

▼ Assignment 1

Matrix-matrix multiplication

Juan López-Ríos

SN: 20121702

INDEX:

- [Description](#)
- [Imports](#)
- [Part 1: Better Function](#)
 - [Part 1.1: Test and Visualize performance](#)
- [Part 2: JIT Implementation](#)
 - [Part 2.1: Fortran/C-style](#)
- [Part 3: Extensions](#)
 - [Part 3.1: Accuracy and Mean Absolute Discrepancy](#)
 - [Part 3.2: Strassen Implementation](#)
- [Bibliography](#)

▼ Description

In this notebook we explored matrix mutiplication trying to find a more efficient and fast way to compute matrix-matrix product putting it to test with the numpy method (@). To continue we use Just In Time Compiler (JIT) from Numba to accelerate this process and to finalise we test Fortran and C-style of arrays to find the most optimal way to compute matrix-matrix product. Added to that we explored possible extensions of the assignment such as the strassen algorithm.

▼ Imports

```
import numpy as np
import matplotlib.pyplot as plt
from timeit import timeit
from numba import jit
from numba import njit
```

Slow Function

AB matrix product

```
def slow_matrix_product(mat1, mat2):
    """Multiply two matrices."""
    assert mat1.shape[1] == mat2.shape[0]
    result = []
    for c in range(mat2.shape[1]):
        column = []
        for r in range(mat1.shape[0]):
            value = 0
            for i in range(mat1.shape[1]):
                value += mat1[r, i] * mat2[i, c]
            column.append(value)
        result.append(column)
    return np.array(result).transpose()
```

matrices

```
matrix1 = np.random.rand(10, 10)
matrix2 = np.random.rand(10, 10)
```

#compare results function vs numpy

```
slow_result = slow_matrix_product(matrix1, matrix2)
true_result = matrix1 @ matrix2
```

if the matrices are not equal it will raise an error

```
if not np.allclose(slow_result, true_result, atol=1e-8):
    raise ValueError("The matrices do not match!")
```

Faster Function

```
# A better function
# Note as it will be seen later the first one is discarded later
def faster_matrix_product(mat1, mat2):
    """
    Multiply tow matrices:
    row of first matrix times column of second on ij of result matrix
    """
    assert mat1.shape[1] == mat2.shape[0]
    result = np.zeros((mat1.shape[0], mat2.shape[1]))

    for i in range(mat1.shape[0]):
        for j in range(mat2.shape[1]):
            result[i, j] = np.dot(mat1[i], mat2[:,j])

    return result
```

alternative #####33

```
def faster_matrix_product2(mat1, mat2):
    """
    Multiply two matrices matrix times vector multiplication
    """
    assert mat1.shape[1] == mat2.shape[0]
    result = np.empty((mat1.shape[1], mat2.shape[0]))
    for col in range(mat2.shape[1]):
        c=np.dot(mat1,mat2[:,col])
        result[:,col] = c
    return result
```

```
# models to test
fast_result = faster_matrix_product(matrix1, matrix2)
fast_result2 = faster_matrix_product2(matrix1, matrix2)
true_result = matrix1 @ matrix2
#print(f"-----,\nfast result: \n{fast_result},\n-----")
```

Test functions in different matrices sizes

```
def tests(iterations):
    """
    Test our two functions to see if they give the same result
    as the numpy implementation "@"
    """
    for n in iterations:
        # create matrices nxn size
        mat1 = np.random.rand(n, n)
        mat2 = np.random.rand(n, n)
        # compute results
        fast_result = faster_matrix_product(mat1, mat2)
        fast_result2 = faster_matrix_product2(mat1, mat2)
        true_result = mat1 @ mat2
        # assert statement if the reuslts are not similar enough it will raise an issue and specify in which dimension it fai
        assert np.allclose(fast_result, true_result, fast_result2), f"Failed for {n}x{n} matrix"
    print("END")
```

```
test_iterations = [2, 3, 4, 5] # Test matrices sizes
tests(test_iterations)
```

END

Comments on performance

- faster_matrix_product2 utilizes np.dot for whole-column multiplication, (benefiting from NumPy's optimized vectorized operations).
- Memory is swiftly allocated with np.empty, skipping initial value assignments.
- Multiplying entire columns at once improves memory access efficiency, given NumPy's column-major storage.
- Direct column assignments in the result are more efficient than list appends.
- Reduced reliance on Python loops minimizes inherent Python overhead.

