

Matrix Multiplication: PyTorch vs. Plain Python Performance Comparison

Bryan Lee
922649673

February 12, 2025

1 Introduction

One of the well-known facts about Python is its relatively slow speed when performing large computational tasks. Compared to other popular programming languages like C++, Java, and even Node.js, Python tends to be slower in execution. One key reason for this performance difference is that Python is an interpreted language, meaning that while it first compiles code into byte code, it still executes instructions line by line at runtime rather than compiling everything into machine code beforehand. However, this limitation can be mitigated through specialized optimizations and external libraries. For instance, libraries like PyTorch use a data structure called a tensor to represent matrices. Tensors store data of a uniform type in contiguous memory, rather than as a collection of objects like Python lists. PyTorch can also leverage GPU acceleration to further improve performance compared to standard Python.

2 Preliminaries

Matrix multiplication is a common linear algebra operation in numerical computing and machine learning. It can be defined as a binary operation that produces a new matrix from two existing matrices. The said matrix multiplication process can be described as follows:

Let $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, and $C \in \mathbb{R}^{m \times p}$,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

The product C consist of:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj},$$

for $i = 1, \dots, m$ and $j = 1, \dots, p$.

Matrix multiplication implementation in plain Python:

```
def multiplyMatrices(A: list[list[float]], B: list[list[float]]) ->
    list[list[float]]:
    """
        Multiplies two matrices A and B using plain python for-loops.
    """
    if len(A[0]) != len(B):
        raise ValueError("Matrix dimensions are not compatible: number of columns
            of matrix A must equal the number of rows of matrix B!");

    # Init result matrix with all 0s
    C = [[0 for _ in range(len(B[0]))] for _ in range(len(A))];

    # Perform matrix multiplication
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                C[i][j] += A[i][k] * B[k][j];

    return C;
```

As we can see, the algorithm runs in $O(n^3)$ time complexity, which is an expensive operation. Additionally, the space complexity is also a concern, as each element in a plain Python list is stored as a separate object, leading to increased memory overhead. That begs the differ, how can we improve the computational speed of doing matrix multiplication?

As previously discussed, PyTorch has become a prominent tool in deep learning due to its flexibility and efficiency. While NumPy arrays are similar to PyTorch's tensor, NumPy is more suited for general-purpose array computing. It has limitations in deep learning applications, particularly because it lacks support for automatic differentiation and GPU acceleration. On the other hand, PyTorch's tensor is specifically designed for deep learning applications, so that will be the focus of our comparison.

3 Approach

To start, we will define a helper function that will generate two matrices, W and X, where W is a $90 \times m$ matrix, and X is a $m \times 110$ matrix, where $m \in \{10, 20, \dots, 100\}$.

Creating the randomized matrices:

```
import random
random.seed(737);

m_vals = [m for m in range(10, 101, 10)];

def generateMatrices(m: int) -> tuple[list[list[float]], list[list[float]]]:
    W = [[random.random() for _ in range(m)] for _ in range(90)];
    X = [[random.random() for _ in range(110)] for _ in range(m)];

    return W, X;
```

The main experiment script will iterate through each m values (representing different matrix sizes) to generate the corresponding W and X matrices, along with their tensor counterparts. It will then use IPython's `timeit` magic command to perform 7 runs of matrix multiplication for both the plain Python and vectorized implementations. We will then take the average of these runs, which will give us a more accurate estimate of the runtime for each implementation.

Running the 10 timing experiments:

```
import torch
import timeit

pyTimeResults = [];
tensorTimeResults = [];
for m in m_vals:
    print(f"Iteration with m value of {m}:");

    # Original list matrices
    W, X = generateMatrices(m);

    # Create the tensor matrices
    W_t = torch.tensor(W);
    X_t = torch.tensor(X);

    # Time the plain Python matrix multiplication
    pythonTime = %timeit -o multiplyMatrices(W, X);
    pyTimeResults.append(pythonTime.average);

    # Time the torch matrix multiplication
    tensorTime = %timeit -o torch.matmul(W_t, X_t);
    tensorTimeResults.append(tensorTime.average);
```

The `timeit` function produces an output for each iteration, shown below. The table reveals that, in the plain Python implementation, the magic function is executed only 10 times, while PyTorch runs it 10,000 times. Additionally, the time per loop is considerably lower for the PyTorch implementation, highlighting its superior speed for matrix multiplication.

