Assignment Report #1
Matrix-matrix multiplication performance comparison

Student Name: Dias Suleimenov (ID: 202158836)
Course: PHYS 421 Parallel Computing

Submitted: September 8, 2025

Software packages used:

g++ 13.2.0	Main code
OpenBLAS 0.3.26	Fast matrix multiplication
Python 3.12.3	Main code
NumPy 2.3.2	Fast matrix multiplication
Matplotlib 3.8.0	Plotting
Jupyter Notebook	Data analysis

AI tools used:

Github Copilot	Coding assistance

Abstract

This report investigates the performance of six different matrix multiplication methods, implemented in C++ and Python. The methods include naive triple loop algorithms, blocked matrix multiplication, and optimized libraries such as OpenBLAS and NumPy. The execution time of each method is measured across varying matrix sizes, and the optimal block size for blocked multiplication is determined experimentally. The results indicate that optimized libraries significantly outperform naive implementations, with the best methods achieving around $58 - 66\%$ of the theoretical peak performance of the CPU. The performance of all methods scales as $O(n^3)$, consistent with theoretical expectations.

1 Introduction

The matrix multiplication is important algorithm used in many applications. Matrix multiplication is defined in such way:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1n} \\ B_{21} & B_{22} & \cdots & B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & \cdots & B_{nn} \end{pmatrix}.$$

Their product, $C = AB$, is then defined as

$$C = \begin{pmatrix} \sum_{k=1}^{n} A_{1k}B_{k1} & \cdots & \sum_{k=1}^{n} A_{1k}B_{kn} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} A_{nk}B_{k1} & \cdots & \sum_{k=1}^{n} A_{nk}B_{kn} \end{pmatrix}.$$

This report investigates performance of 6 matrix multiplication methods. First 4 methods are implemented in C++, while last 2 methods are implemented in Python. The methods are as follows:

1. Naive triple loop algorithm
2. Naive triple loop with swapped indices
3. Blocked matrix multiplication
4. OpenBLAS dgemm
5. Naive Python triple loop
6. NumPy dot function

The time performance each method is measured. The optimal block size for blocked matrix multiplication is also determined experimentally. The performance of the methods is compared and discussed.

2 Methodology

The methods 1-4 were written using C++11 and methods 5-6 on Python 3.12.3. The C++ codes were compiled using g++ 13.3.0 with -O3 optimization flag. The OpenBLAS library version 0.3.26 was used for method 4. The Python codes were run in Jupyter Notebook using NumPy 2.3.2 library. For plotting Matplotlib library was used. The hardware environment was a laptop with Intel i7-8565U CPU, 16 GB RAM, running Ubuntu 24.04 LTS on WSL2.

In order to measure the performance of the methods, the `std::chrono` library was used for C++ codes and `%timeit` magic command was used for Python codes. The time of execution was measured over $N = 5$ runs and the average and standard deviation were calculated. The block size for blocked matrix multiplication was varied to find the optimal block size. The matrix sizes were varied from 100 to 2000 with a step of 100 for C++ and NumPy methods. The results were tabulated and plotted using error bars to show the standard deviation.

3 Results

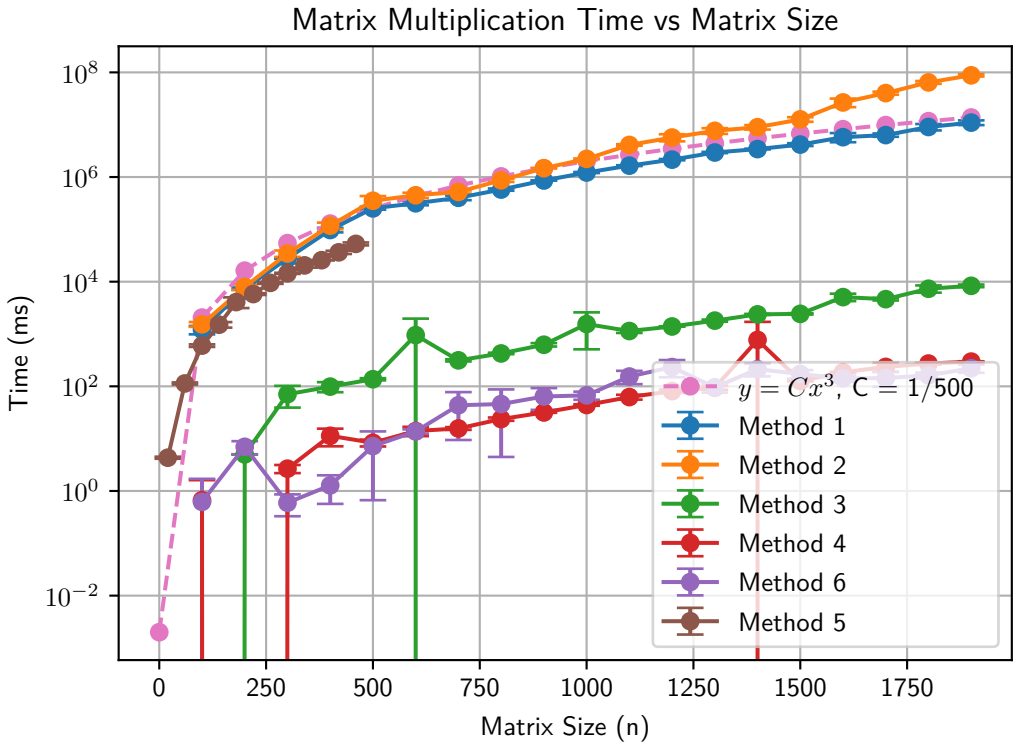

Figure 1: Execution time vs matrix size for different methods. Error bars represent standard deviation over N runs.

