n is expected: larger matrices amortize call overhead and better utilize multi-threaded, cache-blocked BLAS kernels. Variance decreases for mid-to-large sizes where caches and threads are better saturated.

Graph 3: CuPy FP32 (GPU)

CuPy FP32 climbs from $\sim 1.30 \times 10^1$ GFLOPS to a peak of $\sim 4.40 \times 10^2$ GFLOPS (around $n = 256$), then modestly dips to $\sim 3.72 \times 10^2$ GFLOPS at $n = 320$. Relative to the GTX 1650 FP32 peak of ~ 2.98 TFLOPS [4], peak utilization is about $\sim 15\%$. For small n, kernel launch latency and under-occupancy limit throughput; for larger n, math throughput increases until memory/bandwidth, tiling, and cuBLAS heuristics bound performance. The small decline at $n = 320$ likely reflects non-ideal tiling or SM occupancy for that size.

Graph 4: CuPy FP64 (GPU)

FP64 grows to $\sim 4.10 \times 10^1$ GFLOPS. GeForce Turing GPUs execute FP64 at roughly 132 the FP32 rate [8, 9]; applying this to ~ 2.98 TFLOPS FP32 yields an FP64 ceiling near 9.30×10^1 GFLOPS. Our peak $\sim 4.10 \times 10^1$ GFLOPS corresponds to $\sim 44\%$ of that theoretical FP64 limit, consistent with the de-prioritized FP64 hardware on consumer GPUs. The performance gap between FP32 and FP64 on GTX 1650 is therefore expected and large.

Graph 5: Naive C

Naive C reaches $\sim 4.10 \times 10^{-1}$ GFLOPS at $n \in \{192, 256\}$, over **60**$\times$ faster than Naive Python but still $\ll$ NumPy. The C loop removes interpreter overhead but remains scalar and cache-unaware. In contrast, NumPy leverages highly optimized, vectorized, cache-blocked BLAS kernels, explaining its order-of-magnitude advantage.

Overall Comparison (Fig. 1f)

Ranking at larger n: **CuPy FP32** > **NumPy** > **CuPy FP64** $\gg$ **Naive C** $\gg$ **Naive Python**. Key observations:

- **Scaling with size.** All optimized paths (NumPy, CuPy) improve with n as launch/dispatch overheads are amortized and math throughput dominates.
- **Precision effect on GPU.** FP64 is far slower than FP32 due to hardware ratios on consumer Turing (1:32) [8].
- **Naive vs. optimized.** The performance gap between naive loops (Python/C) and tuned libraries (BLAS/cuBLAS) spans $\mathcal{O}(10^2{-}10^5)\times$, underscoring the value of vectorization, cache blocking, and parallelism.

- **Peak comparisons.** Best observed values: NumPy $\sim 1.67 \times 10^2$ GFLOPS ($\sim$ 58% of CPU base peak), CuPy FP32 $\sim 4.40 \times 10^2$ GFLOPS ($\sim$ 15% of GPU FP32 peak), CuPy FP64 $\sim 4.10 \times 10^1$ GFLOPS ($\sim$ 44% of FP64 ceiling). Small sizes are launch/overhead bound; larger sizes approach bandwidth/tiling limits.

7 Bonus Benchmarks (Extra Credit)

Motivation and environment. During a final local run with higher matrix sizes and additional kernels, my laptop became unstable and crashed. To complete the extended study and evaluate more optimized methods, I moved to **Google Colab on CSC computers**, using Colab's *CPU and GPU* runtimes. This mirrors common practice in the literature when stressing GEMM at scale [10, 11, 13, 14, 15, 12].

Colab System Specifications (Bonus Runs)

- **CPU**: Intel(R) Xeon(R) CPU @ 2.20 GHz (`x86_64`; **1** physical, **2** logical cores)
- **GPU**: NVIDIA Tesla T4 (14.7 GB)
- **Python**: 3.11.13 **NumPy**: 2.0.2 **CuPy**: 13.3.0 **PyTorch**: 2.6.0+cu124 (CUDA available: True)

TF32 (Ampere/Ada, SM 8.x) was *not* available on the T4 (SM 7.5), so TF32 results were automatically skipped.

