Slow Matrix product

```
import numpy as np


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()


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

print(slow_matrix_product(matrix1, matrix2))
print(matrix1 @ matrix2)
```

```
[[2.03821862 2.66453713 1.96305785 3.07339303 1.97886242 2.89688568
  3.75532708 2.58233472 3.47127385 1.8705086 ]
 [2.23523101 3.10341445 2.36408763 3.10800565 2.13213843 3.17779003
  3.794979   2.56029627 3.34359197 1.96839005]
 [2.69552649 3.1446682  1.9669168  4.05361401 2.39195991 3.41823403
  3.95103198 3.61177992 3.86385805 2.60033466]
 [2.35754005 3.22819055 2.07990172 3.46036058 1.59980504 3.57079777
  3.84947426 2.57805498 3.17380576 2.05374872]
 [2.20885077 2.91000619 2.0721386  2.85857481 1.83911632 2.95949471
  3.5409434  2.86971157 3.13864963 1.95943152]
 [2.04132721 2.26447229 1.83765007 3.01894079 1.98199882 2.62978236
  2.4038434  2.16339834 2.7210709  2.17143723]
 [2.64944276 3.58325733 2.41913374 3.50356197 1.93049852 3.88336589
  4.08671484 3.32271157 3.53595902 2.49639232]
 [1.82608954 1.97606685 1.15329579 2.66175665 1.66270761 2.54133884
  2.36479942 2.02420538 2.31862593 2.10577748]
 [2.36217515 3.58237214 2.31443748 3.17480857 2.12186536 3.25896792
  4.60708803 3.31233614 3.4274474  2.44188482]
 [2.13266215 2.85097342 1.52247341 2.91773152 1.62176113 3.15097896
  3.2903232  2.91414728 2.83546252 1.93389534]]
[[2.03821862 2.66453713 1.96305785 3.07339303 1.97886242 2.89688568
  3.75532708 2.58233472 3.47127385 1.8705086 ]
 [2.23523101 3.10341445 2.36408763 3.10800565 2.13213843 3.17779003
  3.794979   2.56029627 3.34359197 1.96839005]
 [2.69552649 3.1446682  1.9669168  4.05361401 2.39195991 3.41823403
  3.95103198 3.61177992 3.86385805 2.60033466]
 [2.35754005 3.22819055 2.07990172 3.46036058 1.59980504 3.57079777
  3.84947426 2.57805498 3.17380576 2.05374872]
 [2.20885077 2.91000619 2.0721386  2.85857481 1.83911632 2.95949471
  3.5409434  2.86971157 3.13864963 1.95943152]
 [2.04132721 2.26447229 1.83765007 3.01894079 1.98199882 2.62978236
  2.4038434  2.16339834 2.7210709  2.17143723]
 [2.64944276 3.58325733 2.41913374 3.50356197 1.93049852 3.88336589
  4.08671484 3.32271157 3.53595902 2.49639232]
 [1.82608954 1.97606685 1.15329579 2.66175665 1.66270761 2.54133884
  2.36479942 2.02420538 2.31862593 2.10577748]
 [2.36217515 3.58237214 2.31443748 3.17480857 2.12186536 3.25896792
  4.60708803 3.31233614 3.4274474  2.44188482]
 [2.13266215 2.85097342 1.52247341 2.91773152 1.62176113 3.15097896
  3.2903232  2.91414728 2.83546252 1.93389534]]
```

Part 1: A better function

1. This my own function for faster_matrix_product By using Numpy to calculate the dot prodcut of rows and columns. Because the dot product between two vectors is a scalar, this can avoid computing the full matrix-matrix product by using numpy.

```
import numpy as np
import timeit


# Define the faster_matrix_product function
```

```
def faster_matrix_product(mat1, mat2):

    assert mat1.shape[1] == mat2.shape[0]
    # don't use numpy for whole matrix
    result = np.empty((mat1.shape[0], mat2.shape[1]))
    for r in range(mat1.shape[0]):
        for c in range(mat2.shape[1]):
            result[r, c] = np.dot(mat1[r, :], mat2[:, c])
    return result
    # This return the result by using numpy for the whole matrix
    # return np.dot(mat1, mat2)
```

The improvement campares to "slow function"

- Vectorized computation: By using 'np.dot' to compute the dot product of rows and columns, my function use the numpy's efficient vectorized operations. And is in contrast to 'slow function' which compute in its manual loop and accumulation.
- Memory allocatiojn: In my own function I use 'np.empty' to pre-allocate the memory for the matrix. This could be more efficient than the appending approac in the 'slow function' especially for large matricse.

