Mnożenie macierzy

March 12, 2023

Grzegorz Legęza, Wojciech Ciężobka, Mykola Haltiuk

1 Algorytm klasyczny

Algorytm klasyczny polega na przejściu w pętlach po macierzach w celu wymnożenia i dodania odpowiednich liczb.

[158]:
```python
def multiply_classic(A: Matrix, B: Matrix):
    """
    Multiplies `A` times `B` with a classic algorithm, where
    `A` is an `m x n` matrix and `B` is an `n x l` matrix.
    """

    m, n, l = A.shape[0], A.shape[1], B.shape[1]
    multiply = np.empty((m, l))
    sum = 0
    for i in range(m):                      # rows in multiply
        for j in range(l):                  # columns in multiply
            for k in range(n):              # columns in A and rows in B
                sum += A[i, k] * B[k, j]
            multiply[i, j] = sum
            sum = 0

    return multiply
```

Obliczenie liczby operacji zmiennoprzecinkowych. W dolnym indeksie zapisano typ działania: `add` - dodawanie, `mul` - mnożenie.

Wejście: A - m x n, B - n x l
$FLO_{classic}(m, n, l) =$
$= \sum_{i=0}^{m} \sum_{j=0}^{l} \sum_{k=0}^{n} (1_{add} + 1_{mul}) =$
$= 1_{add}mnl + 1_{mul}mnl =$
$= 2mnl$
W szczególności dla macierzy kwadratowych n x n gdzie $n = 2^k$:
$FLO_{classic}(k) =$
$= 1_{add}n^3 + 1_{mul}n^3 =$
$= 1_{add}8^k + 1_{mul}8^k =$
$= 2 * 8^k$

2 Algorytm Strassena

Algorytm Strassena działa rekurencyjnie, dzieli macierz kwadratową na 4 podmacierze o równym rozmiarze. Na tych macierzach zostają wykonane pewne operacje, włącznie z wywołaniem rekurencyjnym mnożenia. Wyniki tych działań podlegają konkatenacji z powrotem do macierzy w rozmiarze wejściowym.

[159]:
```python
def multiply_strassen(A: Matrix, B: Matrix):
    """
    Multiplies `A` times `B` with a Strassen algorithm, where
    both `A` and `B` are square `n x n` matrices.
    """

    def strassen(A: Matrix, B: Matrix, n: int):

        if n == 1:
            return A * B

        m = n // 2

        A11 = A[:m, :m]
        A12 = A[:m, m:]
        A21 = A[m:, :m]
        A22 = A[m:, m:]

        B11 = B[:m, :m]
        B12 = B[:m, m:]
        B21 = B[m:, :m]
        B22 = B[m:, m:]

        P1 = strassen(A11 + A22, B11 + B22, m)
        P2 = strassen(A21 + A22, B11, m)
        P3 = strassen(A11, B12 - B22, m)
        P4 = strassen(A22, B21 - B11, m)
        P5 = strassen(A11 + A12, B22, m)
        P6 = strassen(A21 - A11, B11 + B12, m)
        P7 = strassen(A12 - A22, B21 + B22, m)

        C = np.concatenate((
            np.concatenate((P1 + P4 - P5 + P7, P3 + P5), axis=1),
            np.concatenate((P2 + P4, P1 - P2 + P3 + P6), axis=1)
        ), axis=0)

        return C

    return strassen(A, B, A.shape[0])
```

Obliczenie liczby operacji zmiennoprzecinkowych. Przy zmianie szeregu na wzór skorzystano z narzędzia Wolfram Alpha.

Wejście: A - n x n, B - n x n and $n = 2^k$ => $k = \log_2 n$
$FLO_{Strassen}(1) = 1_{mul}$
$FLO_{Strassen}(2^k) =$
$= (2_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (1_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (1_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (1_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (1_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (2_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + (2_{add} * 2^{k-1} + FLO_{Strassen}(2^{k-1})) + 8_{add} * 2^{k-1} =$
$= 18_{add} * 2^{k-1} + 7 * FLO_{Strassen}(2^{k-1}) =$
$= 18_{add} * 2^{k-1} + 7 * (18_{add} * 2^{k-2} + 7 * FLO_{Strassen}(2^{k-2})) =$
$= 7^0 * 18_{add} * 2^{k-1} + 7 * 18_{add} * 2^{k-2} + 7^2 * 18_{add} * 2^{k-3} + 7^3 * 18_{add} * 2^{k-4} + ... + 7^{k-1} * 18_{add} * 2^{k-k} + 7^k * 1_{mul} =$
$= \sum_{i=0}^{k-1} (7^i * 18_{add} * 2^{k-i-1}) + 7^k * 1_{mul} =$
$= -\frac{18}{5}(2^k - 7^k) * 1_{add} + 7^k * 1_{mul} =$
$= \frac{23}{5} * 7^k - \frac{18}{5} * 2^k =$
$= \frac{23}{5} 7^{\log_2 n} - \frac{18}{5} n$

Warto porównać liczbę mnożeń z klasycznym algorytmen. W algorytmie Strassena mamy 7^k operacji mnożenia, natomiast w klasycznym będzie to 8^k. Widać, że algorytm Strassena ma znacznie mniej mnożeń, które są najbardziej kosztowne dla komputera.