Overall, these optimizations make faster_matrix_product2 outperform its counterparts.

▼ Part 1.1 Test and Visualize performance

```
import numpy as np
import timeit
from tqdm import tqdm
class MatrixTimer:
    def __init__(self, N, use_jit=False):
        self.N = N
        self.matrix1 = np.random.rand(self.N, self.N)
        self.matrix2 = np.random.rand(self.N, self.N)
        self.use_jit = use_jit

    def slow_mat(self):
        """Computes the matrix product using a slower method."""
        return slow_matrix_product(self.matrix1, self.matrix2)

    def faster_mat(self):
        """Computes the matrix product using a faster method."""
        return faster_matrix_product(self.matrix1, self.matrix2)

    def faster_mat_2(self):
        """Computes the matrix product using an alternative faster method."""
        if self.use_jit:
            return faster_matrix_product2_jit(self.matrix1, self.matrix2)
        else:
            return faster_matrix_product2(self.matrix1, self.matrix2)

    def numpy_mat(self):
        """Computes the matrix product using numpy's built-in method."""
        return self.matrix1 @ self.matrix2

    def time_operation(self, operation):
        """Times the specified matrix multiplication operation."""
        try:
            method_to_time = getattr(self, operation)
            return timeit.timeit(method_to_time, globals=globals(), number=1)
        except AttributeError:
            raise ValueError(f"Invalid operation: {operation} is not a valid method in MatrixTimer.")

# List of matrix sizes to benchmark
N_matrices = [2, 3, 5, 10, 30, 50, 70, 100, 200, 400, 500, 800]

# Dictionary to store results for each method and matrix size
results = {'slow_mat': [], 'faster_mat': [], 'faster_mat_2': []}

# Benchmark each matrix size for each method
for N in tqdm(N_matrices):
    benchmark = MatrixTimer(N)

    results['slow_mat'].append(benchmark.time_operation('slow_mat'))
    results['faster_mat'].append(benchmark.time_operation('faster_mat'))
    results['faster_mat_2'].append(benchmark.time_operation('faster_mat_2'))

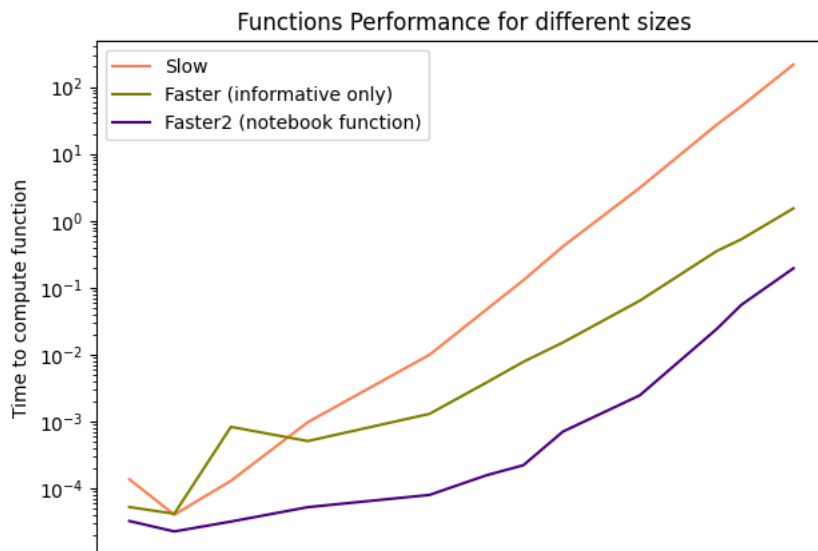
100%|██████████| 12/12 [05:01<00:00, 25.09s/it]

plt.plot(N_matrices, results['slow_mat'], color='coral', label='Slow')
plt.plot(N_matrices, results['faster_mat'], color='olive', label='Faster (informative only)')
plt.plot(N_matrices, results['faster_mat_2'], color='indigo', label='Faster2 (notebook function)')

plt.legend()

plt.xscale("log")
plt.yscale("log")

plt.xticks(N_matrices, N_matrices)
plt.xlabel("Matrix size")
plt.ylabel("Time to compute function")
plt.title(" Functions Performance for different sizes")
plt.tight_layout()
plt.show()
```



