Assignment 1 - Matrix-Matrix multiplication

October 20, 2022

1 Assignment 1 - Matrix-matrix multiplication

Assignment link: https://tbetcke.github.io/hpc_lecture_notes/2022-assignment_1.html

In this notebook, we investigate the speed of matrix-matrix multiplication. We start out with an example of a slow function to multiple two matrices, and attempt to write a faster function for it. We then write a script to compare the speed of execution, before turning to Numba for further optimization.

[1]:
```python
# imports

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

1.1 1. A better function

[3]:
```python
# slow function given in the assignment link

def slow_matrix_product(mat1, mat2):
    """Multiply two matrices (slow)."""
    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()
```

1.1.1 1. (a) Write your own function called faster_matrix_product that computes the product of two matrices more efficiently than slow_matrix_product.

[4]:
```
# Writing a faster function for matrix multiplication

def faster_matrix_product(mat1, mat2):
    """Multiply two matrices (faster)."""
    assert mat1.shape[1] == mat2.shape[0]

    product = np.zeros((mat1.shape[0], mat2.shape[1]))

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

    return product


def np_matrix_product(mat1, mat2):
    """Multiply two matrices together using Numpy"""
    assert mat1.shape[1] == mat2.shape[0]

    return mat1 @ mat2
```

1.1.2 1. (b) Write a Python script using an assert statement that checks that your function gives the same result as using @ for random 2 by 2, 3 by 3, 4 by 4, and 5 by 5 matrices.

[4]:
```
# Result Output Check

mat_sizes = [2, 3, 4, 5]

for size in mat_sizes:
    # generating random matrices
    matrix1 = np.random.rand(size, size)
    matrix2 = np.random.rand(size, size)

    # run functions
    faster_product = faster_matrix_product(matrix1, matrix2)
    np_product = np_matrix_product(matrix1, matrix2)

    # compare results
    assert np.allclose(faster_product, np_product)
```

1.1.3 1. (c) Give two brief reasons (1-2 sentences for each) why your function is better than slow_matrix_product.

1. The faster function contains 1 less nested loop than the slow one (2 instead of 3); it uses Numpy's vector multiplication and sum instead. The slow function is therefore expected to run in ~O(n^3) time complexity while the faster one between O(n^2) and O(n^3), where n is the side length of the input matrices.
2. In the faster function, memory allocation is done just once before the loops (the variable "product" holds a Numpy array whose shape is defined then and once); whereas in the slower one, memory must be allocated whenever a new element is appended to a list ("column.append(…)" & "result.append(…)". This occurs n^2 times instead of once (where n is the size length of the input matrices).

1.1.4 1. (d) Write a Python script that runs the two functions for matrices of a range of sizes, and use matplotlib to create a plot showing the time taken for different sized matrices for both functions.

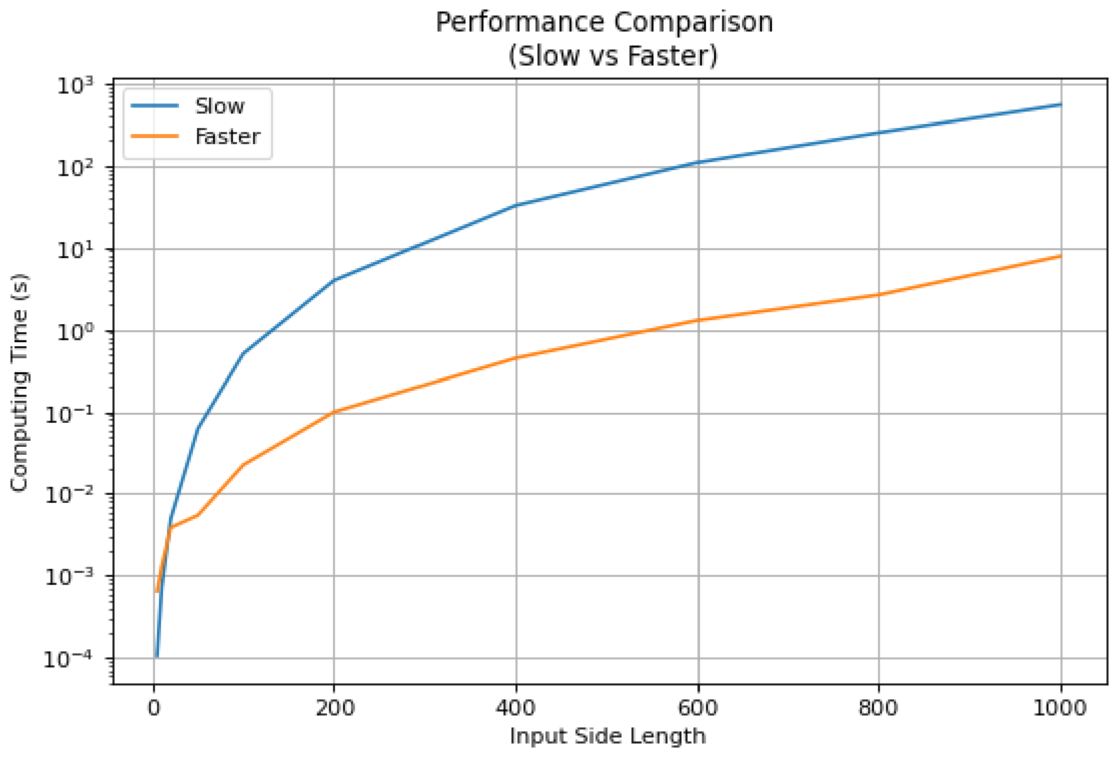
[5]:
```
def how_long(fun, args):
    """
    times how long the function 'fun' takes to execute
    with arguments 'args'
    """
    start = time.time()
    fun(*args)
    end = time.time()