2. By using these code to check the correctness for my own matrix function and compares its results with the Numpy's matrix multiplication

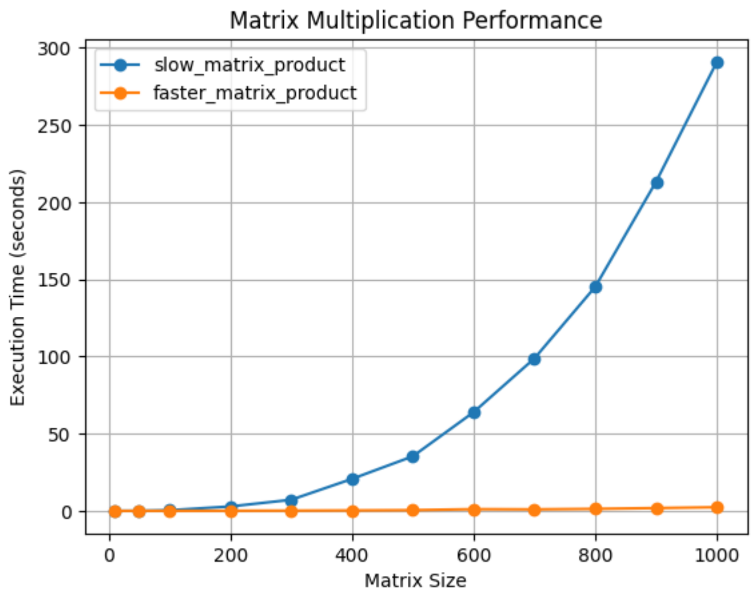
```
# Check correctness of faster_matrix_product
matrix_2x2 = np.random.rand(2, 2)
matrix_3x3 = np.random.rand(3, 3)
matrix_4x4 = np.random.rand(4, 4)
matrix_5x5 = np.random.rand(5, 5)
# The assert ensure the results are close to numpy's results.
assert np.allclose(faster_matrix_product(matrix_2x2, matrix_2x2), matrix_2x2 @ matrix_2x2)
assert np.allclose(faster_matrix_product(matrix_3x3, matrix_3x3), matrix_3x3 @ matrix_3x3)
assert np.allclose(faster_matrix_product(matrix_4x4, matrix_4x4), matrix_4x4 @ matrix_4x4)
assert np.allclose(faster_matrix_product(matrix_5x5, matrix_5x5), matrix_5x5 @ matrix_5x5)
# print("All tests passed!")
```

Compare performance of slow_matrix_product and faster_matrix_product

```
# select 12 matrix size from 10 to 1000 manually
matrix_sizes = [10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

# arrays for store the time for two function
slow_times = []
faster_times = []

for size in matrix_sizes:
    # generate two matrix for specific size
    matrix1 = np.random.rand(size, size)
    matrix2 = np.random.rand(size, size)


    slow_time = timeit.timeit(lambda: slow_matrix_product(matrix1, matrix2), number=1)
    faster_time = timeit.timeit(lambda: faster_matrix_product(matrix1, matrix2), number=1)

    slow_times.append(slow_time)
    faster_times.append(faster_time)
```