NOTICE THAT WE ARE PLOTTING THE 2 FASTER MATRICES FUNCTIONS BUT ONLY `faster_matrix_product_2` IS FURTHER USED, BUT THE OTHER IS ADDED ASWELL FOR A MORE INFORMATIVE PLOT

As expected, `faster_matrix_product2` outperforms the other methods. Interestingly, for small matrices, the `slow_matrix_product` function seems faster, but this advantage vanishes as matrix size increases. The superior performance of `faster_matrix_product2` can be attributed to its utilization of numpy functions. However, these functions have an initialization overhead, including memory setup and the establishment of optimized computation paths. For very small matrices, this overhead can take more time than the computation itself. However, as the matrix size grows, the benefits of these optimizations become apparent, making the `faster_matrix_product2` method more efficient.

▼ Part 2 JIT Implementation

```
# Implementation of JIT on function 2 since it shows better performance!
faster_matrix_product2_jit = njit()(faster_matrix_product2)

# Warm-up the JIT compilation
warmup = MatrixTimer(2, use_jit=True)
# Execute the JIT compiled function multiple times to ensure it's fully optimized
for _ in range(10):
    warmup.faster_mat_2()

# Dictionary to store time results
results_2 = {
    'faster_mat': [],
    'fast_mat_jit': [],
    'numpy_mat': [] }

# Benchmark each matrix multiplication method for each matrix size
for N in N_matrices:
    # Create benchmark objects for regular and JIT compiled functions
    benchmark_jit = MatrixTimer(N, use_jit=True)
    benchmark = MatrixTimer(N, use_jit=False)

    # Store the time taken for each method in the results dictionary
    results_2['faster_mat'].append(benchmark.time_operation('faster_mat_2'))
    results_2['fast_mat_jit'].append(benchmark_jit.time_operation('faster_mat_2'))
    results_2['numpy_mat'].append(benchmark.time_operation('numpy_mat'))
```

▼ Why warm up?

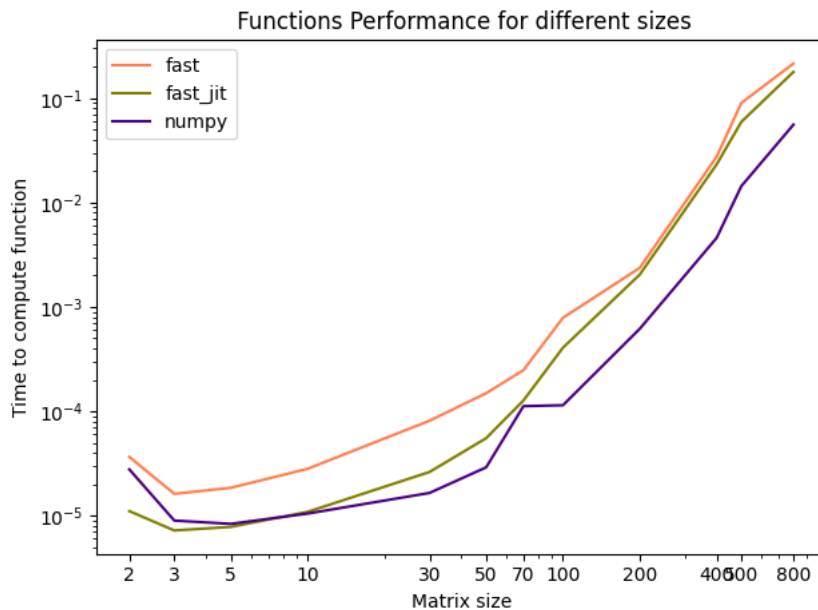
- It ensures the code is run enough times for the JIT compiler to identify and optimize hotspots.
- Without this, the initial runs would not be performing at its full, since it needs some runs to take effect (on extensions there is a better comparison of this).

```
plt.plot(N_matrices, results_2['faster_mat'], color='coral', label='fast')
plt.plot(N_matrices, results_2['fast_mat_jit'], color='olive', label='fast_jit')
plt.plot(N_matrices, results_2['numpy_mat'], color='indigo', label='numpy')

plt.legend()

plt.xscale("log")
plt.yscale("log")
```

```
plt.xticks(N_matrices, N_matrices)
plt.xlabel("Matrix size")
plt.ylabel("Time to compute function")
plt.title("Functions Performance for different sizes")
plt.tight_layout()
plt.show()
```



