# Algorytmy Macierzowe

## Laboratorium 4
Hierarchiczna kompresja macierzy i operacje na H-macierzach

Marcel Duda
Jan Gawroński

28 stycznia 2026

# 1 Metoda kompresji hierarchicznej

Kompresja macierzy odbywa się poprzez **rekurencyjną dekompozycję drzewa czwórkowego** (quadtree):

1. Macierz dzielona jest rekurencyjnie na cztery równe bloki $2 \times 2$.
2. Dla każdego bloku (węzła liścia) sprawdzana jest możliwość **aproksymacji niskiego rzędu**.
3. Jeśli blok jest wystarczająco "gładki"(niska złożoność), stosowana jest dekompozycja SVD z obcięciem (randomized SVD):

$$B \approx U \cdot \Sigma \cdot V^T \approx U_r \cdot V_r^T \tag{1.1}$$

   gdzie $r \ll \min(m, n)$ jest efektywnym rzędem aproksymacji.
4. W przeciwnym razie blok jest dalej dzielony rekurencyjnie.

Przyjęte parametry kompresji:

- Maksymalny rząd aproksymacji: $r = 8$,
- Tolerancja błędu SVD: $\epsilon = 10^{-6}$.

# 2 Mnożenie Macierz-Wektor

## 2.1 Algorytm

Operacja mnożenia hierarchicznej macierzy przez wektor $y = H \cdot x$ wykonywana jest rekurencyjnie zgodnie ze strukturą drzewa:

**Przypadek 1: Węzeł liścia (Low-Rank Block)** Jeśli blok jest skompresowany w postaci $B \approx U \cdot V^T$, mnożenie wykonywane jest jako:

$$y = U \cdot (V^T \cdot x) \tag{2.1}$$

Złożoność: $\mathcal{O}(r \cdot n)$, gdzie $r$ jest rzędem aproksymacji.

**Przypadek 2: Węzeł wewnętrzny (Internal Node)** Jeśli blok jest podzielony rekurencyjnie na cztery synów:

$$H = \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix}, \quad x = \begin{bmatrix} x_{top} \\ x_{bottom} \end{bmatrix} \tag{2.2}$$

Wynik obliczany jest przez sumowanie wyników z czterech podproblemów:

$$\begin{bmatrix} y_{top} \\ y_{bottom} \end{bmatrix} = \begin{bmatrix} H_{11} \cdot x_{top} + H_{12} \cdot x_{bottom} \\ H_{21} \cdot x_{top} + H_{22} \cdot x_{bottom} \end{bmatrix} \tag{2.3}$$

## 2.2 Implementacja

Poniżej przedstawiono pseudokod funkcji `matrix_vector_mult`:

```cpp
Vector hMatrixVectorMult(const std::shared_ptr<HNode>& H, const Vector& x) {
    if (!H || H->rows == 0) {
        return zeroVector(H ? H->rows : 0);
    }

    // Leaf case: Y = U * (V * x)
    if (H->isLeaf()) {
        if (H->rank == 0) {
            return zeroVector(H->rows);
        }

        // First: temp = V * x (rank x cols) * (cols x 1) = (rank x 1)
        Vector temp = matrixVectorMult(H->V, x);

        // Then: Y = U * temp (rows x rank) * (rank x 1) = (rows x 1)
        return matrixVectorMult(H->U, temp);
    }

    // Internal node case: split vector and recurse
    int splitPoint = H->sons[0]->cols;

    Vector x1(x.begin(), x.begin() + splitPoint);
    Vector x2(x.begin() + splitPoint, x.end());

    Vector y1_1 = hMatrixVectorMult(H->sons[0], x1);
    Vector y1_2 = hMatrixVectorMult(H->sons[1], x2);
    Vector y2_1 = hMatrixVectorMult(H->sons[2], x1);
    Vector y2_2 = hMatrixVectorMult(H->sons[3], x2);

    // Combine: [y1_1 + y1_2; y2_1 + y2_2]
    Vector result;
    result.reserve(H->rows);

    for (size_t i = 0; i < y1_1.size(); ++i) {
        result.push_back(y1_1[i] + y1_2[i]);
    }
    for (size_t i = 0; i < y2_1.size(); ++i) {
        result.push_back(y2_1[i] + y2_2[i]);
    }

    return result;
}
```