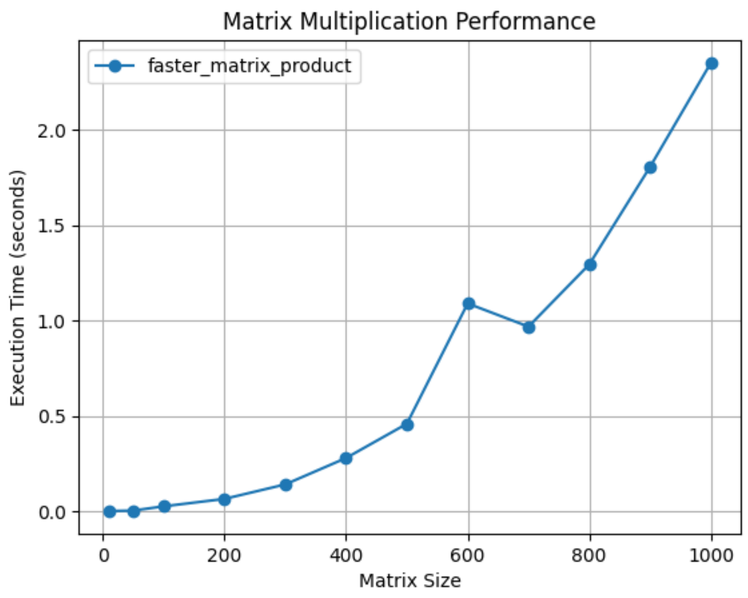
Create a plot to compare execution times

```
import matplotlib.pyplot as plt
# plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, slow_times, label='slow_matrix_product', marker='o')
plt.plot(matrix_sizes, faster_times, label='faster_matrix_product', marker='o')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.title('Matrix Multiplication Performance')
plt.grid(True)
plt.show()
```



By display the faster_matrix_product only, to see what is the exact result for large matrix

```
# plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, faster_times, label='faster_matrix_product', marker='o')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.title('Matrix Multiplication Performance')
plt.grid(True)
plt.show()
```



Summary for part1

From the results of plot, we can find that with the increase of matrix size, the execution time of the slow function increases exponentially. Although the faster function also increases, the change is very small compare to the slow function when the matrix is large.

Part 2: speeding it up with Numba

Fast function by using numba

using JIT to compile "faster_matrix_product" function

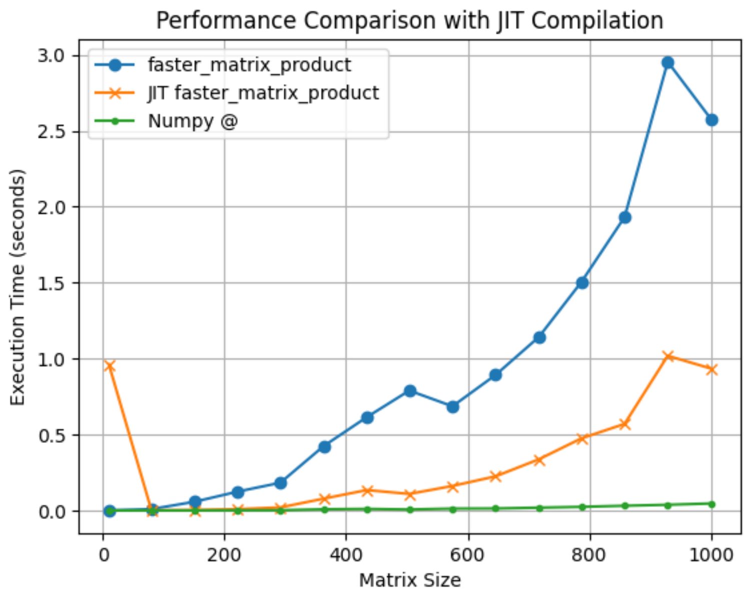
```
import numpy as np
import numba


@numba.jit(nopython=True)
def jit_faster_matrix_product(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]
    result = np.empty((mat1.shape[0], mat2.shape[1]))
    for r in range(mat1.shape[0]):
        for c in range(mat2.shape[1]):
            result[r, c] = np.dot(mat1[r, :], mat2[:, c])
    return result
```