The function compiled with Just-In-Time (JIT) compilation `faster_matrix_product2_jit` shows better performance than its non-JIT version, especially for smaller matrices (this is also due to the previous warm-up in one of the extensions we will study how this affects JIT function performance). JIT compilation translates Python code into machine code right before execution. This makes the compiled code more efficient and can run faster.

The "warm-up" phase is relevant so JIT can perform. After JIT-compiled code is run a few times it ensures it is optimized. This gets rid of any overhead of compiling the code during the actual test, allowing the function to execute faster.

Numpy performs better than the other two methods for various reasons, built on an optimized C core allows more control over the computer's processors, enabling efficient low-level operations. Furthermore, Numpy makes use of special libraries such as BLAS for matrix operations (level 3) or matrix-vector (level2) operations. On top of that Numpy can perform multi-threading techniques to maximize computational power. This is dividing complex tasks (matrix-matrix multiplications in this case) into smaller sub-tasks and execute them simultaneously across various CPU cores (each one computes one thread, one sub-task) in parallel.

The reason for the spike in performance (between matrices of size 30-70) could be due to many factors. One explanation could be that NumPy might be using an algorithm that doesn't interact well with the hardware.

In conclusion while JIT offers an advantage compared to Non-JIT-function, it still can't compare to Numpy @function. Numpy is compiled and optimized (Use of C, allowing a more direct control of computer's processor) for an efficient memory usage and multi-threading, giving it a great advantage for high-complexity calculations, making use of BLAS specially tuned for these tasks.

▼ Part 2.1 Fortran/C-style

(For simplicity Fortran -> F and C-style -> C)

```
# Timing of "func"
def time_function(func, N):
    def wrapper(): # Calls "func" with N
        return func(N)
    return timeit.timeit(wrapper, number=5)

# Create Matrix NxN of C or F(ortran) style
def create_matrix(N, order='C'):
    if order == 'F':
        return np.asfortranarray(np.random.rand(N, N))
    return np.ascontiguousarray(np.random.rand(N, N))

# Possible scenarios described in the assignment (F-F, FC, C-C, C-F)
# Lambda explains what each scenario is
scenarios = {
    'FF': lambda N: faster_matrix_product2_jit(create_matrix(N, 'F'), create_matrix(N, 'F')),
    'FC': lambda N: faster_matrix_product2_jit(create_matrix(N, 'F'), create_matrix(N, 'C')),
    'CC': lambda N: faster_matrix_product2_jit(create_matrix(N, 'C'), create_matrix(N, 'C')),
    'CF': lambda N: faster_matrix_product2_jit(create_matrix(N, 'C'), create_matrix(N, 'F'))
}
```

```

'CF': lambda N: faster_matrix_product2_jit(create_matrix(N, 'C'), create_matrix(N, 'F')),
}

# Pre-allocate memory
times = {key: np.zeros(len(N_matrices)) for key in scenarios.keys()}


# Warm-up JIT
N = 3
for scenario, func in scenarios.items():
    time_function(func, N)

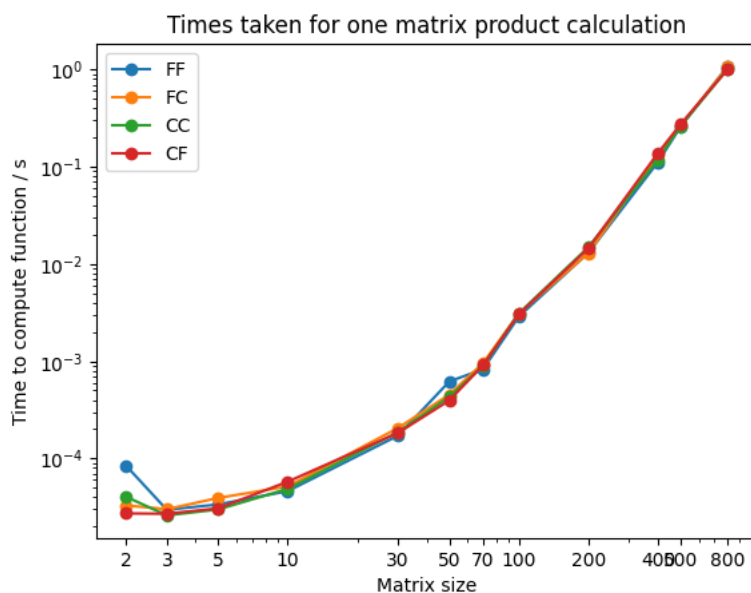
# Benchmark each case
for i, N in enumerate(N_matrices):
    for scenario, func in scenarios.items():
        times[scenario][i] = time_function(func, N)

for scenario, time_data in times.items():
    plt.plot(N_matrices, time_data, marker='o', label=scenario)

# Scaling
plt.xscale("log")
plt.yscale("log")
plt.xticks(N_matrices, N_matrices)
plt.xlabel("Matrix size")
plt.ylabel("Time to compute function / s")
plt.legend()
plt.title("Times taken for one matrix product calculation")
plt.show()