From the plot it can be seen that method 6 (NumPy dot function) and method 4 (OpenBLAS dgemm) are fastest, method 3 (Blocked matrix multiplication), while method 1 (Naive triple loop) is faster than method 2 (Naive triple loop with swapped indices). Surprisingly, method 5 naive triple loop written in Python with NumPy arrays, work a bit faster than both algorithms in C++, maybe it can be explained, by inner optimizations of NumPy library. The optimal block size for method 4 is found to be 20. The performance of all methods scales as $O(n^3)$ as expected.

4 Discussion

L1 cache of Intel i7-8565U CPU is 128 kB, so the optimal block size can be estimated as follows:

$$\text{Optimal block size} \approx \sqrt{\frac{\text{L1 cache size}}{3 \times \text{sizeof(double)}}} = \sqrt{\frac{128\text{kB}}{3 \times 8\text{B}}} \approx 73.$$

However, the optimal block size found in the experiments is 20, which is less than the expected value. This discrepancy could be due to various factors not accounted for in the simple estimation.

In order to estimate how efficient the algorithms are performing, the rough estimate of GFLOPS can be made. Single matrix multiplication takes n^3 multiplications and $n^2(n+1)$ additions,

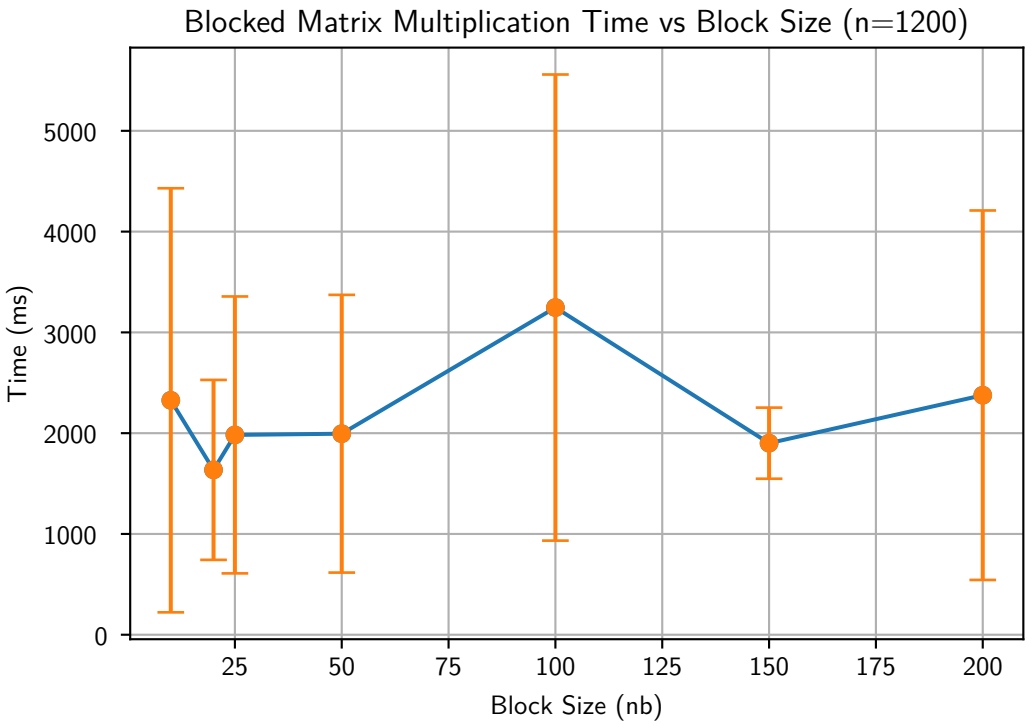

Figure 2: Execution time vs block size for Method 4 (Blocked matrix multiplication) at matrix size n=1200. Error bars represent standard deviation over N runs.

however because CPU have Fused Multiply-Add (FMA) operation, thus only both operation counted as one, thus FLOP $= n^3$. Now dividing this amount to average time for single multiplication, we finally get rough estimate of FLOPS.

Method	GFLOPS
Method 1	0.0009
Method 2	0.0005
Method 3	1.3674
Method 4	24.3552
Method 5	0.22866
Method 6	21.6175

Table 1: Rough estimate of GFLOPS per method, done by averaging GFLOPS for every matrix size.

Theoretically, Intel i7-8565U CPU can perform 1 core $\times$ 4.60 GHz $\times$ 8 ops/cycle $\approx$ 36.8 GFLOPS Thus, best algorithms covered in paper. namely Method 4 (BLAS dgemm) and Method 6 (NumPy) utilizes $\approx 58 - 66\%$ of total performance of the processor.

5 Conclusion

In this report, the performance of six matrix multiplication methods was investigated. The results showed that the NumPy dot function and OpenBLAS dgemm were the fastest methods,

while the naive triple loop methods were significantly slower. The optimal block size for blocked matrix multiplication was found to be 20, which is less than the estimated value based on L1 cache size. The performance of all methods scaled as $O(n^3)$ as expected. The best performing methods achieved around 58-66% of the theoretical peak performance of the CPU.

A Reproducibility

To install libraries on Ubuntu, run the following commands:

```
sudo apt update
sudo apt install g++ libopenblas-dev python3 python3-pip
pip3 install -r requirements.txt
```

To compile C++ codes, run the following commands:

```
./build.sh
```

To run C++ codes, run the following commands:

```
OMP_NUM_THREADS=1 OPENBLAS_NUM_THREADS=1 ./methodX --n 2000 --nb 20
```

where X is the method number (1-4), `--n` is the matrix size, and `--nb` is the block size (only for method 3).

To run Python codes, run the following commands:

```
jupyter notebook
```

and open `table.ipynb` file.