

# **Matrix Multiplication Speedup**

Uzair Hamed Mohammed

CSC 871, Spring 2026

Due 2/17

## Contents

1. Introduction .....	1
2. Methodology .....	1
2.1. Matrix Generation .....	1
2.2. Plain Python Multiplication .....	1
2.3. Conversion to PyTorch Tensors .....	2
2.4. Vectorized Multiplication .....	2
2.5. Timing Procedure .....	2
3. Results .....	3
4. Conclusion .....	3
5. Acknowledgments .....	4
References .....	4

## 1. Introduction

Matrix multiplication is a common operation in deep learning, and its raw implementation in pure Python, using nested loops, can be extremely slow for large matrices. To solve this issue, libraries like PyTorch leverage highly optimized, vectorized routines to achieve incredible speedups. In this assignment, we quantify this performance improvement by measuring the execution time of multiplying two matrices  $W(90 \times m)$  and  $X(m \times 110)$  for ten values of iterator  $m$ ,  $10 \leq m \leq 100$ . We compare a plain Python loop implementation against PyTorch’s built-in matrix multiplication (via the @ operator) and plot the resulting speedup ratio. The results confirm that vectorized operations are orders of magnitude faster, with the speedup increasing as the matrix dimensions grow.

## 2. Methodology

Since I am using a  Snapdragon X (ARM64) computer without a CUDA-compatible GPU, I optimized the PyTorch implementation by explicitly casting the tensors to float32. While the prebuilt PyTorch distribution I downloaded likely leverages a general-purpose BLAS library like OpenBLAS, this library itself contains optimized kernels for ARM64. These kernels leverage the CPU’s ARM NEON<sup>1</sup> vector instructions, providing a significant speedup over the standard Python implementation while maintaining high precision.

### 2.1. Matrix Generation

```
def gen_random_matrix(rows, cols):
    return [[random.random() for _ in range(cols)] for _ in range(rows)]
```

This simple function takes in the specifications of a matrix as `rows` and `columns`, and returns a matrix of that size. I chose to write this helper function to keep the code clean and easily understandable.

### 2.2. Plain Python Multiplication

The “vanilla” matrix multiplication is implemented with three nested `for` loops, following the definition

$$(WX)_{i,j} = \sum_{k=1}^m W_{i,k}X_{k,j}$$

The code below creates a result matrix of zeros and fills it by iterating over rows of  $W$ , columns of  $X$ , and the common dimension  $m$ .

```
def vanilla_matmul(W, X):
    rows_W = len(W)
    cols_W = len(W[0])
    rows_X = len(X)
    cols_X = len(X[0])
    if cols_W != rows_X:
        raise ValueError("Matrix dimensions do not match")
    result = [[0.0 for _ in range(cols_X)] for _ in range(rows_W)]
    for i in range(rows_W):
        for j in range(cols_X):
            total = 0.0
            for k in range(cols_W):
                total += W[i][k] * X[k][j]
```

---

<sup>1</sup>ARM NEON technology: <https://www.arm.com/technologies/neon>

```

        result[i][j] = total
    return result

```

A quick test with small matrices confirmed the function produces the correct product.

```

def test_vanilla():
    w = [[2,2], [3,4]]
    x = [[5,6], [7,8]]

    print("Multiplying matrices:")
    multot = vanilla_matmul(w, x)
    print(multot)

test_vanilla() # Returns [[24.0, 28.0], [43.0, 50.0]]

```

### 2.3. Conversion to PyTorch Tensors

After generating the plain Python matrices for a given value of  $m$ , they are converted to PyTorch tensors  $W_t$  and  $X_t$ . This is achieved using the `torch.tensor()` function, which creates a tensor object that can leverage PyTorch's optimized operations.

```

W_t = torch.tensor(W, dtype=torch.float32)
X_t = torch.tensor(X, dtype=torch.float32)

```

To ensure efficient use of the CPU's SIMD capabilities, the tensors are explicitly cast to `torch.float32`. This conversion is performed outside the timed code blocks so that the overhead of creating tensors does not affect the measurements of the multiplication itself.

### 2.4. Vectorized Multiplication

PyTorch provides a highly optimized matrix multiplication routine via the `@` operator. The operation

```
W_t @ X_t
```

computes the product in a single, vectorized step. Under the hood, PyTorch calls BLAS libraries that leverage CPU vector instructions, among other things. This results in execution speeds that are much faster than a pure Python loop, especially for larger matrices.