    return end - start
```

[6]:
```
# Investigating the time efficiency of faster_matrix_product

mat_sizes = [5, 10, 20, 50, 100, 200, 400, 600, 800, 1000]
times_taken = []

for size in mat_sizes:

    print("Testing for matrices of size length", size)

    # generating random matrices
    input_matrices = [np.random.rand(size, size), np.random.rand(size, size)]

    # timing functions
    slow_time = how_long(slow_matrix_product, input_matrices)
    faster_time = how_long(faster_matrix_product, input_matrices)

    # storing result
    times_taken.append([size, slow_time, faster_time])
```

```
# Transforming to Numpy array for cleaner access of nested list elements
times_taken = np.array(times_taken)
```

```
Testing for matrices of size length 5
Testing for matrices of size length 10
Testing for matrices of size length 20
Testing for matrices of size length 50
Testing for matrices of size length 100
Testing for matrices of size length 200
Testing for matrices of size length 400
Testing for matrices of size length 600
Testing for matrices of size length 800
Testing for matrices of size length 1000
```

[7]:
```
# Plot comparison of performances
plt.figure(figsize=(8, 5), dpi=80)
plt.plot(times_taken[:, 0], times_taken[:, 1], label="Slow")
plt.plot(times_taken[:, 0], times_taken[:, 2], label="Faster")
plt.xlabel("Input Side Length")
plt.ylabel("Computing Time (s)")
plt.title("Performance Comparison \n (Slow vs Faster)")
plt.yscale('log')
plt.grid()
plt.legend()
plt.show()
```



1.2 2. Speeding it up with Numba

[8]:
```
from numba import njit
```

1.2.1 2. (a) Create a copy of your function faster_matrix_product that is just-in-time (JIT) compiled using Numba.

[9]:
```
@njit
def numba_matrix_product(mat1, mat2):
    """Multiply two matrices together using Numba's JIT compilation"""
    assert mat1.shape[1] == mat2.shape[0]

    product = np.zeros((mat1.shape[0], mat2.shape[1]))