using SIMD for more speed test

```
import numpy as np
import numba
from numba import prange


@numba.jit(nopython=True, parallel=True, fastmath=True)
def jit_faster_matrix_product(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]
    result = np.empty((mat1.shape[0], mat2.shape[1]))
    for r in prange(mat1.shape[0]):
        for c in prange(mat2.shape[1]):
            result[r, c] = np.dot(mat1[r, :], mat2[:, c])
    return result
```

```
import time
import matplotlib.pyplot as plt


matrix_sizes = np.linspace(10, 1000, 15, dtype=int)
times_original = []
times_jit = []
times_numpy = []

for size in matrix_sizes:
    mat1 = np.random.rand(size, size)
    mat2 = np.random.rand(size, size)

    start = time.time()
    _ = faster_matrix_product(mat1, mat2)
    times_original.append(time.time() - start)

    start = time.time()
    _ = jit_faster_matrix_product(mat1, mat2)
    times_jit.append(time.time() - start)

    start = time.time()
    _ = mat1 @ mat2
    times_numpy.append(time.time() - start)

# plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, times_original, label='faster_matrix_product', marker='o')
plt.plot(matrix_sizes, times_jit, label='JIT faster_matrix_product', marker='x')
plt.plot(matrix_sizes, times_numpy, label='Numpy @', marker='.')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.title('Performance Comparison with JIT Compilation')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-3-8be089ff4403>:12: NumbaPerformanceWarning: np.dot() is faste
  result[r, c] = np.dot(mat1[r, :], mat2[:, c])
```



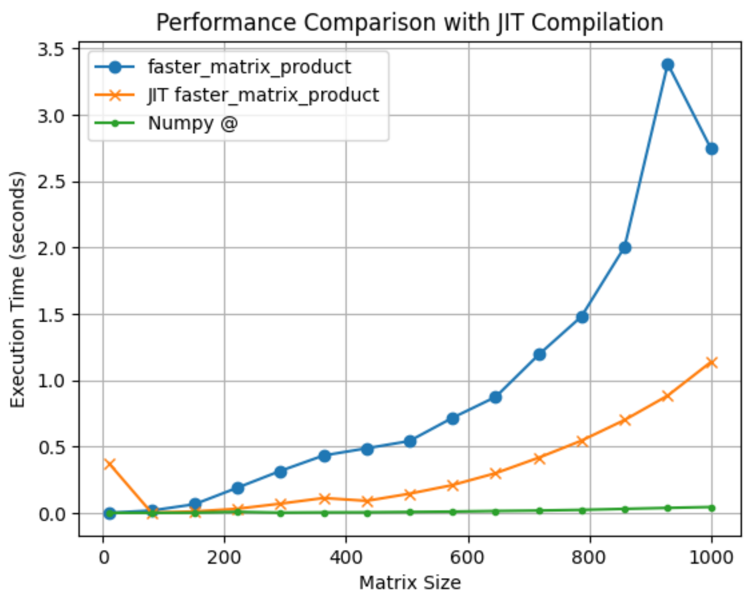
Summary for using SIMD in numba

In this task, I optimized matrix operations using SIMD with Numba, but there was no significant improvement observed. The possible reason could be that the dataset we used wasn't very large, and the benefits of SIMD performance may not be apparent with a single operation on a 1000x1000 matrix. As a result, I did not implement SIMD parallel processing in the final sta

Compare function 'JIT_fast', 'fast' and Numpy

```python
import time
import matplotlib.pyplot as plt


matrix_sizes = np.linspace(10, 1000, 15, dtype=int)
times_original = []
times_jit = []
times_numpy = []

for size in matrix_sizes:
    mat1 = np.random.rand(size, size)
    mat2 = np.random.rand(size, size)

    start = time.time()
    _ = faster_matrix_product(mat1, mat2)
    times_original.append(time.time() - start)

    start = time.time()
    _ = jit_faster_matrix_product(mat1, mat2)
    times_jit.append(time.time() - start)

    start = time.time()
    _ = mat1 @ mat2
    times_numpy.append(time.time() - start)

# plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, times_original, label='faster_matrix_product', marker='o')
plt.plot(matrix_sizes, times_jit, label='JIT faster_matrix_product', marker='x')
plt.plot(matrix_sizes, times_numpy, label='Numpy @', marker='.')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.title('Performance Comparison with JIT Compilation')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-9-6a00be421f34>:11: NumbaPerformanceWarning: np.dot() is faste
  result[r, c] = np.dot(mat1[r, :], mat2[:, c])
```



Summary for Numba:

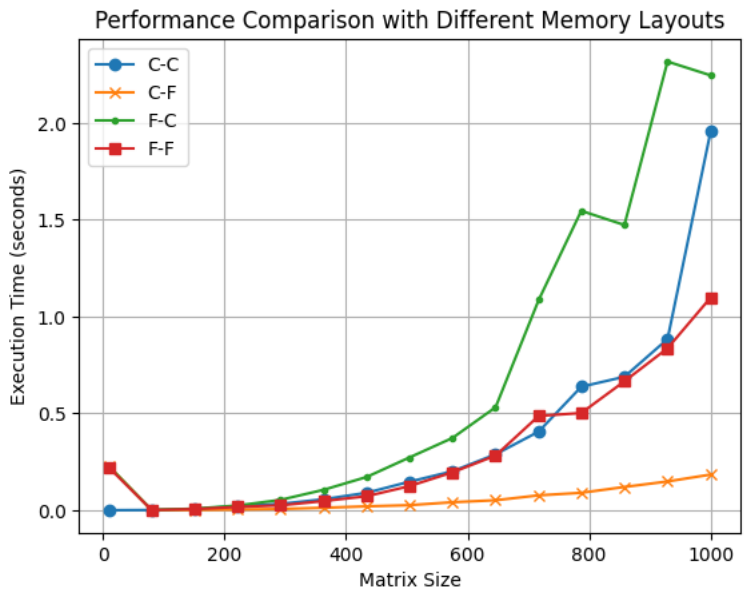
From the result we can see that the function which is using numba to acclerate is faster than "fast_matrix_function" but it still cann't be so effecient like numpy.

Further speed up

In this part we combined C-style and Fortan-style by different order on the two matrix which is being compute. and compare the speed for thess combination.

```
times_CC = []  # Both matrices in C-style ordering
times_CF = []  # First matrix in C-style and second in Fortran-style ordering
times_FC = []  # First matrix in Fortran-style and second in C-style ordering
times_FF = []  # Both matrices in Fortran-style ordering

for size in matrix_sizes:
    # init the 2 matrix by two different styles
    C_mat1 = np.random.rand(size, size)
    C_mat2 = np.random.rand(size, size)
    F_mat1 = np.asfortranarray(C_mat1)
    F_mat2 = np.asfortranarray(C_mat2)

    start = time.time()
    _ = jit_faster_matrix_product(C_mat1, C_mat2)
    times_CC.append(time.time() - start)

    start = time.time()
    _ = jit_faster_matrix_product(C_mat1, F_mat2)
    times_CF.append(time.time() - start)

    start = time.time()
    _ = jit_faster_matrix_product(F_mat1, C_mat2)
    times_FC.append(time.time() - start)

    start = time.time()
    _ = jit_faster_matrix_product(F_mat1, F_mat2)
    times_FF.append(time.time() - start)

# plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, times_CC, label='C-C', marker='o')
plt.plot(matrix_sizes, times_CF, label='C-F', marker='x')
plt.plot(matrix_sizes, times_FC, label='F-C', marker='.')
plt.plot(matrix_sizes, times_FF, label='F-F', marker='s')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.title('Performance Comparison with Different Memory Layouts')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-9-6a00be421f34>:11: NumbaPerformanceWarning: np.dot() is faste
  result[r, c] = np.dot(mat1[r, :], mat2[:, c])
<ipython-input-9-6a00be421f34>:11: NumbaPerformanceWarning: np.dot() is faste
  result[r, c] = np.dot(mat1[r, :], mat2[:, c])
```



Summary for part2

- By analysing the result from the result, we can see that the 'C-F' layout which first matrix is c-style and second is Fortran-style ordering, is the fastest performance across all the matrix size.
- The rest of combination is slower than the 'C-F', and the 'F-F'layouts after 800 there is a significant difference from the other three
- The reason for this result:
  - In my opinion, I think there are two reason result in 'C-F'is the best performance one. First is **contigous memory access**, that is because when the first matrix is in C-style ordering, accessing its rows becomes contiguous in memory. Similarly, when the second matrix uses Fortran-style ordering, accessing its columns is contiguous. This make sure that both row and column accesses benefit from contiguous memory blocks.
  - The second reason I think it may be **cache locality**, When data is accessed in contiguous blocks it improves spatial locality. And the 'C-F' combination maximizes this spatial locality benefit for both matrices.
- In contrast, that expalined the reason why 'F-F' is the poorest performance. The larger matrix sizes, results in non-contuguous memory accesses for the rows of the first matrix, which lead to frequent cache misses and increased execution times.