Iteration with m value	Per loop (mean \pm std. dev.)	
	10 loops each (Plain Python)	10,000 loops each (PyTorch)
10	25.7 ms \pm 4.07 ms	23.9 μ s \pm 2.73 μ s
20	41.2 ms \pm 262 μ s	23 μ s \pm 329 ns
30	57.2 ms \pm 418 μ s	25 μ s \pm 1.34 μ s
40	81.6 ms \pm 8.88 ms	28.1 μ s \pm 1.18 μ s
50	96.7 ms \pm 2.26 ms	28.7 μ s \pm 688 ns
60	111 ms \pm 1.48 ms	32.2 μ s \pm 4.72 μ s
70	134 ms \pm 592 μ s	30.5 μ s \pm 570 ns
80	150 ms \pm 3.18 ms	32.2 μ s \pm 264 ns
90	172 ms \pm 608 μ s	38.2 μ s \pm 2.54 μ s

Table 1: Performance metrics for different m values.

There is an edge case for the last iteration with $m = 100$, where the plain Python implementation executes only 1 loop. This is likely due to performance and bottleneck considerations when handling such a large computation.

Iteration with m value	Per loop (mean \pm std. dev.)	
	1 loop (Plain Python)	10,000 loops (PyTorch)
100	183 ms \pm 3.94 ms	38.5 μ s \pm 3.03 μ s

Table 2: Performance metrics for m value of 100.

We can also see the significant difference between plain Python and PyTorch when we print out the average of the `timeit` function below.

Displaying the average time results for the matrix multiplication experiment:

```
print("Time results for timeit tests on regular python matrices:");
print(pyTimeResults);
```

```
Time results for timeit tests on regular python matrices:
[0.02568126999987206, 0.041184372857346066, 0.05717241571442823,
 0.08162844285720243, 0.09673111714291736, 0.1107446571429526,
 0.13369511714263355, 0.1497053142857372, 0.17162189714269646,
 0.1831118428568256]
```

```
print("Time results for timeit tests on tensor matrices:");
print(tensorTimeResults);
```

```
Time results for timeit tests on tensor matrices:
```

```
[2.386830857140012e-05, 2.3007501428635025e-05, 2.5019531428759887e-05,
 2.8062202857212727e-05, 2.870331714283176e-05, 3.223921571438301e-05,
 3.049636571408233e-05, 3.221754571433329e-05, 3.8243817142730616e-05,
 3.848362714291267e-05]
```

4 Visualizing the Results

Using the retrieved data, we will plot the speed improvement ratio between the plain Python and PyTorch implementations, calculated using the following formula:

$$\text{speed improvement ratio} = \sum_{i=0}^{i < \text{len}(m)} \frac{\text{pyTimeResults}[i]}{\text{tensorTimeResults}[i]}$$

Additionally, I will also compute the line of best fit for the data points using the least squares method. It will be computed as follows:

$$A = \begin{bmatrix} 1 & 10 \\ 1 & 20 \\ \vdots & \vdots \\ 1 & 100 \end{bmatrix}, \quad b = \begin{bmatrix} \text{speed_improvement_ratio}[0] \\ \text{speed_improvement_ratio}[1] \\ \vdots \\ \text{speed_improvement_ratio}[9] \end{bmatrix}$$

Where A is a vertical matrix with the first column consisting of all ones, and the second column consisting of the m values. On the other hand, b is a vertical vector of the speed improvement ratio values. Now for the computation:

$$x^* = (A^T A)^{-1} A^T b$$

Giving us the line of best fit line:

$$\text{bestFitLine}(m) = \alpha + \beta * m$$

Plotting the graph:

```
import matplotlib.pyplot as plt
import numpy as np

speedImprovementRatio = [pyTimeResults[i] / tensorTimeResults[i] for i in
    range(len(pyTimeResults))];

# Compute the line of best fit using least squares solution
A = np.column_stack((np.ones(len(m_vals)), m_vals)); # Col of 1s and m_vals
b = np.array(speedImprovementRatio); # Vertical vector of ratios

x_star = np.linalg.inv(A.T @ A) @ (A.T @ b); #  $x^* = (A' * A)^{-1} * A' * b$ 

alpha = x_star[0];
beta = x_star[1];

bestFitLine = [alpha + beta * m for m in m_vals]; # Formulate the best fit line

plt.figure(figsize=(10, 6));
plt.plot(m_vals, speedImprovementRatio, marker='o', linestyle='--', color='b',
    label="Speed Improvement");
plt.plot(m_vals, bestFitLine, marker='^', linestyle='--', color='r', label="Best
    Fit Line");

plt.xticks(m_vals);
plt.xlabel("Matrix Size (m)");
plt.ylabel("Speed Improvement Ratio (Plain / Vectorized)");
plt.title("Speed Improvement of PyTorch Vectorized Multiplication Over Plain
    Python");
plt.grid(True);
plt.legend();
plt.show();
```

