Algorytmy Macierzowe

Sprawozdanie z laboratorium nr.1

Władysław Nieć, Paweł Surdyka

Zadania

1. Rekurencyjne mnożenie macierzy metodą Binét'a (10 punktów)
2. Rekurencyjne mnożenie macierzy metodą Strassena (10 punktów)
3. Mnożenie macierzy metodą znalezioną przez sztuczną inteligencję (10 punktów)

Wstęp

Zaimplementowaliśmy rekurencyjne algorytmy mnożenia macierzy: zwykły, Strassena oraz metodę odkrytą przez sieć neuronową Alpha Tensor. Dodatkowo, każdy z zaimplementowanych algorytmów działa dla macierzy dowolnych wymiarów (o ile ich kształt pozwala na mnożenie).

Algorytm Zwykły

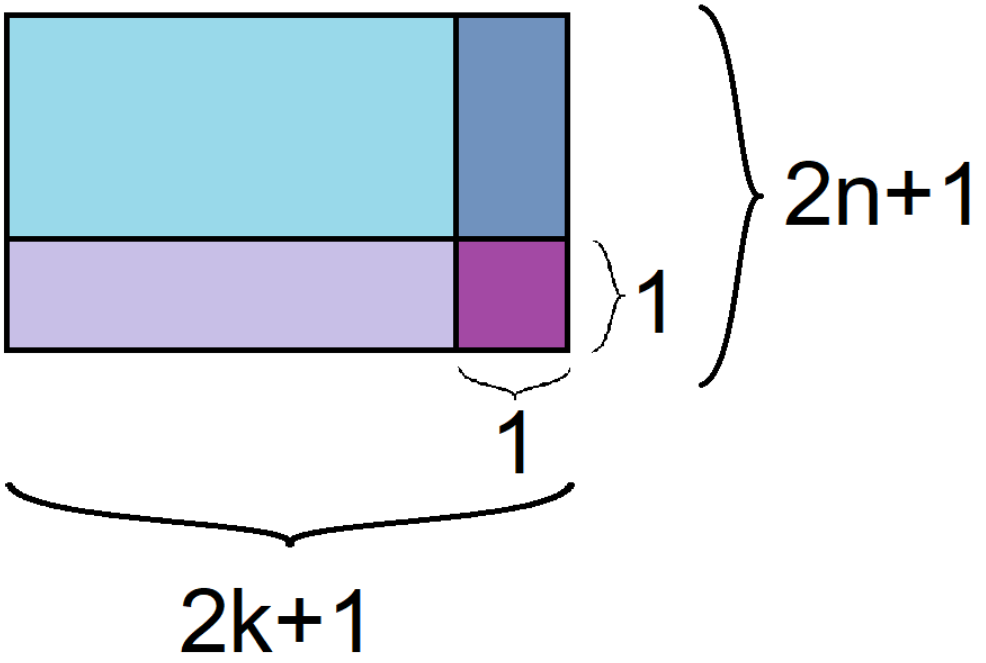
w przypadku standardowego algorytmu mnożenia macierzy, dokonujemy, o ile to możliwe podziału macierzy na 4 części i mnożymy je tak jakby były to zwykłe liczby. Dla każdego z takich mnożeń stosujemy rekurencyjnie ten sam algorytm. Jeżeli podział macierz ma rozmiary nieparzyste, to czwórki nie są idealnie równe wysokości i/lub szerokości macierzy po podziałach mogą różnić się od siebie o 1. Jeżeli wymiaru macierzy nie da się już bardziej podzielić (wynosi on 1) to nie dzielimy macierzy wzdłuż niego, analogicznie zmieniając algorytm mnożenia macierzy, tak jak gdyby było to mnożenie macierzy przez wektor. Wreszcie, gdy obie z mnożonych macierzy mają już rozmiar 1×1, zwracamy ich iloczyn.

```
def multiply(A,B):
    X,Y,Z <- rozmiary macierzy
    if X==1 and Y==1 and Z==1:
        return A*B, 1, 1
    else:
        divide(A, B, X, Y, Z) -> podzielenie macierzy na ćwiartki
        get_c_list(div_a, div_b) -> mnożenie ćwiartek przez siebie (w tej funkcji następuje wywołanie funkcji multiply(A,B))
        C_list -> ćwiartki macierzy wynikowej (puste)
        for i in range(4):
            a,b -> kolejne przemnożone ćwiartki macierzy (np. A11 z B11, A12 z B21)
            if type(a[0]) is not int and type(b[0]) is not int:
                C_list[i] = a[0]+b[0]
            elif type(a[0]) is not int:
                C_list[i] = a[0]
            elif type(b[0]) is not int:
                C_list[i] = b[0]
            else:
                C_list[i] = 0

        U = hstack(C_list[0],C_list[1]) <-scalanie macierzy wynikowej
        L = hstack(C_list[2],C_list[3]) <-scalanie macierzy wynikowej
        C = vstack(U,L) <-scalanie macierzy wynikowej
        return C
```