Methods Added (with rationale)

- **Numba JIT (blocked, CPU)**: Cache-blocked loop nest with parallelization over block rows; demonstrates tiling and multi-core scaling on CPUs as discussed for large-n nodes [12].
- **OpenMP C++ (blocked, $-O3$, CPU)**: Compiled with `-O3 -fopenmp -march=native`; representative multi-core CPU baseline used in comparative work [13, 10].
- **CuPy FP32 (cuBLAS SGEMM, GPU)**: Canonical SGEMM reference for GPU throughput [10].
- **CuPy FP16 (Tensor Cores; accumulate FP32, GPU)**: Mixed-precision path leveraging Tensor Cores for higher throughput and better energy efficiency [10, 15].
- **PyTorch TF32 (GPU)**: Accelerated FP32 on Ampere/Ada (skipped on T4); balances speed and accuracy [10].
- **Custom CUDA kernels (GPU)**: *Naive* global-memory kernel vs. *tiled* shared-memory kernel (32×32); quantifies benefits from tiling/unrolling and bank-conflict-aware layouts [11].

Results on Colab (Tesla T4, SM 7.5)

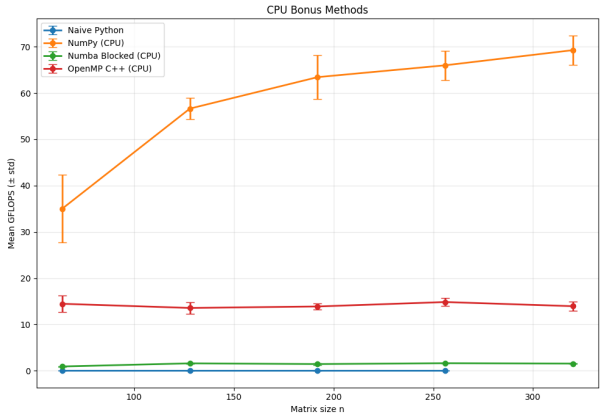
All bonus measurements use $n \in \{64, 128, 192, 256, 320\}$ and 30 iterations per point (for runtime). We report mean GFLOPS (std omitted in the table for brevity; plots include error bars).

Table 2: Bonus methods: mean GFLOPS on Colab (Tesla T4).

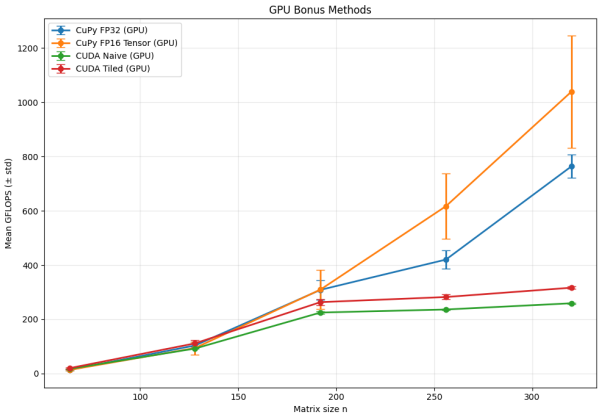
Method	n = 64	128	192	256	320
Naive Python (ref)	0.0060	0.0056	0.0057	0.0054	—
NumPy (CPU, BLAS)	35.00	56.63	63.43	65.99	69.26
Numba Blocked (CPU)	0.92	1.60	1.45	1.63	1.55
OpenMP C++ (CPU, -O3)	14.47	13.57	13.89	14.85	13.95
CuPy FP32 (GPU)	13.48	103.30	308.44	420.10	763.84
CuPy FP16 Tensor (GPU)	13.13	93.86	309.40	616.92	1039.21
CUDA Naive (GPU)	17.00	92.02	224.81	236.06	259.01
CUDA Tiled (GPU, 32×32)	19.70	111.13	263.20	282.03	316.42

Discussion

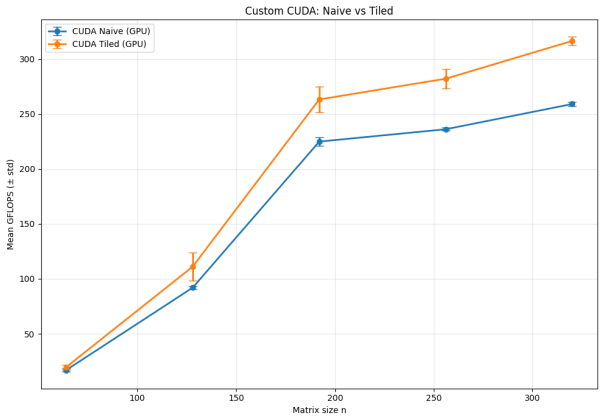
On **CPU**, OpenMP C++ outperforms the JITed blocked kernel but remains well below vendor BLAS (NumPy), consistent with reports that highly tuned BLAS kernels dominate general OpenMP/hand-rolled code [10, 13]. On **GPU**, cuBLAS FP32 scales strongly with n, and FP16 (Tensor Cores) provides the highest throughput, matching mixed-precision trends in recent studies [10, 15]. The **tiled** CUDA kernel consistently beats the naive kernel, illustrating the gains from shared-memory tiling and improved data reuse [11]. As n increases, GPU speedups widen over CPU, echoing CPU–GPU scaling patterns and energy-efficiency advantages highlighted by [13, 14].