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

    return product
```

1.2.2 2. (b) Make a plot (similar to that you made in the first part) that shows the times taken to multiply matrices using faster_matrix_product, faster_matrix_product with Numba JIT compilation, and Numpy (@).

[10]:
```
# Running Numba optimized function once to compile it pre-speed testing
numba_matrix_product(np.random.rand(2, 2), np.random.rand(2, 2))
```

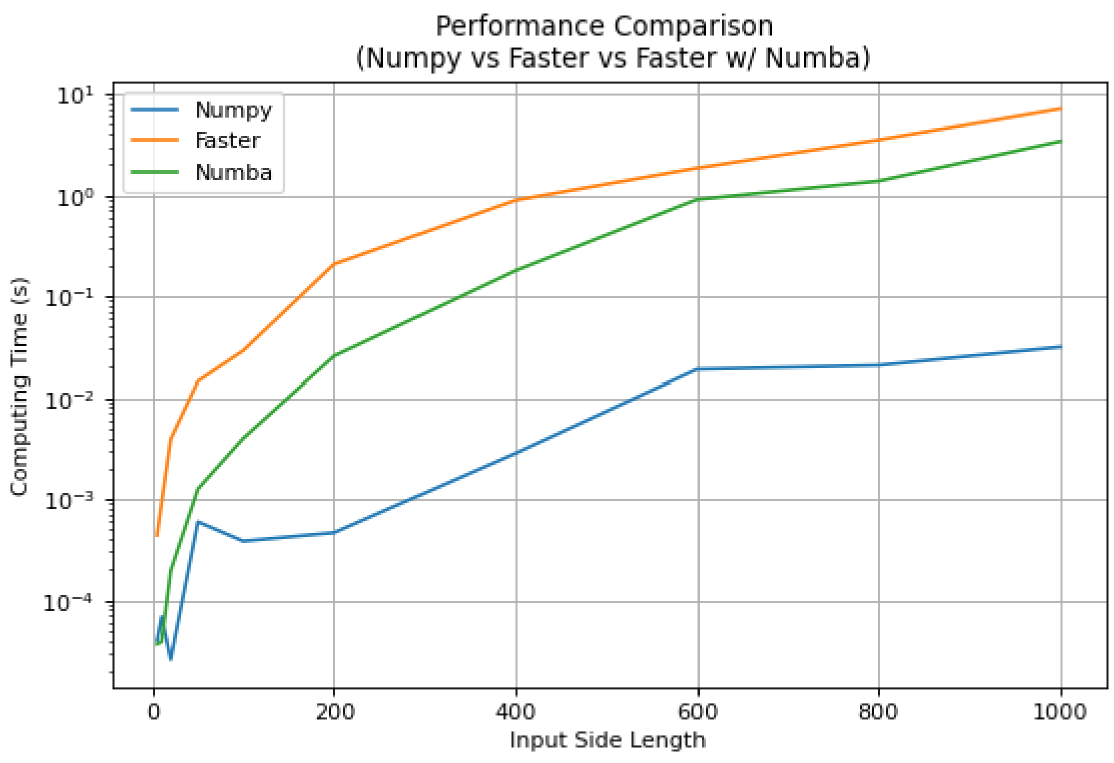
```
/var/folders/kf/pn8vnxyn0ml0wl398c9x4xq00000gn/T/ipykernel_60807/2942898090.py:1
0: NumbaPerformanceWarning: np.dot() is faster on contiguous arrays,

called on (array(float64, 1d, C), array(float64, 1d, A))
  product[r, i] = np.dot(mat1[r, :], mat2[:, i])
```

[10]:
```
array([[0.74639213, 0.3867548 ],
       [0.51726393, 0.34704819]])
```

[11]:
```
# Investigating the time efficiency of faster_matrix_product
mat_sizes = [5, 10, 20, 50, 100, 200, 400, 600, 800, 1000]

times_taken = []

for size in mat_sizes:
    print("Testing for matrices of size", size)

    # generating random matrices
    input_matrices = [np.random.rand(size, size), np.random.rand(size, size)]

    # timing functions
    np_time = how_long(np_matrix_product, input_matrices)
    faster_time = how_long(faster_matrix_product, input_matrices)
    numba_time = how_long(numba_matrix_product, input_matrices)