Pseudokod algorytmu zwykłego

Algorytm Strassena

W przypadku algorytmu Strassena, jeżeli którakolwiek z mnożonych macierzy ma jeden z wymiarów będący wielkością nieparzystą, stosowany jest zwykły algorytm mnożenia, tak by uzyskać macierze o obu wymiarach parzystych i na nich wykonywać dalej mnożenie algorytmem Strassena. Dla macierzy o rozmiarach parzystych dokonywane są podziały na 4 równe części w celu dokonania dalszego mnożenia tym samym algorytmem. W odróżnieniu od poprzedniego algorytmu, algorytm Strassena dokonuje jedynie 7 mnożeń, obniżając teoretyczną złożoność obliczeniową algorytmu



Przykład podziału macierzy o nieparzystych wymiarach w algorytmie Strassena.

```python
def strassen_multiply(A,B,alphaTensor=False):
    if A is None or B is None:
        return 0,0,0
    if A.shape[1] != B.shape[0]:
        raise ValueError
    X,Y,Z = A.shape[0],A.shape[1],B.shape[1]
    if X==0 or Y==0 or Z==0:
        raise ArithmeticError
    if X==1 and Y==1 and Z==1:
        return A*B, 1, 1
    else:
        flops = 0
        mults = 0
        if X%2==1 or Y%2==1 or Z%2==1:
            div_a, div_b = divide_odd(A, B, X, Y, Z)
            c_list = get_c_list(div_a,div_b,not alphaTensor,alphaTensor)
            C_list = [0 for i in range(4)]
            for i in range(4):
                a,b = c_list[i]
                if type(a[0]) is not int and type(b[0]) is not int:
                    flops += a[0].size
                    C_list[i] = a[0]+b[0]
                elif type(a[0]) is not int:
                    C_list[i] = a[0]
                elif type(b[0]) is not int:
                    C_list[i] = b[0]
                else:
                    C_list[i] = 0

            flops +=sum([k[1]+l[1] for k,l in c_list])
            mults +=sum([k[2]+l[2] for k,l in c_list])

            U = hstack(C_list[0],C_list[1])
            L = hstack(C_list[2],C_list[3])
            return vstack(U,L), flops, mults
        else:
            div_a, div_b = divide(A, B, X, Y, Z)
            #dzielę macierze na ćwiartki
            M, flops, mults = get_M_list(div_a,div_b,alphaTensor)#7 mnożeń macierzy
            C_list, n_flops = strassen_add_M(M)#n_flops: dodawania/odejmowania tych macierzy
            U = hstack(C_list[0],C_list[1])
            L = hstack(C_list[2],C_list[3])
            return vstack(U,L), flops + n_flops, mults
```

Główna funkcja algorytmu strassena

```python
def divide_odd(A, B, X, Y, Z):
    x,y,z = X-X%2, Y-Y%2, Z-Z%2
    A11,A12,A21,A22 = A[:x,:y],A[:x,y:],A[x:,:y],A[x:,y:]
    B11,B12,B21,B22 = B[:y,:z],B[:y,z:],B[y:,:z],B[y:,z:]
    if x==0:
        A11, A12 = None, None
    if y==0:
        B11, A11, B12, A21 = None, None, None, None
    if z==0:
        B11, B21 = None, None
    if x==X:
        A21, A22 = None, None
    if y==Y:
        A12, A22, B21, B22 = None, None, None, None
    if z==Z:
        B12, B22 = None, None
    return((A11,A12,A21,A22),(B11,B12,B21,B22))
```