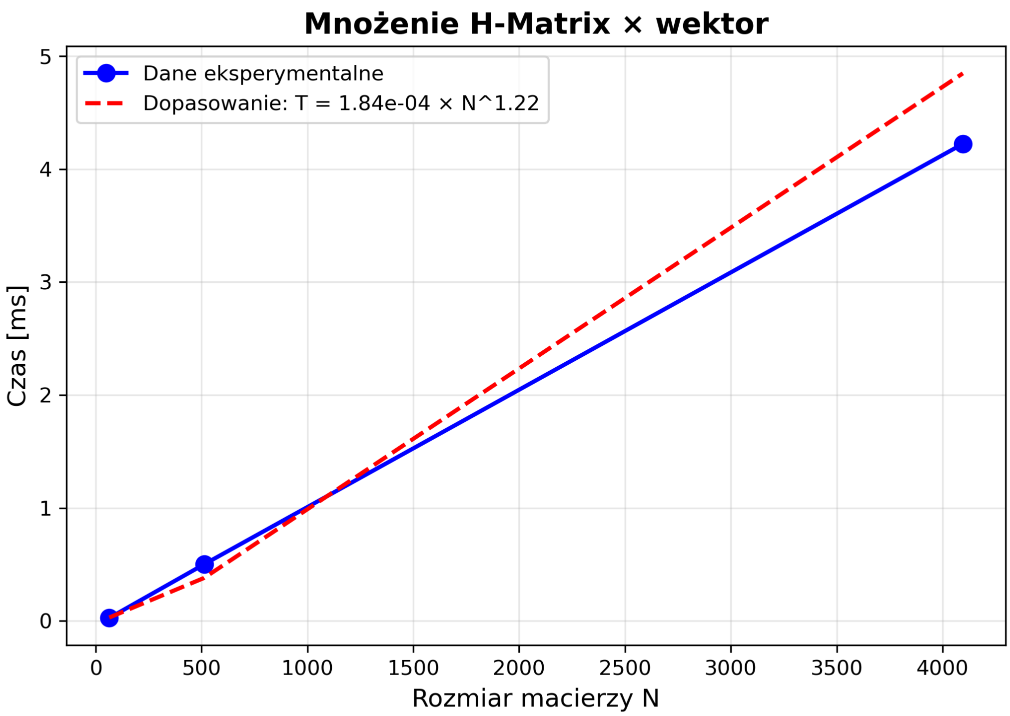
Listing 1: Algorytm mnożenia H-macierzy przez wektor

## 2.3 Wyniki eksperymentalne

### 2.3.1 Czasy wykonania

Tabela 2.1: Czasy wykonania operacji $H \cdot x$ [ms]

| $k$ | Rozmiar $N$ | Czas [ms] |
|---|---|---|
| 2 | 64 | 0.026 |
| 3 | 512 | 0.501 |
| 4 | 4096 | 4.223 |




Rysunek 2.1: Czas wykonania mnożenia H-macierzy przez wektor w funkcji rozmiaru $N$

### 2.3.2 Analiza złożoności

Dopasowanie eksperymentalne funkcji postaci:

$$T(N) = \alpha \cdot N^{\beta} \tag{2.4}$$

Wyniki regresji nieliniowej:

- Współczynnik $\alpha$: 0.00184
- Wykładnik $\beta$: 1.22

## 2.4 Analiza błędu

Błąd aproksymacji mierzony jest jako norma euklidesowa różnicy:

$$\text{Error}_{\text{vec}} = \|A \cdot x - H \cdot x\|_2 = \sqrt{\sum_{i=1}^{N} (A \cdot x)_i - (H \cdot x)_i)^2} \tag{2.5}$$

Tabela 2.2: Błędy aproksymacji dla mnożenia macierz-wektor

| $k$ | Rozmiar $N$ | $\|A \cdot x - H \cdot x\|_2$ |
|---|---|---|
| 2 | 64 | $2.16 \times 10^2$ |
| 3 | 512 | $7.31 \times 10^2$ |
| 4 | 4096 | $2.17 \times 10^3$ |

# 3 Mnożenie Macierz-Macierz

## 3.1 Algorytm

Operacja mnożenia dwóch H-macierzy $C = A \cdot B$ (w szczególności podnoszenie do kwadratu $A^2 = A \cdot A$) wymaga implementacji dwóch funkcji pomocniczych:

### 3.1.1 Dodawanie H-macierzy (`matrix_matrix_add`)

Dodawanie $C = A + B$ wymaga **re-kompresji** wynikowych bloków:

1. Jeśli oba bloki są skompresowane jako $A \approx U_A V_A^T$ i $B \approx U_B V_B^T$:

$$C = U_A V_A^T + U_B V_B^T \approx [U_A \mid U_B] \cdot \begin{bmatrix} V_A^T \\ V_B^T \end{bmatrix} \tag{3.1}$$

2. Wykonywana jest ponowna dekompozycja SVD złączonej macierzy, aby utrzymać niski rząd $r$.

3. W przypadku węzłów wewnętrznych dodawanie jest rekurencyjne dla odpowiednich bloków.

### 3.1.2 Mnożenie H-macierzy (`matrix_matrix_mult`)

Dla węzłów wewnętrznych wykorzystywany jest wzór blokowy:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{3.2}$$

Co prowadzi do:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \tag{3.3}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \tag{3.4}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \tag{3.5}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \tag{3.6}$$

Każdy z 8 wymaganych iloczynów generuje rekurencyjne wywołania mnożenia, a następnie sumowania (z re-kompresją).

## 3.2 Implementacja

```cpp
std::shared_ptr<HNode> hMatrixAdd(const std::shared_ptr<HNode>& A,
                                   const std::shared_ptr<HNode>& B,
                                   int maxRank, double epsilon) {
    if (!A || !B || A->rows != B->rows || A->cols != B->cols) {
        throw std::runtime_error("Invalid matrix dimensions for addition"
            );
    }

    auto result = std::make_shared<HNode>(A->rows, A->cols);

    // Case 1: Both are leaves
    if (A->isLeaf() && B->isLeaf()) {
        // Both zero
        if (A->rank == 0 && B->rank == 0) {
            result->rank = 0;
            result->U = zeroMatrix(A->rows, 0);
            result->V = zeroMatrix(0, A->cols);
            return result;
        }

        // Concatenate U matrices and V matrices, then recompress
        Matrix U_combined;
        Matrix V_combined;

        if (A->rank > 0) {
            U_combined = A->U;
            V_combined = A->V;
        }

        if (B->rank > 0) {
            // Extend U_combined with B->U
            if (U_combined.empty()) {
                U_combined = B->U;
                V_combined = B->V;
            } else {
                for (int i = 0; i < A->rows; ++i) {
                    for (int j = 0; j < B->rank; ++j) {
                        U_combined[i].push_back(B->U[i][j]);
                    }
                }
                for (int i = 0; i < B->rank; ++i) {
                    V_combined.push_back(B->V[i]);
                }
            }
        }

        // Recompress: compute full matrix and do SVD
        Matrix dense = matrixMultiply(U_combined, V_combined);
        auto [U_new, S_new, V_new] = svd_decomposition(dense, maxRank,
            epsilon);

        result->rank = static_cast<int>(S_new.size());
        result->U = U_new;
        result->V = V_new;

        return result;
    }

    // Case 2: Both are internal nodes
    if (!A->isLeaf() && !B->isLeaf()) {
```

```cpp
        result->sons.resize(4);
        for (int i = 0; i < 4; ++i) {
            result->sons[i] = hMatrixAdd(A->sons[i], B->sons[i], maxRank,
                epsilon);
        }
        return result;
    }

    // Case 3: Mixed (one leaf, one internal)
    // Split leaf and recursively add with internal's sons
    // ... (skrocone dla zwiezlosci)
    return result;
}
```

Listing 2: Algorytm dodawania H-macierzy

```cpp
std::shared_ptr<HNode> hMatrixMult(const std::shared_ptr<HNode>& A,
                                   const std::shared_ptr<HNode>& B,
                                   int maxRank, double epsilon) {
    if (!A || !B || A->cols != B->rows) {
        throw std::runtime_error("Invalid matrix dimensions for
            multiplication");
    }

    auto result = std::make_shared<HNode>(A->rows, B->cols);

    // Case 1: Both are leaves
    if (A->isLeaf() && B->isLeaf()) {
        if (A->rank == 0 || B->rank == 0) {
            result->rank = 0;
            return result;
        }

        // Multiply: (U_A * V_A) * (U_B * V_B) = U_A * (V_A * U_B) * V_B
        Matrix middle = matrixMultiply(A->V, B->U);
        Matrix U_result = matrixMultiply(A->U, middle);
        Matrix V_result = B->V;

        // Recompress
        Matrix dense = matrixMultiply(U_result, V_result);
        auto [U_new, S_new, V_new] = svd_decomposition(dense, maxRank,
            epsilon);

        result->rank = static_cast<int>(S_new.size());
        result->U = U_new;
        result->V = V_new;

        return result;
    }

    // Case 2: Both are internal nodes (block multiplication)
    if (!A->isLeaf() && !B->isLeaf()) {
        result->sons.resize(4);

        // Top-left: A1*B1 + A2*B3
        auto temp1 = hMatrixMult(A->sons[0], B->sons[0], maxRank, epsilon
            );
        auto temp2 = hMatrixMult(A->sons[1], B->sons[2], maxRank, epsilon
            );
        result->sons[0] = hMatrixAdd(temp1, temp2, maxRank, epsilon);