3 Połączone mnożenie za pomocą algorytmu Strassena i klasycznego

Algorytm do mnożenia wykorzystuje algorytm Strassena, jeśli macierz ma rozmiar (jeden z wymiarów) większy od zadanego parametru `size_classic`. W szczególności gdy macierz w wywołaniu rekurencyjnym jest mniejsza niż ten parametr wykonujemy klasyczne mnożenie, zamiast kolejnego mnożenia algorytmem Strassena.

[163]:
```
def multiply_strassen_with_classic(A: Matrix, B: Matrix, size_classic: int = 1):
    """
    Multiplies `A` times `B` with a Strassen algorithm, where
    both `A` and `B` are square `n x n` matrices.
    If submatrix size is less or equal than `size_classic` uses␣
 ↪`multiply_classic`.
    """

    def strassen(A: Matrix, B: Matrix, n: int):

        if n == 1:
            return A * B
        elif n <= size_classic:
            return multiply_classic(A, B)

        m = n // 2

        A11 = A[:m, :m]
        A12 = A[:m, m:]
        A21 = A[m:, :m]
```

```
        A22 = A[m:, m:]

        B11 = B[:m, :m]
        B12 = B[:m, m:]
        B21 = B[m:, :m]
        B22 = B[m:, m:]

        P1 = strassen(A11 + A22, B11 + B22, m)
        P2 = strassen(A21 + A22, B11, m)
        P3 = strassen(A11, B12 - B22, m)
        P4 = strassen(A22, B21 - B11, m)
        P5 = strassen(A11 + A12, B22, m)
        P6 = strassen(A21 - A11, B11 + B12, m)
        P7 = strassen(A12 - A22, B21 + B22, m)

        C = np.concatenate((
            np.concatenate((P1 + P4 - P5 + P7, P3 + P5), axis=1),
            np.concatenate((P2 + P4, P1 - P2 + P3 + P6), axis=1)
        ), axis=0)

        return C

    if A.shape[0] <= size_classic:
        return multiply_classic(A, B)

    return strassen(A, B, A.shape[0])
```

4 Testy algorytmów

Podczas testów wykorzystano macierze 4x4. Jako wzorzec przyjęto wynik mnożenia policzony z pomocą biblioteki `numpy`.

[161]:
```
test_multiplication(multiply_classic)
```

```
=== Testing multiply_classic algorithm ===

Matrix A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

Matrix B:
[[17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]
 [29 30 31 32]]
```

```
Matrix AxB:
[[ 250.  260.  270.  280.]
 [ 618.  644.  670.  696.]
 [ 986. 1028. 1070. 1112.]
 [1354. 1412. 1470. 1528.]]

Matrix AxB (numpy):
[[ 250  260  270  280]
 [ 618  644  670  696]
 [ 986 1028 1070 1112]
 [1354 1412 1470 1528]]

Test Passed!
```

[162]:
```python
test_multiplication(multiply_strassen)
```

```
=== Testing multiply_strassen algorithm ===

Matrix A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

Matrix B:
[[17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]
 [29 30 31 32]]

Matrix AxB:
[[ 250  260  270  280]
 [ 618  644  670  696]
 [ 986 1028 1070 1112]
 [1354 1412 1470 1528]]

Matrix AxB (numpy):
[[ 250  260  270  280]
 [ 618  644  670  696]
 [ 986 1028 1070 1112]
 [1354 1412 1470 1528]]

Test Passed!
```