Funkcja odpowiadająca za dzielenie macierzy w taki sposób aby obie lewe górne podmacierze miały szerokość i wysokość długości parzystej

```python
def get_c_list(div_a, div_b, strassen = False,alphaTensor=False):
    A11,A12,A21,A22 = div_a
    B11,B12,B21,B22 = div_b
    if not strassen and not alphaTensor:
        c1 = multiply(A11,B11)
    elif strassen:
        c1 = strassen_multiply(A11,B11)
    else:
        c1 = alpha_tensor_multiply(A11,B11)
    c2 = multiply(A12,B21)


    c3 = multiply(A11,B12)
    c4 = multiply(A12,B22)


    c5 = multiply(A21,B11)
    c6 = multiply(A22,B21)

    c7 = multiply(A21,B12)
    c8 = multiply(A22,B22)

    return [(c1,c2),(c3,c4),(c5,c6),(c7,c8)]
```

Funkcja odpowiadająca za mnożenie podmacierzy. Lewe górne podmacierze są mnożone zgodnie z podaną metodą a pozostałe są mnożone metodą prostszą.

```python
def get_M_list(div_a, div_b,alphaTensor=False):
    A11,A12,A21,A22 = div_a
    B11,B12,B21,B22 = div_b
    multiplication = alpha_tensor_multiply if alphaTensor else strassen_multiply

    #print(div_a, div_b)
    flops = 0
    mults = 0
    M = [0 for i in range(7)]

    M[0] =multiplication(A11+A22,B11+B22)
    flops += (A11.size+B11.size)#dodawania

    M[1] =multiplication(A21+A22,B11)
    flops += A21.size#dodawania

    M[2] =multiplication(A11,B12-B22)
    flops += B12.size#odejmowania

    M[3] =multiplication(A22,B21-B11)
    flops += B21.size#odejmowania

    M[4] =multiplication(A11+A12,B22)
    flops += A11.size#dodawania

    M[5] =multiplication(A21-A11,B11+B12)
    flops += A11.size+B11.size#odejmowania+dodawania

    M[6] =multiplication(A12-A22,B21+B22)
    flops += A12.size+B21.size#odejmowania+dodawania

    flops += sum(i[1] for i in M)#sumowanie flopsów z multiplikacji
    mults += sum(i[2] for i in M)#sumowanie mnożeń z multiplikacji
    return M, flops, mults
```

Mnożenie 7 podmacierzy w metodzie Strassen’a wraz z liczeniem liczby operacji zmiennoprzecinkowych.

```python
def strassen_add_M(M):
    C11 =M[0][0]+M[3][0]+M[6][0]-M[4][0]
    C12 =M[2][0]+M[4][0]
    C21 =M[1][0]+M[3][0]
    C22 =M[0][0]+M[2][0]+M[5][0]-M[1][0]
    flops = 0
    for i in (M[0],M[3],M[6],M[4]):
        if type(i[0])!=int:
            flops += i[0].size
        if type(C11)!=int:
            flops -= C11.size
    for i in (M[2],M[4]):
        if type(i[0])!=int:
            flops += i[0].size
        if type(C12)!=int:
            flops -= C11.size
    for i in (M[1],M[3]):
        if type(i[0])!=int:
            flops += i[0].size
        if type(C21)!=int:
            flops -= C11.size
    for i in (M[0],M[2],M[5],M[1]):
        if type(i[0])!=int:
            flops += i[0].size
        if type(C22)!=int:
            flops -= C11.size
    return [C11,C12,C21,C22], flops
```

Obliczanie ćwiartek macierzy wynikowej zgodnie z metodą Strassen'a.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

gdzie

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}); \\ M_2 &= (A_{21} + A_{22})B_{11}; \\ M_3 &= A_{11}(B_{12} - B_{22}); \\ M_4 &= A_{22}(B_{21} - B_{11}); \\ M_5 &= (A_{11} + A_{12})B_{22}; \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}); \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}), \end{aligned}$$

Oraz obliczanie ilości operacji zmiennoprzecinkowych.

Algorytm Alpha Tensor

Sieć neuronowa Alpha Tensor znalazła całą rodzinę algorytmów mnożenia macierzy. Opisu tych algorytmów można dokonać przy użyciu trójwymiarowego tensora. Plik .npz zawierający takie tensory oraz funkcję odczytującą taki opis i tworzącą algorytm mnożenia macierzy wykorzystaliśmy gotowe, pochodzące z:

https://github.com/google-deepmind/alphatensor

```python
def alpha_tensor_multiply(A,B):
    if A is None or B is None:
        return 0,0,0
    if A.shape[1] != B.shape[0]:
        raise ValueError
    X,Y,Z = A.shape[0],A.shape[1],B.shape[1]
    #print(X," ",Y," ",Z)
    if X==0 or Y==0 or Z==0:
        raise ArithmeticError
    if X==1 and Y==1 and Z==1:
        return A*B, 1, 1
    else:
        flops = 0
        mults = 0
        if X!=Y or X!=Z:
            print
            div_a, div_b = divide_non_square(A, B, X, Y, Z)
            c_list = get_c_list(div_a,div_b,strassen = False, alphaTensor = True)
            C_list = [0 for i in range(4)]
            for i in range(4):
                a,b = c_list[i]
                if type(a[0]) is not int and type(b[0]) is not int:
                    flops += a[0].size
                    C_list[i] = a[0]+b[0]
                elif type(a[0]) is not int:
                    C_list[i] = a[0]
                elif type(b[0]) is not int:
                    C_list[i] = b[0]
                else:
                    C_list[i] = 0

            flops +=sum([k[1]+l[1] for k,l in c_list])
            mults +=sum([k[2]+l[2] for k,l in c_list])
            U = hstack(C_list[0],C_list[1])
            L = hstack(C_list[2],C_list[3])
            return vstack(U,L), flops, mults
        else:
            key = ",".join(str(X) for i in range(3))
            if key in square_factorizations:
                u, v, w = factorizations[key]
                n = _get_n_from_factors([u, v, w])
                a = block_split(A, n, n)
                b = block_split(B, n, n)
                algorithm = algorithm_from_factors([u, v, w])
                result = algorithm(a,b)
                return np.array(result[0]).reshape(X,X),result[1], result[2]
            else:
                #szukam algorytmu mnożenia macierzy o bokach będących dzielnikami X
                for k in square_keys:
                    if X%k==0:
                        a = block_split(A, k, k)
                        b = block_split(B, k, k)
                        u, v, w = square_factorizations[k]
                        algorithm = algorithm_from_factors([u, v, w])
                        result = algorithm(a,b)
                        return np.array(result[0]).reshape(X,X),result[1], result[2]
                #jeżeli nie znalazłem algorytmu, to stosuję metodę strassena:
                return strassen_multiply(A,B,alphaTensor=True)
```

Główna funkcja algorytmu Alpha Tensor

```
filename = 'factorizations_r.npz'
with open(filename, 'rb') as f:
    factorizations = dict(np.load(f, allow_pickle=True))
```

Utworzenie słownika faktoryzacji (tensorów). Będzie nam potrzebny w znajdowaniu odpowiednich algorytmów mnożenia macierzy.

```
def divide_non_square(A, B, X, Y, Z):
    p = min(X,Y,Z)
    A11,A12,A21,A22 = A[:p,:p],A[:p,p:],A[p:,:p],A[p:,p:]
    B11,B12,B21,B22 = B[:p,:p],B[:p,p:],B[p:,:p],B[p:,p:]
    if p==X:
        A21, A22 = None, None
    if p==Y:
        A12, A22, B21, B22 = None, None, None, None
    if p==Z:
        B12, B22 = None, None
    return((A11,A12,A21,A22),(B11,B12,B21,B22))
```

Funkcja odpowiadająca za dzielenie macierzy w taki sposób aby obie lewe górne podmacierze były takiego samego rozmiaru oraz miały długość kwadratu liczby naturalnej.

```
def _get_n_from_factors(factors: np.ndarray) -> int:
    u, v, w = factors
    # Assert that the tensor is a cube.
    assert u.shape[0] == v.shape[0]
    assert u.shape[0] == w.shape[0]
    n = int(np.sqrt(u.shape[0]))
    assert u.shape[0] == n ** 2
    return n
```

Oblicza rozmiar matrix multiplication tensor n na podstawie współczynników (factors). Np. przy mnożeniu macierzy 2x2 metodą Strassena współczynniki wynoszą [4, 7], a ta funkcja zwróci 2.

Wejście: Współczynniki - tablica NumPy w kształcie [3, n^2, R] reprezentująca faktoryzacje T_n

Wyjście: n, rozmiar macierzy jest mnożony przez algorytm reprezentowany przez współczynniki

```
def block_split(matrix: np.ndarray, n_rows: int, n_cols: int):
    rows = np.split(matrix, n_rows, axis=0)
    return [np.split(row, n_cols, axis=1) for row in rows]
```

Dzieli macierz na macierz blokową o wymiarach n_rows x n_cols.