### 2.5. Timing Procedure

To obtain reliable execution times, the `timeit` module was utilized. As per instructions, the code loops over the ten values of  $m = 10, 20, \dots, 100$ . For each  $m$ :

1. New matrices  $W$  and  $X$  are generated using the `gen_random_matrix()` function.
2. The plain Python multiplication function is timed by calling

```

REPT_VANILLA = 10
#...
timeit.timeit(lambda: vanilla_matmul(W, X), number=REPT_VANILLA)

```

where `REPT_VANILLA` is the number of repetitions. Because the plain loop is relatively slow, a small number of repetitions proves enough to obtain a stable average while keeping the total runtime manageable.

3. The average time per plain multiplication is computed as  $\frac{\text{total time}}{\text{number of repetitions}}$ .

4. The matrices are converted to PyTorch tensors as described above.
5. The vectorized multiplication is timed with

```
REPT_VEC = 1000
#...
timeit.timeit(lambda: W_t @ X_t, number=REPT_VEC)
```

Since the vectorized operation is extremely fast, a larger number of repetitions is used so that the total measured time is large enough to be accurate.

6. The average time per vectorized multiplication is computed as  $\frac{\text{total time}}{\text{number of repetitions}}$ .

The choice of different repetition counts for the two methods does not bias the comparison because we always compare the average time per single multiplication. This code yields ten pairs of average times, one pair for each  $m$ .

### 3. Results

Figure 1 [1] shows the speedup ratio  $\frac{t_{\text{plain}}}{t_{\text{vec}}}$  plotted against the common dimension  $m$ .

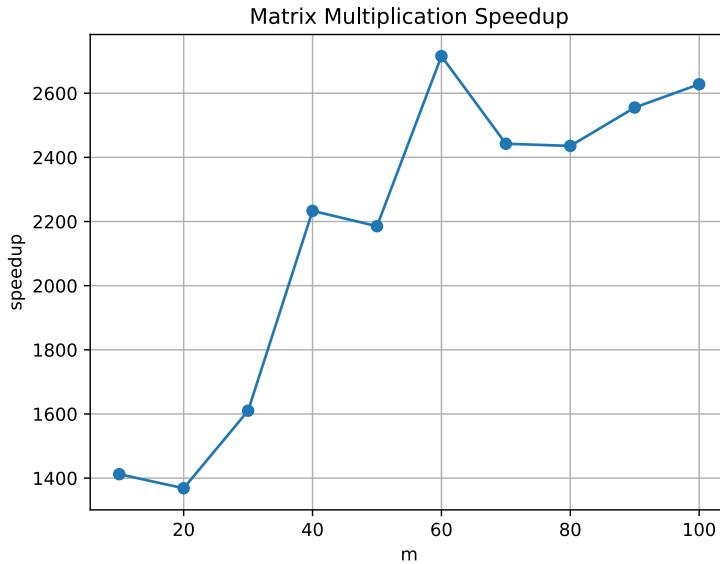


Figure 1: Speedup ratio of plain over vectorized multiplication.

The speedup increases from approximately 1390 at  $m = 20$  to roughly 2700 at  $m = 60$ . The speedup stabilizes a little for following values until  $m = 90$ , where we start to see an increase again. The data points exhibit a clear upward trend as the size of  $m$  increases.

### 4. Conclusion

The experiment carried out in this assignment quantified the performance gap between a simple matrix multiplication function written in pure Python and the vectorized implementation provided by PyTorch. For matrices of size  $90 \times m$  and  $m \times 110$ , with  $m$  ranging from 10 to 100 in increments of 10, the vectorized version was found to be thousands of times faster, with the speedup generally increasing as the matrices grew larger. The results affirm the importance of using highly optimized libraries for such linear algebra operations, especially in deep learning where operations are performed repeatedly.

## 5. Acknowledgments

I acknowledge the use of DeepSeek [1] as a learning assistant for this assignment. The AI helped troubleshoot matplotlib display issues, refine Typst formatting, and improve the clarity of certain sections. All code, analysis, and conclusions are my own work.

## References

- [1] DeepSeek, “DeepSeek R1 assisted with matplotlib troubleshooting, Typst syntax, and paper formatting standards.” [Online]. Available: <https://www.deepseek.com/>
- [2] U. H. Mohammed, “Matrix Multiplication Speedup – Source Code.” [Online]. Available: <https://github.com/gamersekofy/csc-871>