    # storing result
    times_taken.append([size, np_time, faster_time, numba_time])


# Transforming to Numpy array for cleaner access of nested list elements
times_taken = np.array(times_taken)
```

```
Testing for matrices of size 5
Testing for matrices of size 10
Testing for matrices of size 20
Testing for matrices of size 50
Testing for matrices of size 100
Testing for matrices of size 200
Testing for matrices of size 400
Testing for matrices of size 600
Testing for matrices of size 800
Testing for matrices of size 1000
```

[12]:
```
# Plot comparison of performances
plt.figure(figsize=(8, 5), dpi=80)
plt.plot(times_taken[:, 0], times_taken[:, 1], label="Numpy")
plt.plot(times_taken[:, 0], times_taken[:, 2], label="Faster")
plt.plot(times_taken[:, 0], times_taken[:, 3], label="Numba")
plt.xlabel("Input Side Length")
plt.ylabel("Computing Time (s)")
plt.title("Performance Comparison \n (Numpy vs Faster vs Faster w/ Numba)")
plt.yscale("log")
plt.legend()
plt.grid()
plt.show()
```



1.2.3 2. (c) Make a plot that compares the times taken by your JIT compiled function when the inputs have different combinations of C-style and Fortran-style ordering.

```
[13]: # Running Numba optimized function once to compile it pre-speed testing
m = np.random.rand(2, 2)
numba_matrix_product(m, np.asfortranarray(m))
numba_matrix_product(np.asfortranarray(m), m)
numba_matrix_product(np.asfortranarray(m), np.asfortranarray(m))
```

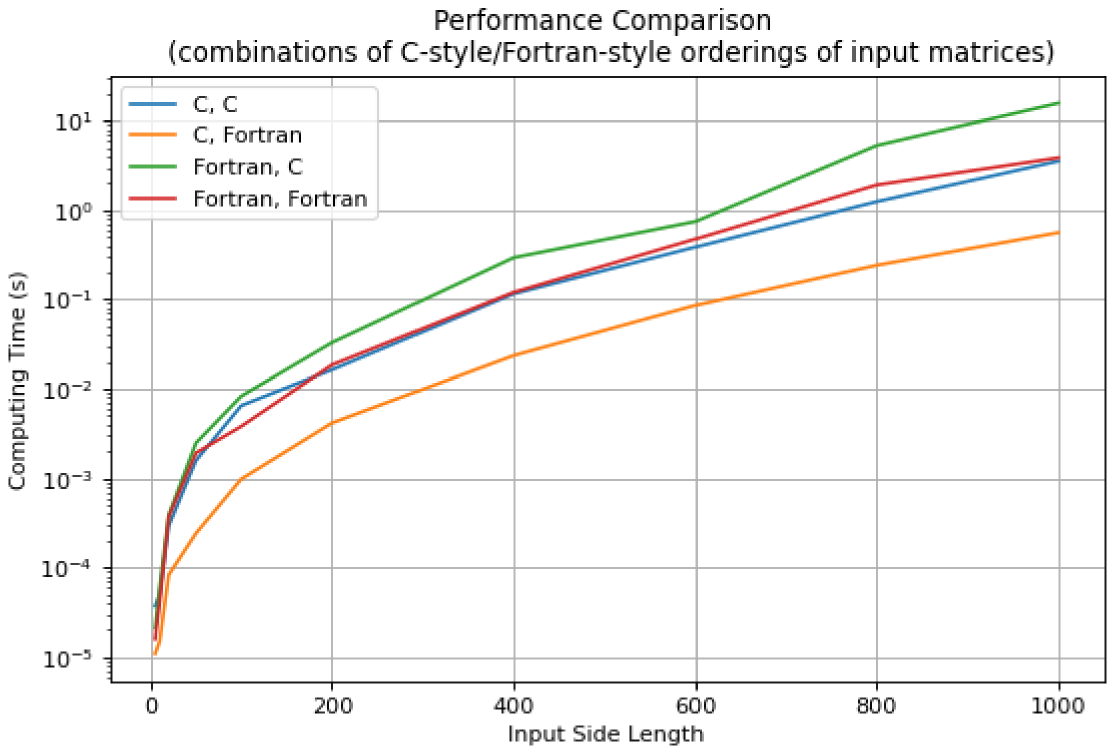
```
/var/folders/kf/pn8vnxyn0ml0wl398c9x4xq00000gn/T/ipykernel_60807/2942898090.py:1
0: NumbaPerformanceWarning: np.dot() is faster on contiguous arrays,

