Raport 1
“Rekurencyjne mnożenie macierzy”

Karol Markowicz, Paulina Dziwak

1) Opis pseudokodu algorytmu Bineta z modyfikacją

Rekurencyjny algorytm Bineta opiera się na rekurencyjnym dzieleniu macierzy na cztery mniejsze podmacierze tak długo, aż wielkość uzyskanych podmacierzy będzie równa 2x2. Następnie wykonuje się klasyczne mnożenie takich macierzy po czym macierze wynikowe składa się z powrotem w większą macierz (z czterech elementów tworzymy jeden większy). Algorytm ten zakłada, że mnożone macierze są kwadratowe o rozmiarze: $2^k \; x \; 2^k, \; k \in Z$.

W modyfikacji algorytmu Bineta wprowadzić możemy parametr *l*, którego wartość decyduje od jakiej wielkości macierzy zmieniamy podejście. Podaną w argumencie macierz dzielimy rekurencyjnie do momentu uzyskania rozmiaru $2^l \; x \; 2^l, \; l \in Z$, wówczas stosujemy mnożenie tradycyjne. Tak więc:

Gdy rozmiar macierzy jest <= $2^l, \; l \in Z$:

- Macierze mnożone są klasycznym mnożeniem macierzy.
- Wynik mnożenia jest zwracany jako rezultat.

W przeciwnym przypadku:

- Macierze dzielone są na 4 podmacierze (A11, A12, A21, A22).
- Rekurencyjnie obliczane są cztery iloczyny macierzy.
- Wyniki mnożenia są dodawane do siebie.
- Tworzona jest macierz wynikowa, która łączy te podmacierze i zostaje zwrócona jako wynik.

2) Implementacja algorytmów:

a) algorytm klasyczny:

```
private static float[][] classicalMatrixMultiplication(float[][] A, float[][] B){
    int result_rows = A.length;
    int result_columns = B[0].length;
    int rows = B.length;
    float[][] product = new float[result_rows][result_columns];

    for (int i=0; i < result_rows; i++){
        for(int j=0; j < result_columns; j++){
            product[i][j] = 0;
            for(int k=0; k < rows; k++){
                product[i][j] += A[i][k] * B[k][j];
                operationsCount += 2;
            }
        }
    }
    operationsCount -= (int) Math.pow(A.length,2);
    return product;
}
```

b) algorytm Bineta:

```
private static float[][] binetMatrixMultiplication(float[][] A, float[][] B){
    int n = A.length;
    float[][] C = new float[n][n];
    int halfSize = n/2;
    if(n == 2){
        C = classicalMatrixMultiplication(A, B);
    }
    else{
        float[][] A11 = new float[halfSize][halfSize];
        float[][] A12 = new float[halfSize][halfSize];
        float[][] A21 = new float[halfSize][halfSize];
        float[][] A22 = new float[halfSize][halfSize];
        float[][] B11 = new float[halfSize][halfSize];
        float[][] B12 = new float[halfSize][halfSize];
        float[][] B21 = new float[halfSize][halfSize];
        float[][] B22 = new float[halfSize][halfSize];

        splitMatrix(A, A11,  rowID: 0,  colID: 0);
        splitMatrix(A, A12,  rowID: 0, halfSize);
        splitMatrix(A, A21, halfSize,  colID: 0);
        splitMatrix(A, A22, halfSize, halfSize);
        splitMatrix(B, B11,  rowID: 0,  colID: 0);
        splitMatrix(B, B12,  rowID: 0, halfSize);
        splitMatrix(B, B21, halfSize,  colID: 0);
        splitMatrix(B, B22, halfSize, halfSize);


        float[][] C11 = matrixAddition(binetMatrixMultiplication(A11, B11), binetMatrixMultiplication(A12, B21));
        float[][] C12 = matrixAddition(binetMatrixMultiplication(A11, B12), binetMatrixMultiplication(A12, B22));
        float[][] C21 = matrixAddition(binetMatrixMultiplication(A21, B11), binetMatrixMultiplication(A22, B21));
        float[][] C22 = matrixAddition(binetMatrixMultiplication(A21, B12), binetMatrixMultiplication(A22, B22));

        combineMatrix(C11, C,  rowID: 0,  colID: 0);
        combineMatrix(C12, C,  rowID: 0, halfSize);
        combineMatrix(C21, C, halfSize,  colID: 0);
        combineMatrix(C22, C, halfSize, halfSize);
    }
    return C;
}
```

c) algorytm Bineta z modyfikacją:

Implementacja tego algorytmu wygląda analogicznie do tradycyjnego algorytmu Bineta z drobną modyfikacją w instrukcji warunkowej:

```
if(n <= Math.pow(2,l)){
    C = classicalMatrixMultiplication(A, B);
}
```

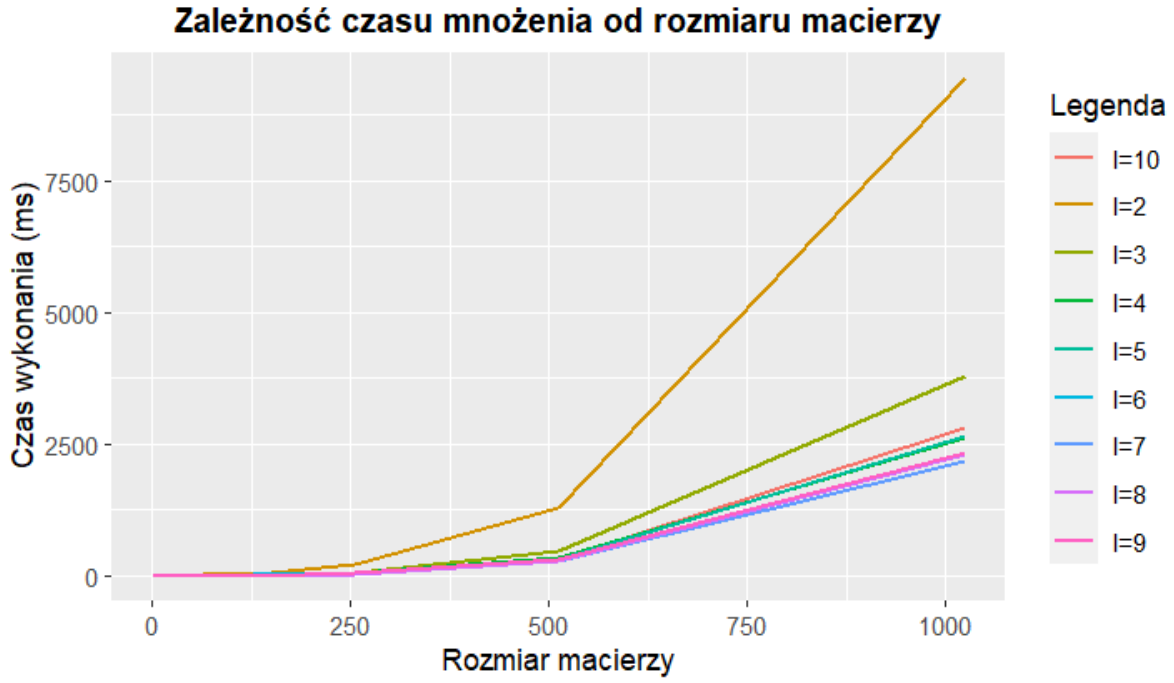
(Pełna implementacja oraz funkcje pomocnicze znajdują się w pliku Main.java)

Program wykonujący mnożenie macierzy generuje 2 pliki. Jeden z nich zawiera informacje na temat czasu wykonania mnożenia, natomiast drugi zawiera liczbę operacji zmiennoprzecinkowych.