```

 <ipython-input-4-4e340cb63f23>:27: NumbaPerformanceWarning: np.dot() is faster than np.dot(mat1, mat2[:, col])



Theoretically we could expect FF (Fortran-Fortran) or CC to be the fastest because they use the same memory layout for a more efficient use of cache. However, we can see how CF seems to perform better, by a slight margin but still noticeable. Essentially we are doing

$$A \times B$$

This is result matrix C:

$$C_{ij} = \sum_{k=1}^n a_{ik} b_{kj},$$

The reason for what it could run faster CF style is the memory layouts.

- **C-Style (Row major)** In C-Style arrays, rows are contiguous in memory. For A trasnversing row A_i , means accesing contiguous memory locations (easier and faster access).

A =

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix}$$

- **Fortran-Style(Column major)** In Fortran-Style arrays, columns are contiguous in memory. For B , transversing column B_j means accessing contiguous memory locations (easier and faster access in this case since you iterate by columns).

$B =$

$$\begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m,1} & B_{m,2} & \cdots & B_{m,n} \end{pmatrix}$$

In CF (C-Fortran) scenario, the dot product of C_{ij} would involve a row from A and a column from B , both of which are contiguous in memory.

This improves efficiency because these two access operations are cache-friendly, and therefore reads faster. Another reason is that the SIMD (Multiple Data) allows one instruction to perform multiple operations, this is more efficient on contiguous data (performs in parallel) speeding up the calculations.

In conclusion CF layout aligns well with SIMD ability to perform multiple operations simultaneously and maximizing memory access.

Part 3: Extensions

▼ Part 3.1: Accuracy and Mean Absolute Discrepancy

```
mat_test1=[]
mat_test2=[]
validate_result = []
test_results = []

for n in range(1,100):
    mat_test1.append(np.random.rand(n,n))
    mat_test2.append(np.random.rand(n,n))

for (m1,m2) in zip(mat_test1, mat_test2):

    validate_result.append(m1 @ m2)
    test_results.append(faster_matrix_product2_jit(m1,m2))

tolerance = 1e-20
correct_predictions = sum(np.allclose(t, p, atol=tolerance) for t, p in zip(validate_result, test_results))
accuracy = correct_predictions / len(test_results)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Accuracy: 100.00%

However we knew this already, it will be studied accuracy in a more in depth way.