```

```
        // Top-right: A1*B2 + A2*B4
        temp1 = hMatrixMult(A->sons[0], B->sons[1], maxRank, epsilon);
        temp2 = hMatrixMult(A->sons[1], B->sons[3], maxRank, epsilon);
        result->sons[1] = hMatrixAdd(temp1, temp2, maxRank, epsilon);

        // Bottom-left: A3*B1 + A4*B3
        temp1 = hMatrixMult(A->sons[2], B->sons[0], maxRank, epsilon);
        temp2 = hMatrixMult(A->sons[3], B->sons[2], maxRank, epsilon);
        result->sons[2] = hMatrixAdd(temp1, temp2, maxRank, epsilon);

        // Bottom-right: A3*B2 + A4*B4
        temp1 = hMatrixMult(A->sons[2], B->sons[1], maxRank, epsilon);
        temp2 = hMatrixMult(A->sons[3], B->sons[3], maxRank, epsilon);
        result->sons[3] = hMatrixAdd(temp1, temp2, maxRank, epsilon);

        return result;
    }

    // Case 3: Mixed (one leaf, one internal) - convert and recurse
    // ... (skrocone dla zwiezlosci)
    return result;
}
```

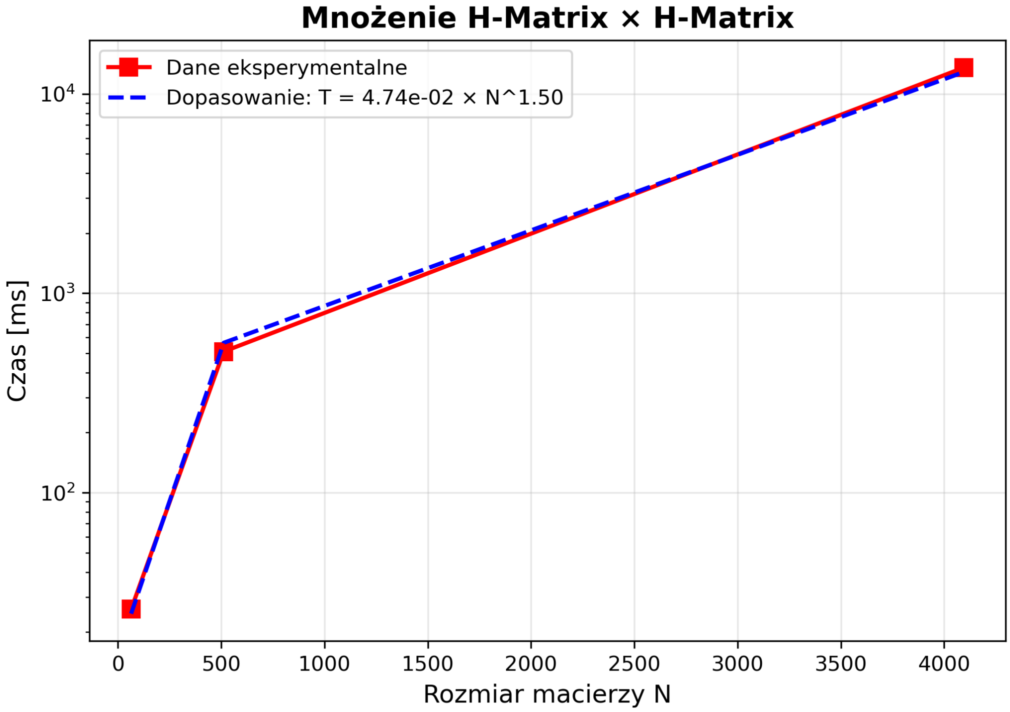
Listing 3: Algorytm mnożenia H-macierzy

## 3.3 Wyniki eksperymentalne

### 3.3.1 Czasy wykonania podnoszenia do kwadratu

Tabela 3.1: Czasy wykonania operacji $H \cdot H$ [ms]

| $k$ | Rozmiar $N$ | Czas [ms] |
|---|---|---|
| 2 | 64 | 26 |
| 3 | 512 | 510 |
| 4 | 4096 | 13558 |