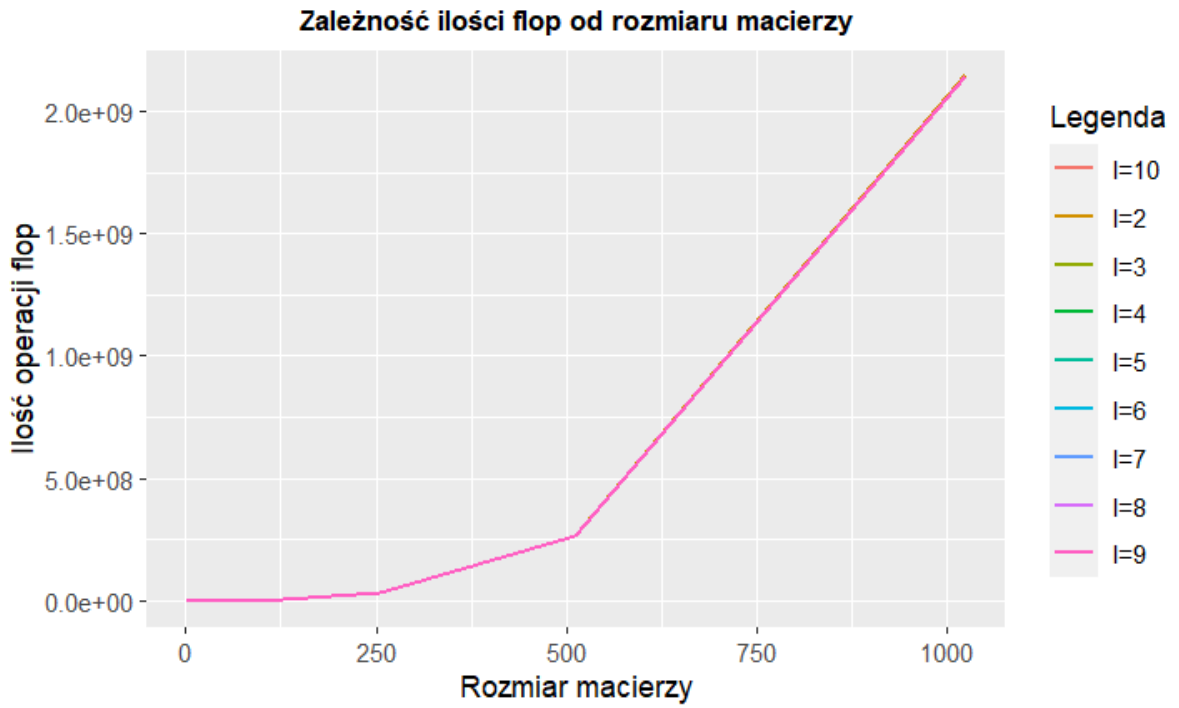
1 2 4 8 16 32 64 128 256 512 1024
l=2
0 0 0 0 0 0 1 5 27 213 1273 9459
l=3
0 0 0 0 0 0 0 1 6 52 459 3797
l=4
0 0 0 0 0 0 0 1 5 49 340 2624
l=5
0 0 0 0 0 0 0 2 9 38 320 2638
l=6
0 0 0 0 0 0 0 1 6 40 280 2170
l=7
0 0 0 0 0 0 0 1 4 32 261 2173
l=8
0 0 0 0 0 0 0 0 3 35 275 2296
l=9
0 0 0 0 0 0 0 0 4 37 309 2341
l=10
0 0 0 0 0 0 0 1 4 35 295 2804

Przykładowo, na powyższym zdjęciu widzimy pomiary czasu. Pierwsza linia zawiera rozmiary macierzy, następne linie wypisują parametr l oraz czas mnożenia dla macierzy o rozmiarach odpowiadających tym znajdującym się w pierwszej linii.
Na podstawie wygenerowanych plików, tworzymy wykresy (przy pomocy języka R)

Wykresy:



W przypadku zależności czasu mnożenia od rozmiaru macierzy widzimy, że wartość parametru “l” pozytywnie wpływa na czas wykonania mnożenia ale tylko do pewnego momentu. Z naszych obserwacji wynika, że warto wspomagać algorytm Bineta mnożeniem tradycyjnym, ale najlepiej od rozmiaru od $2^7 \, x \, 2^7$. Stosowanie tradycyjnego mnożenia dla większych macierzy jest mniej wydajne, dlatego w przypadku naprawdę dużych macierzy warto je przed tradycyjnym mnożeniem podzielić na podmacierze.



FLOPS (Floating Point Operations Per Second) to miara wydajności obliczeniowej komputera, która określa liczbę operacji na liczbach zmiennoprzecinkowych (floating point) wykonywanych przez komputer w ciągu jednej sekundy. W algorytmie Bineta wspomaganym tradycyjnym mnożeniem zliczamy wszystkie operacje zmiennoprzecinkowe, a w naszym przypadku na operacje te składa się mnożenie oraz dodawanie macierzy. Jednakże z naszych obserwacji wynika, że wielkość parametru l nie pomaga w redukcji ilości operacji.