```

discrepancies = [test - valid for test, valid in zip(test_results, validate_result)]
threshold = 1e-20
# Assuming discrepancies is a list where each item is a matrix of discrepancies
mean_discrepancies = [np.mean(np.abs(d)) for d in discrepancies]

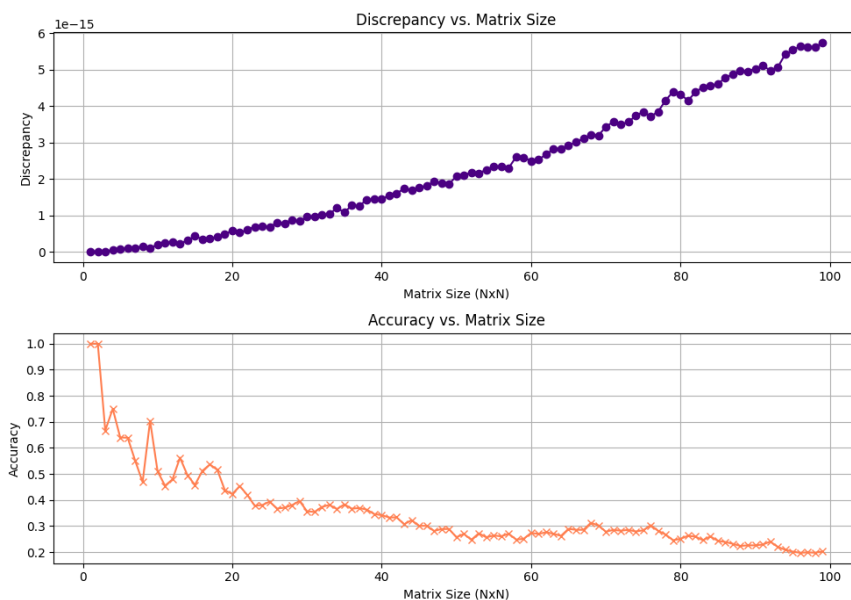
# Calculate accuracy for each matrix
accuracies = []
for discrepancy in discrepancies:
    correct_entries = np.sum(np.abs(discrepancy) < threshold)
    total_entries = discrepancy.size
    accuracy = correct_entries / total_entries
    accuracies.append(accuracy)

matrix_sizes = list(range(1, 100))

fig, ax = plt.subplots(2, 1, figsize=(10, 7))
ax[0].plot(matrix_sizes, mean_discrepancies, marker='o', color='indigo')
ax[0].set_xlabel("Matrix Size (NxN)")
ax[0].set_ylabel("Discrepancy")
ax[0].set_title("Discrepancy vs. Matrix Size")
ax[0].grid(True)
ax[1].plot(matrix_sizes, accuracies, marker='x', color='coral')
ax[1].set_xlabel("Matrix Size (NxN)")
ax[1].set_ylabel("Accuracy")
ax[1].set_title("Accuracy vs. Matrix Size")
ax[1].grid(True)

fig.tight_layout()
plt.show()

```



As it can be seen the accuracy reduces drastically with larger matrices, this was expected since it involves more complicated operations and the potential error accumulation. The more calculations, the chances of rounding errors and inaccuracies increase, leading to the decrease that we see.

In the case of the first plot Mean Absolute Discrepancy, the increase in error due to matrix size is because the number of calculations also increases, therefore higher chances of accumulating errors, explaining this trend of the results.

▼ Part 3.2: Strassen Algorithm Implementation

The Strassen Algorithm is a "*divide-and-conquer*" [1](#) strategy to multiply matrices. Given matrices A and B , each would be divided into four sub-matrices, such as:

A =

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

B =

$$\begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Seven multiplications (reducing from 8 to 7) are then performed using combinations of these sub-matrices, for example:

$$p1 = a \times (f - h) \text{ and } p5 = (a + d) \times (e + h)$$

These compute the four quadrants of the result matrix:

C =

$$\begin{pmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{pmatrix}$$

This process is then repeated in each multiplication, reducing the time complexity of the task from $O(n^3)$ to $O(n^{2.81})$. This could potentially compete with our previous functions.

Behind there is an implementation of the algorithm:

```
class StrassenAlgorithm:

    def split(self, matrix):
        row, col = matrix.shape
        row2, col2 = row // 2, col // 2
        return matrix[:row2, :col2], matrix[:row2, col2:], \
            matrix[row2:, :col2], matrix[row2:, col2:]

    def strassenMultiply(self, matrixA, matrixB):
        if len(matrixA) == 1:
            return matrixA * matrixB

        a, b, c, d = self.split(matrixA)
        e, f, g, h = self.split(matrixB)

        p1 = self.strassenMultiply(a, f - h)
        p2 = self.strassenMultiply(a + b, h)
        p3 = self.strassenMultiply(c + d, e)
        p4 = self.strassenMultiply(d, g - e)
        p5 = self.strassenMultiply(a + d, e + h)
        p6 = self.strassenMultiply(b - d, g + h)
        p7 = self.strassenMultiply(a - c, e + f)

        c11 = p5 + p4 - p2 + p6
        c12 = p1 + p2
        c21 = p3 + p4
        c22 = p1 + p5 - p3 - p7

        c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))
        return c

# Standalone function to multiply matrices of size N using Strassen's algorithm
def multiply_matrices_strassen(N):
    strassen_instance = StrassenAlgorithm()
    matrix1 = np.random.randint(100, 999, size=(N, N))
    matrix2 = np.random.randint(100, 999, size=(N, N))
    return strassen_instance.strassenMultiply(matrix1, matrix2)

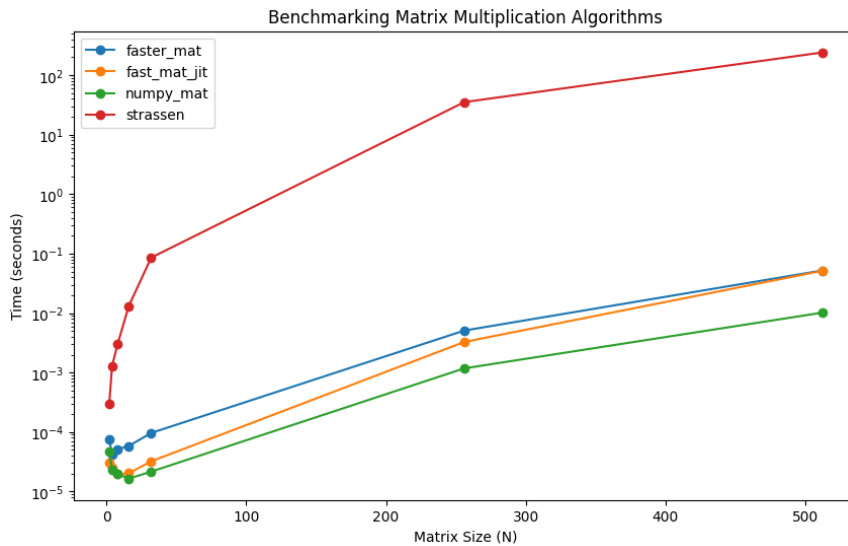
N_values = [2, 4, 8, 16, 32, 256, 512] # Only for matrices sizes that are powers of 2
results_3 = {
    'faster_mat': [],
    'fast_mat_jit': [],
    'numpy_mat': [],
    'strassen': []
}

for N in N_values:
    benchmark_jit = MatrixTimer(N, use_jit=True)
    benchmark = MatrixTimer(N, use_jit=False)
    results_3['faster_mat'].append(benchmark.time_operation('faster_mat_2'))
    results_3['fast_mat_jit'].append(benchmark_jit.time_operation('faster_mat_2'))
    results_3['numpy_mat'].append(benchmark_jit.time_operation('numpy_mat'))
    results_3['strassen'].append(timeit.timeit("multiply_matrices_strassen(N)", globals=globals(), number=1))
```

```
plt.figure(figsize=(10, 6))
plt.title("Benchmarking Matrix Multiplication Algorithms")
plt.xlabel("Matrix Size (N)")
plt.ylabel("Time (seconds)")
plt.yscale("log") # Log scale to better visualize differences

# Plot each algorithm's results
for label, times in results_3.items():
    plt.plot(N_values, times, label=label, marker='o')

plt.legend()
plt.show()
```



When comparing the Strassen Algorithm with traditional methods using JIT and Numpy operations, it can be expected the algorithm to perform better because of the reduce in time complexity theoretically. The main motivation for this extension is to explore wheter a theoretically faster algortihm could compete with the most recent optimizations and the practical aspects.

The outcomes were relatively unexpected, given that it performs much worse than the faster functions, although when considering the overhead of recursion and how the matrices are splitted, on top of that Strassen does not make use of JIT or BLAS libraries of any type. In conclusion this result makes sense and aligns with the research because when cosnidering more practical aspects like multi-threading, these