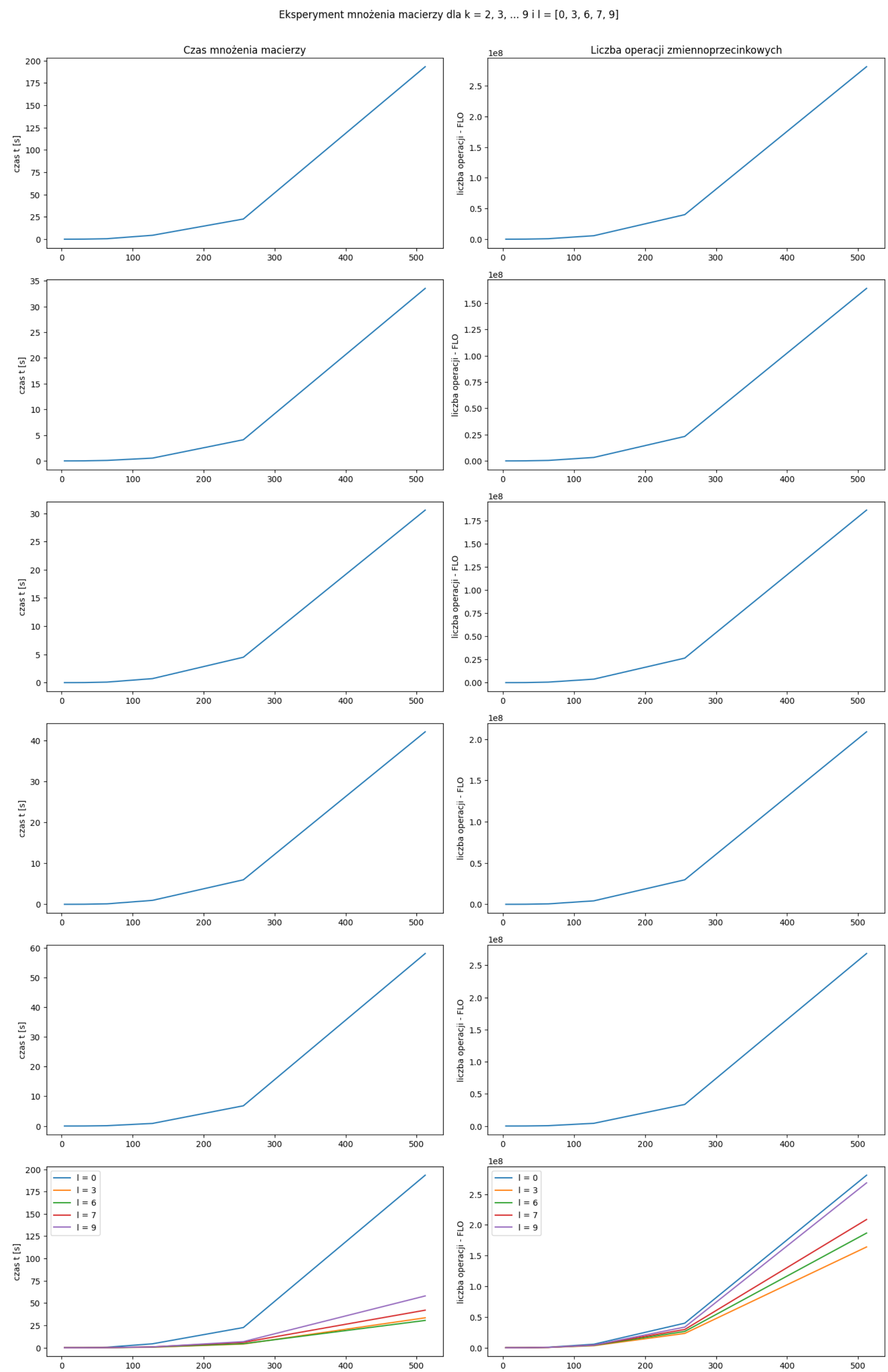
[164]:
```python
test_multiplication(multiply_strassen_with_classic, size_classic=2)
```

```
=== Testing multiply_strassen_with_classic algorithm ===
```

```
Matrix A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

Matrix B:
[[17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]
 [29 30 31 32]]

Matrix AxB:
[[ 250.  260.  270.  280.]
 [ 618.  644.  670.  696.]
 [ 986. 1028. 1070. 1112.]
 [1354. 1412. 1470. 1528.]]

Matrix AxB (numpy):
[[ 250  260  270  280]
 [ 618  644  670  696]
 [ 986 1028 1070 1112]
 [1354 1412 1470 1528]]

Test Passed!
```

5 Eksperymenty

Zostały przeprowadzone eksperymenty dla różnych wartości k i l. Jeden eksperyment polegał na wygenerowaniu dwóch macierzy o rozmiarze 2^k x 2^k i wymnożeniu je z parametrem `size_classic` wynoszącym 2^l. Dla każdego eksperymentu policzono również liczbę operacji zmiennoprzecinkowych. Wyniki zostały zebrane i przedstawione na poniższej grafice. Każdy rząd reprezentuje jedną wartość prametru l, natomiast w ostatnim rzędzie zostały przedstawione zbiorcze wykresy dla wszystkich eksperymentów.


Eksperyment mnożenia macierzy dla k = 2, 3, ... 9 i l = [0, 3, 6, 7, 9]
Czas mnożenia macierzy
Liczba operacji zmiennoprzecinkowych
czas t [s]
liczba operacji - FLO
1e8
l = 0
l = 3
l = 6
l = 7
l = 9

Kształty wykresów są podobne, niezależnie od parametru l. Wynika to z faktu, że porównywane algorytmy mają podobną złożoność obliczeniową. Ciekawe wnioski możemy wysnuć oglądając wykresy zbiorcze. Wynika, z nich oczywisty fakt, że czas obliczeń jest skorelowany z liczbą operacji zmiennoprzecinkowych. Możemy również zauważyć, że dla l równego 3 uzyskujemy najlepszy czas. Oznacza to, że istnieje taki rozmiar macierzy, dla której lepszym znacząco rozwiązaniem jest pomnożenie klasyczne niż wykonanie rekurencyjne. I tak samo opłaca się dla tych większych użyć algorytmu Strassena.