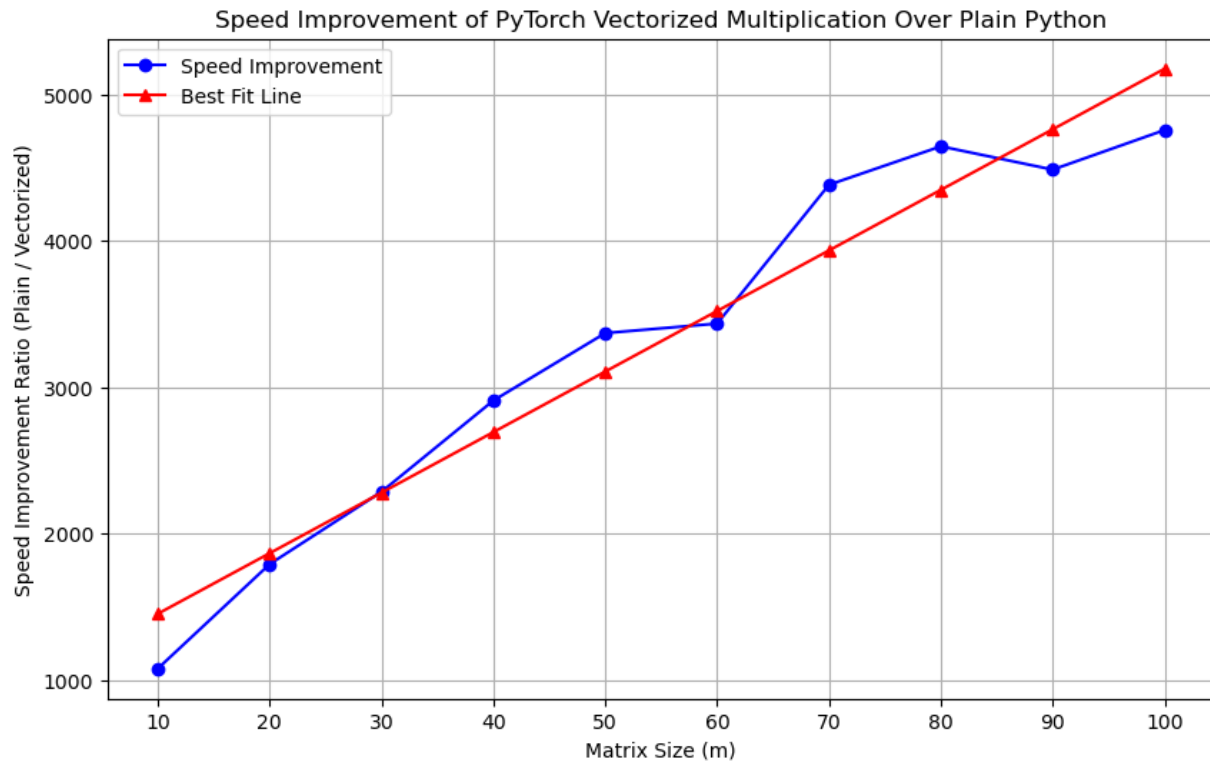


Figure 1: A simple diagram of the camera matrix: $x = PX$.

5 Conclusion

In conclusion, from the experiments, we can see that using PyTorch tensors for matrix multiplication offers a significant performance improvement over the plain Python implementation. By enforcing a homogenous data type for the tensors, the computation time for matrix multiplication can be significantly reduced. Not only can we reduce the need to reference a new object for each matrix element, we can also compact the size of its data by almost 6x less. Such optimizations are necessary for deep learning applications, where intensive matrix operations are frequently performed.

References

- [1] GeeksforGeeks. (2023). *Internal working of Python*. <https://www.geeksforgeeks.org/internal-working-of-python/>.
- [2] GeeksforGeeks. (2024). *PyTorch Tensor vs Numpy Array*. <https://www.geeksforgeeks.org/pytorch-tensor-vs-numpy-array/>.
- [3] Radovich, V. (2024). *Why tensors perform insanely faster than lists? – An introduction to storage*. <https://medium.com/@valentinradovich/why-tensors-perform-insanely-faster-than-lists-an-introduction-to-storage-bf36723bbaf2>.
- [4] Wikimedia Foundation. (2024). *Matrix multiplication*. Wikipedia. https://en.wikipedia.org/wiki/Matrix_multiplication.