Rysunek 3.1: Czas wykonania mnożenia H-macierzy przez H-macierz w funkcji rozmiaru $N$ (skala logarytmiczna)

### 3.3.2 Analiza złożoności

Eksperymentalne dopasowanie funkcji $T(N) = \alpha \cdot N^{\beta}$:

- Współczynnik $\alpha$: 0.0474
- Wykładnik $\beta$: 1.5

## 3.4 Analiza błędu

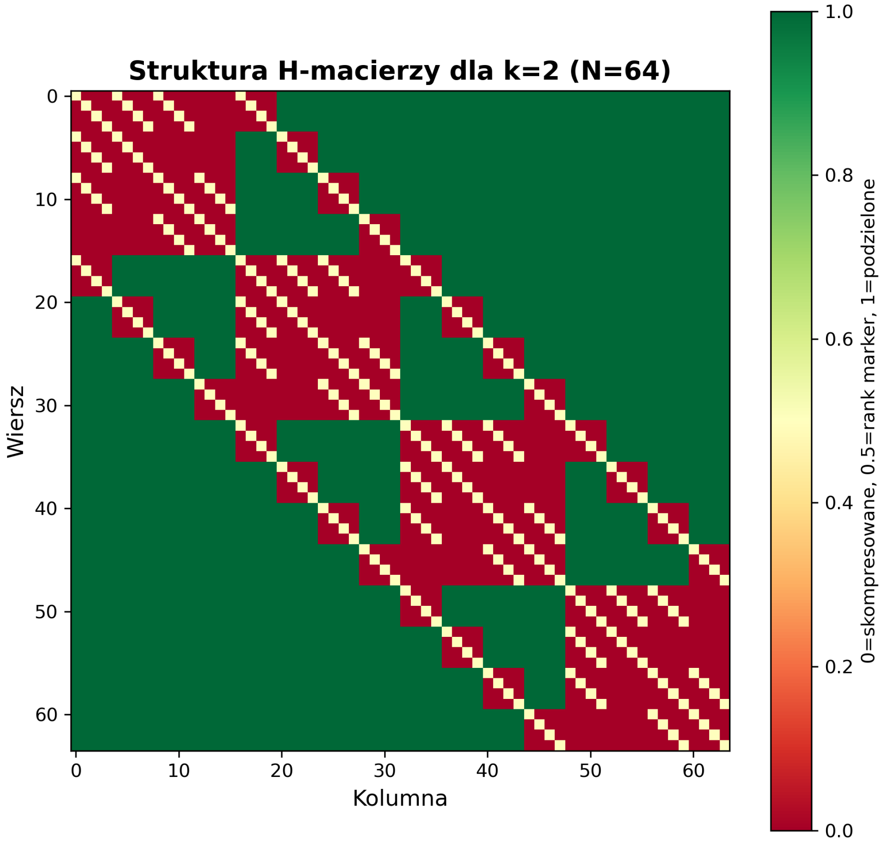
Błąd mierzony jest jako norma Frobeniusa różnicy gęstych macierzy:

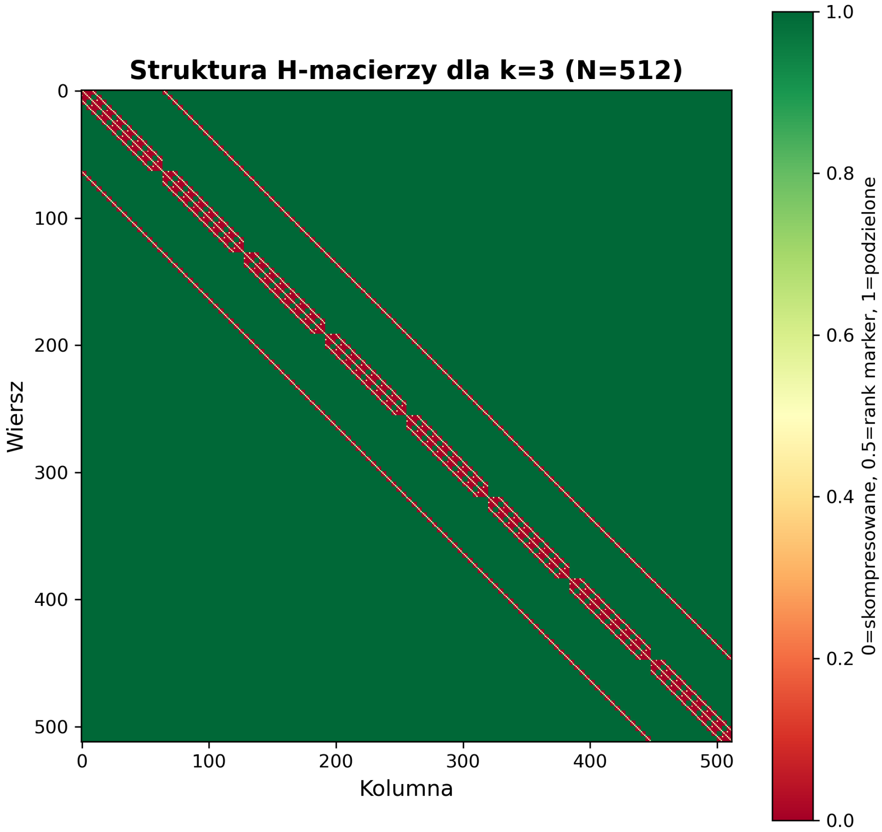
$$\text{Error}_{\text{mat}} = \|A^2 - H^2\|_F = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}((A^2)_{ij} - (H^2)_{ij})^2} \tag{3.7}$$

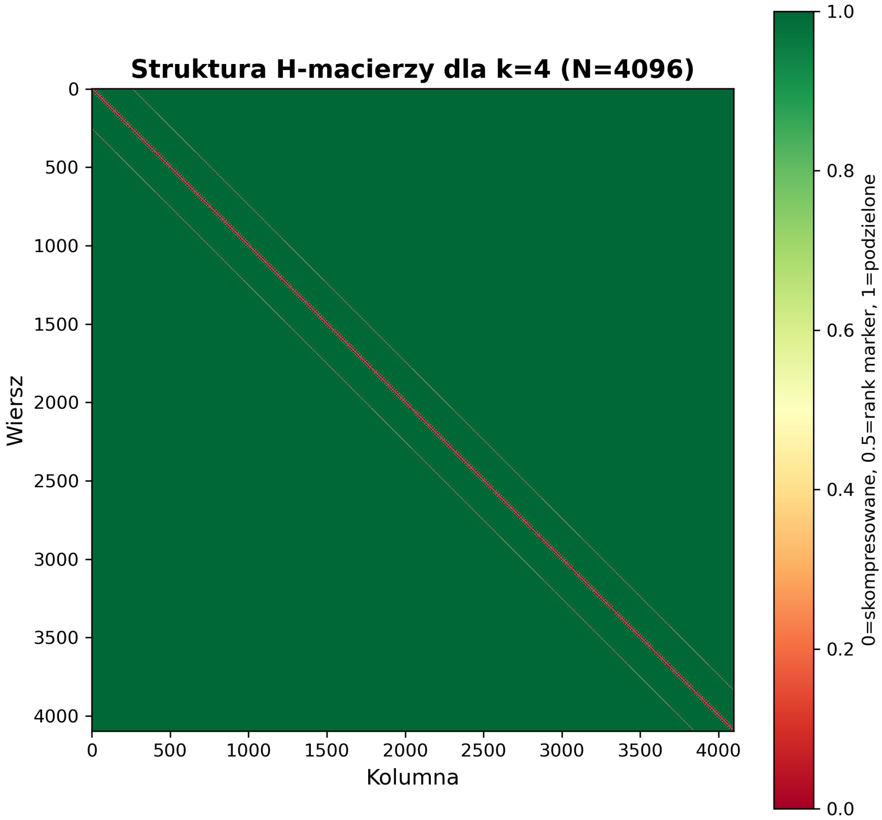
Tabela 3.2: Błędy aproksymacji dla mnożenia macierz-macierz

| $k$ | Rozmiar $N$ | $\|A^2 - H^2\|_F$ |
|---|---|---|
| 2 | 64 | $1.18 \times 10^4$ |
| 3 | 512 | $4.20 \times 10^4$ |
| 4 | 4096 | $1.28 \times 10^5$ |

# 4 Wizualizacja struktury H-macierzy




Struktura H-macierzy dla k=4 (N=4096)
Wiersz
Kolumna
0=skompresowane, 0.5=rank marker, 1=podzielone
0
500
1000
1500
2000
2500
3000
3500
4000
0.0
0.2
0.4
0.6
0.8
1.0