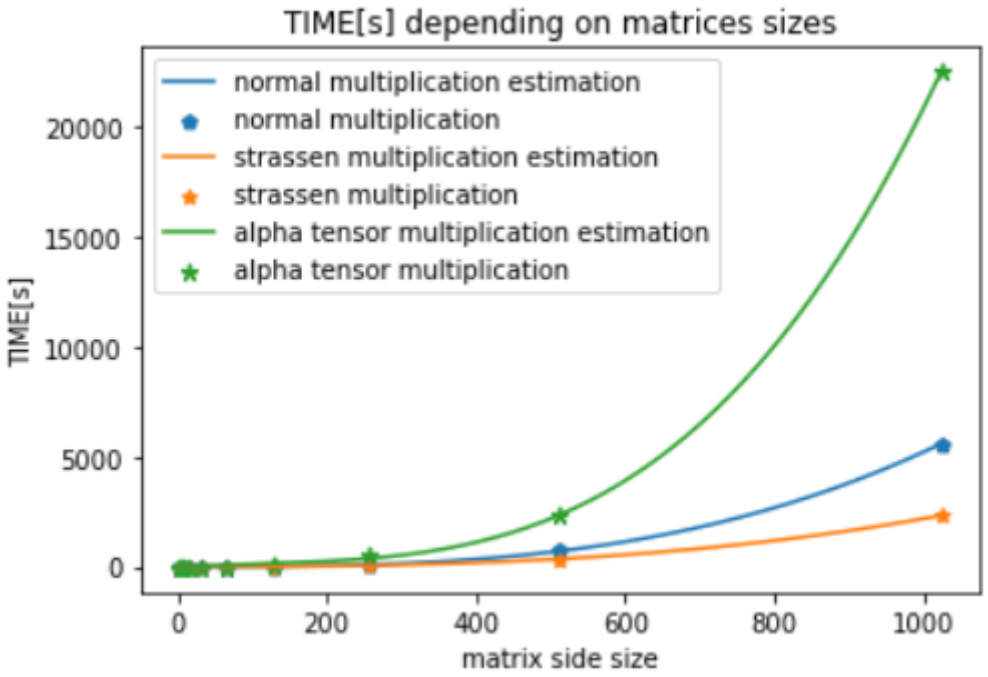
```python
def algorithm_from_factors(factors: np.ndarray) -> Callable[[BlockMatrix, BlockMatrix], BlockMatrix]:
    assert factors[0].shape[0] == factors[1].shape[0]
    assert factors[1].shape[0] == factors[2].shape[0]
    factors = [factors[0].copy(), factors[1].copy(), factors[2].copy()]
    n = int(np.sqrt(factors[0].shape[0]))
    rank = factors[0].shape[-1]
    factors[0] = factors[0].reshape(n, n, rank)
    factors[1] = factors[1].reshape(n, n, rank)
    factors[2] = factors[2].reshape(n, n, rank)
    # The factors are for the transposed (symmetrized) matrix multiplication
    # tensor. So to use the factors, we need to transpose back.
    factors[2] = factors[2].transpose(1, 0, 2)

    def f(a: BlockMatrix, b: BlockMatrix) -> BlockMatrix:
        """Multiplies block matrices `a` and `b`."""
        n = len(a)
        flops = 0
        mults = 0
        result = [[None] * n for _ in range(n)]
        for alpha in range(rank):
            left = None
            for i in range(n):
                for j in range(n):
                    if factors[0][i, j, alpha] != 0:

                        curr = factors[0][i, j, alpha] * a[i][j]#jeżeli factor ==1 lub -1, nie muszę wykonywać mnożenia
                        ops = a[i][j].size if abs(factors[0][i, j, alpha])!=1 else 0
                        flops += ops
                        mults += ops
                        if left is None:
                            left = curr
                        else:
                            flops += left.size
                            left += curr

            right = None
            for j in range(n):
                for k in range(n):
                    if factors[1][j, k, alpha] != 0:
                        curr = factors[1][j, k, alpha] * b[j][k]
                        ops = b[j][k].size if abs(factors[1][j, k, alpha])!=1 else 0
                        flops += ops
                        mults += ops
                        if right is None:
                            right = curr
                        else:
                            flops += right.size
                            right += curr
            matrix_product,flops_n, mults_n = alpha_tensor_multiply(left,right) # left oraz right są wektorami o rozmiarach 1x

            for i in range(n):
                for k in range(n):
                    if factors[2][i, k, alpha] != 0:
                        curr = factors[2][i, k, alpha] * matrix_product
                        ops = matrix_product.size if abs(factors[2][i, k, alpha])!=1 else 0
                        flops += ops
                        mults += ops
                        if result[i][k] is None:
                            result[i][k] = curr
                        else:
                            flops += curr.size
                            result[i][k] += curr
        return result,flops+flops_n,mults+mults_n

    return f
```