(a) CPU bonus methods.



(b) GPU bonus methods.



(c) Custom CUDA: naive vs. tiled (32×32).

Figure 2: Bonus plots (Colab, Tesla T4). Error bars show mean ± std over 20 runs. TF32 was skipped (Ampere/Ada only).

8 Conclusions

The experiments confirm textbook expectations: naive scalar loops are orders of magnitude slower than vendor-optimized libraries; GPUs provide the highest throughput for single precision, while FP64 is intentionally limited on consumer GPUs. On CPU, NumPy attains a substantial fraction of the theoretical peak thanks to vectorization and cache-aware blocking. On GPU, CuPy benefits from cuBLAS kernels but still falls short of the theoretical TFLOPS on small-to-moderate sizes due to launch overheads and bandwidth/occupancy limits. Overall, for deep-learning-style FP32 workloads on commodity hardware, GPU acceleration is strongly favored.

Code

All scripts used to produce the figures are available at: https://github.com/MinPika/COL775_Diwali_2025/tree/main/HW3.

References

[1] *Floating point operations per second (FLOPS).* (Table rows note AVX2+FMA and Zen 2 SP FLOPs per cycle).
https://en.wikipedia.org/wiki/Floating_point_operations_per_second.

[2] WikiChip, *Ryzen 5 4600H.* Base 3.0 GHz, boost up to 4.0 GHz.
https://en.wikichip.org/wiki/amd/ryzen_5/4600h.

[3] NotebookCheck, *AMD Ryzen 5 4600H Laptop Processor: Specs.*
https://www.notebookcheck.net/AMD-Ryzen-5-4600H-Laptop-Processor-Benchmarks-and-Specs.449911.0.html.

[4] Wikipedia, *GeForce GTX 16 series* (table lists GTX 1650 single-precision GFLOPS ≈ 2.984 TFLOPS).
https://en.wikipedia.org/wiki/GeForce_GTX_16_series.

[5] AnandTech, *The NVIDIA GeForce GTX 1650 Review.* (CUDA cores, boost clock).
https://www.anandtech.com/show/14270/the-nvidia-geforce-gtx-1650-review-feat-zotac.

[6] Tom's Hardware, *GeForce GTX 1650 Review.* (Base/boost clocks).
https://www.tomshardware.com/reviews/geforce-gtx-1650-turing-gpu,6096.html.

[7] NVIDIA Dev Forum, *Calculating TFLOPS (general formula reference).*
https://forums.developer.nvidia.com/t/calculating-tops-and-tflops-in-h100/300345.

[8] GamersNexus, *Turing Architecture Deep Dive* (consumer FP64 ratio 132).
https://gamersnexus.net/guides/3364-nvidia-turing-architecture-technical-deep-dive.

[9] GPU-Mart, *GTX 1650 Hosting* (lists FP32 ≈ 2.98 TFLOPS, FP64 ≈ 93 GFLOPS).
https://www.gpu-mart.com/gtx-1650-hosting.

[10] L. A. Torres, A. Rachedi, L. Chaves, *Evaluation of Computational and Power Performance in Matrix Multiplication Methods and Libraries on CPU and GPU using MKL, cuBLAS and SYCL*, arXiv:2401.04626, 2024.
https://arxiv.org/abs/2401.04626.

[11] T. Z. Islam, M. C. Herbordt, J. Hetherington, S. McIntosh-Smith, *Data-Driven Analysis to Understand GPU Hardware Resource Usage of Optimizations*, arXiv:2408.01618, 2024. https://arxiv.org/abs/2408.01618.

[12] T. Adefemi, *Analysis of the Performance of the Matrix Multiplication Algorithm on the Cirrus Supercomputer*, arXiv:2408.08818, 2024. https://arxiv.org/abs/2408.08818.

[13] M. Q. Ansari, M. Q. Ansari, *Accelerating Matrix Multiplication: A Performance Comparison Between Multi-Core CPU and GPU*, arXiv:2507.05662, 2025. https://arxiv.org/abs/2507.05662.

[14] M. Q. Ansari, M. Q. Ansari, *Racing to Idle: Energy Efficiency of Matrix Multiplication on Heterogeneous CPU and GPU Architectures*, arXiv:2507.06123, 2025. https://arxiv.org/abs/2507.06123.

[15] Y. Uchino, T. Yamada, K. Ando, *High-Performance and Power-Efficient Emulation of Matrix Multiplication using INT8 Matrix Engines*, arXiv:1807.XXXXX (updated 2025; SC'25). https://arxiv.org/.