called on (array(float64, 1d, A), array(float64, 1d, A))
  product[r, i] = np.dot(mat1[r, :], mat2[:, i])
/var/folders/kf/pn8vnxyn0ml0wl398c9x4xq00000gn/T/ipykernel_60807/2942898090.py:1
0: NumbaPerformanceWarning: np.dot() is faster on contiguous arrays,

called on (array(float64, 1d, A), array(float64, 1d, F))
  product[r, i] = np.dot(mat1[r, :], mat2[:, i])
```

```
[13]: array([[0.46049574, 0.51695758],
             [0.33897476, 0.39388798]])
```

[14]:
```python
# Investigating the time efficiency of faster_matrix_product
mat_sizes = [5, 10, 20, 50, 100, 200, 400, 600, 800, 1000]

times_taken = []

for size in mat_sizes:
    print("Testing for matrices of size", size)

    # creating inputs of same values but different C-style and Fortran-style␣
 ↪orderings
    cc_input = [np.random.rand(size, size), np.random.rand(size, size)]
    cf_input = [cc_input[0], np.asfortranarray(cc_input[1])]
    fc_input = [np.asfortranarray(cc_input[0]), cc_input[1]]
    ff_input = [np.asfortranarray(cc_input[0]), np.asfortranarray(cc_input[1])]

    # timing functions
    cc_time = how_long(numba_matrix_product, cc_input)
    cf_time = how_long(numba_matrix_product, cf_input)
    fc_time = how_long(numba_matrix_product, fc_input)
    ff_time = how_long(numba_matrix_product, ff_input)

    # storing result
    times_taken.append([size, cc_time, cf_time, fc_time, ff_time])


# Transforming to Numpy array for cleaner access of nested list elements
times_taken = np.array(times_taken)
```

```
Testing for matrices of size 5
Testing for matrices of size 10
Testing for matrices of size 20
Testing for matrices of size 50
Testing for matrices of size 100
Testing for matrices of size 200
Testing for matrices of size 400
Testing for matrices of size 600
Testing for matrices of size 800
Testing for matrices of size 1000
```

[18]:
```python
# Plot comparison of performances
plt.figure(figsize=(8, 5), dpi=80)
plt.plot(times_taken[:, 0], times_taken[:, 1], label="C, C")
plt.plot(times_taken[:, 0], times_taken[:, 2], label="C, Fortran")
plt.plot(times_taken[:, 0], times_taken[:, 3], label="Fortran, C")
plt.plot(times_taken[:, 0], times_taken[:, 4], label="Fortran, Fortran")
plt.xlabel("Input Side Length")
plt.ylabel("Computing Time (s)")
plt.title("Performance Comparison \n (combinations of C-style/Fortran-style␣
 ↪orderings of input matrices)")
plt.yscale("log")
plt.legend()
plt.grid()
plt.show()
```



1.2.4 2. (d) Comment (in 1-2 sentences) on why one of these orderings appears to be fastest that the others.

In matrix multiplication, we are accessing the 1st matrix elements row by row and the 2nd matrix elements column by column. Hence, the optimal ordering has the 1st matrix elements stored in C-style (row by row) and the 2nd in Fortran-style (column by column) to minimize the distance between the bits accessed in memory. This explains why the "CF" function is fastest, while the "FC" function (which has both matrices stored in the wrong ordering) is slowest.

```
[ ]:
```