Zwraca funkcję realizującą algorytm opisany przez ‘factors’

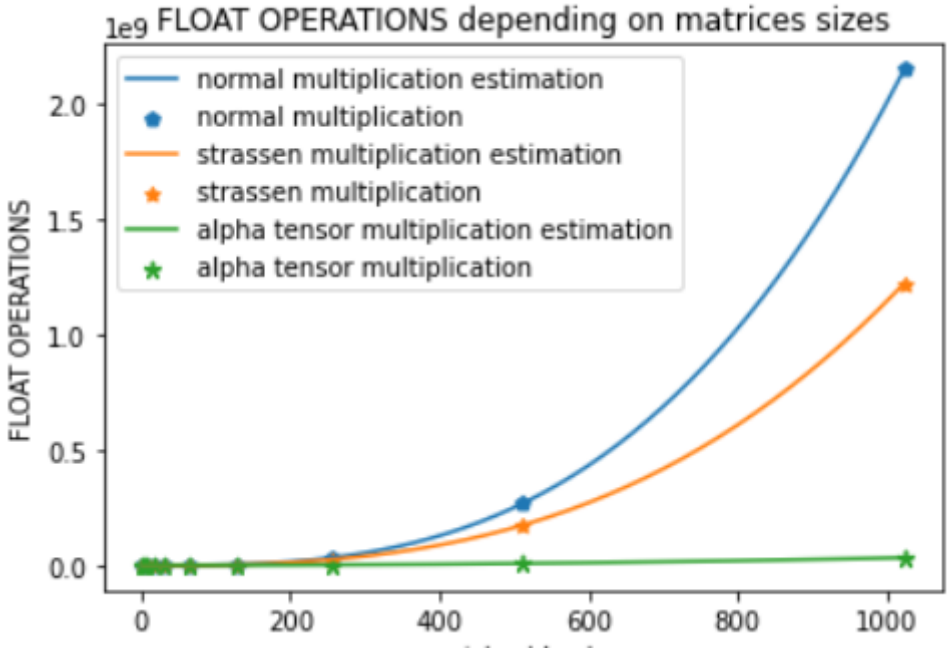
Wejście: Macierzowa faktoryzacja matrix multiplication tensor tj. tablica w postaci [3, n, n, rank].

Wyjście: Funkcja, która z dwóch macierzy blokowych `a` i `b` zwraca macierz blokową `c` postaci c = a @ b (gdzie @ to mnożenie kolejnym macierzy blokowych).

Wykresy



Wykres 1: wykres zależności czasu od rozmiaru macierzy



Wykres 2: wykres zależności ilości operacji od rozmiaru macierzy

Wykresy przedstawiają (interpolowane wielomianem stopnia 3) wyniki dla rozmiarów macierzy w postaci $2^k \times 2^k$, $k \in 1, 2, \ldots, 10$.

Złożoność obliczeniowa

Dla tradycyjnego mnożenia macierzy kwadratowych mamy złożoność $O(n^3)$, ponieważ algorytm dla dwóch macierzy o wymiarach $n \times n$ dokonuje 8 mnożeń na każdym etapie. liczba podziałów macierzy wynosi $log_2(n)$. Zatem liczba operacji zmiennoprzecinkowych wynosi $8^{log_2 n} = (2^3)^{log_2 n} = (2^{log_2 n})^3 = n^3$.

Dla mnożenia macierzy kwadratowych metodą Strassena mamy analogicznie złożoność $O(n^{log_2 7})$ (w przybliżeniu $O(n^{2.479})$), ponieważ algorytm dla dwóch macierzy o wymiarach $n \times n$ dokonuje 7 mnożeń na każdym etapie.

Szacowana złożoność algorytmu Alpha tensor wynosi około $O(n^{2.371})$.

Wnioski

Pomimo lepszej złożoności czasowej i znacznie mniejszej ilości operacji zmiennoprzecinkowych algorytm Alpha Tensor dał gorsze wyniki czasowe, jest to spowodowane tym, że algorytm Strassen'a ma możliwość równoległego przetwarzania danych co znacząco przyspiesza obliczenia w przypadku wielordzeniowych procesorów.