Algorytmy macierzowe
Laboratorium 2

Marcel Duda, Jan Gawroński

17.11.2025

1 Rekurencyjne odwracanie macierzy

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{1}$$

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1} A_{12} S_{22}^{-1} A_{21} A_{11}^{-1} & -A_{11}^{-1} A_{12} S_{22}^{-1} \\ -S_{22}^{-1} A_{21} A_{11}^{-1} & S_{22}^{-1} \end{bmatrix} \tag{2}$$

gdzie $S_{22} = A_{22} - A_{21} A_{11}^{-1} A_{12}$, jeśli A jest macierzą 1×1, to $A^{-1} = \left[\frac{1}{a_{11}}\right]$.

1.1 Implementacja

```
Matrix inverse(const Matrix &A, std::unique_ptr<IMnozenie> & multImpl) {
    if (rows(A) == 1) {
        Matrix invA = zeroMatrix(1, 1);
        invA[0][0] = 1.0 / A[0][0];
        opCounterAdd({0, 0, 0, 1});
        return invA;
    }
    if (rows(A) % 2 == 0) {
        memCounterEnterCall(rows(A), cols(A), 3);
        int halfSize = rows(A) / 2;
        Matrix invA11 = inverse(subMatrix(A, 0, 0, halfSize, halfSize), multImpl);
        Matrix A12 = subMatrix(A, 0, halfSize, halfSize, halfSize);
        Matrix A21 = subMatrix(A, halfSize, 0, halfSize, halfSize);
        Matrix A22 = subMatrix(A, halfSize, halfSize, halfSize, halfSize);

        Matrix T1 = multImpl->multiply(invA11, A12);
        Matrix T2 = multImpl->multiply(A21, invA11);

```

```
        Matrix invS22 = inverse(A22 - multImpl->multiply(A21, T1),
            multImpl);

        Matrix T3 = multImpl->multiply(T1, invS22);

        Matrix B11 = invA11 + multImpl->multiply(T3, T2);
        Matrix B12 = negate(T3);
        Matrix B21 = negate(multImpl->multiply(invS22, T2));
        Matrix B22 = invS22;

        memCounterExitCall(rows(A), cols(A), 3);
        return combine(B11, B12, B21, B22);
    } else {
        Matrix A_padded = pad(A, rows(A) + 1, cols(A) + 1);
        A_padded[rows(A)][cols(A)] = 1.0;
        Matrix inv_padded = inverse(A_padded, multImpl);
        return trim(inv_padded, rows(A), cols(A));
    }
}
```

1.2 Wykresy

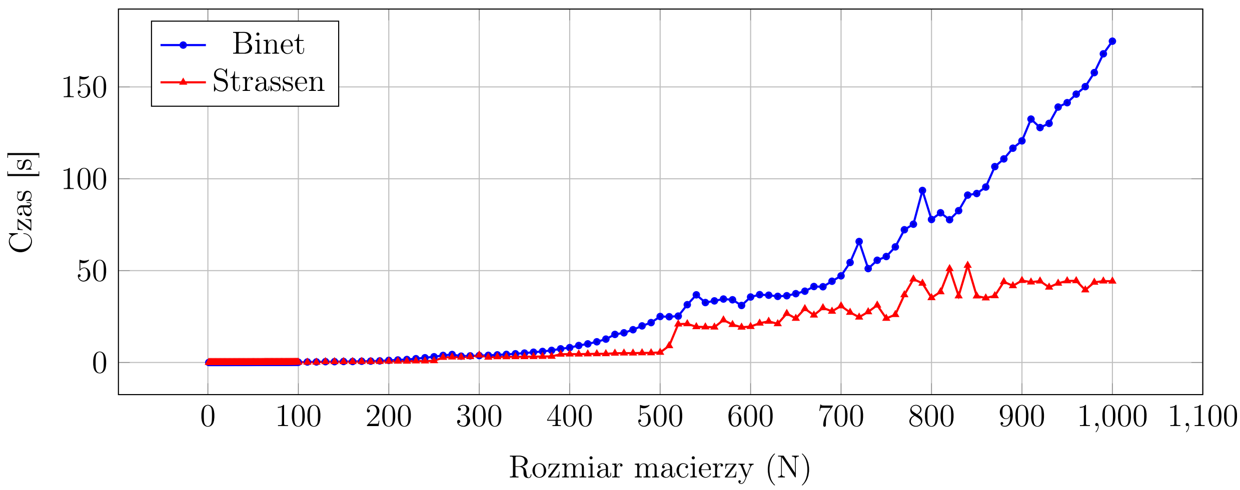

Figure 1.2.1: Czas działania (Binet vs Strassen)

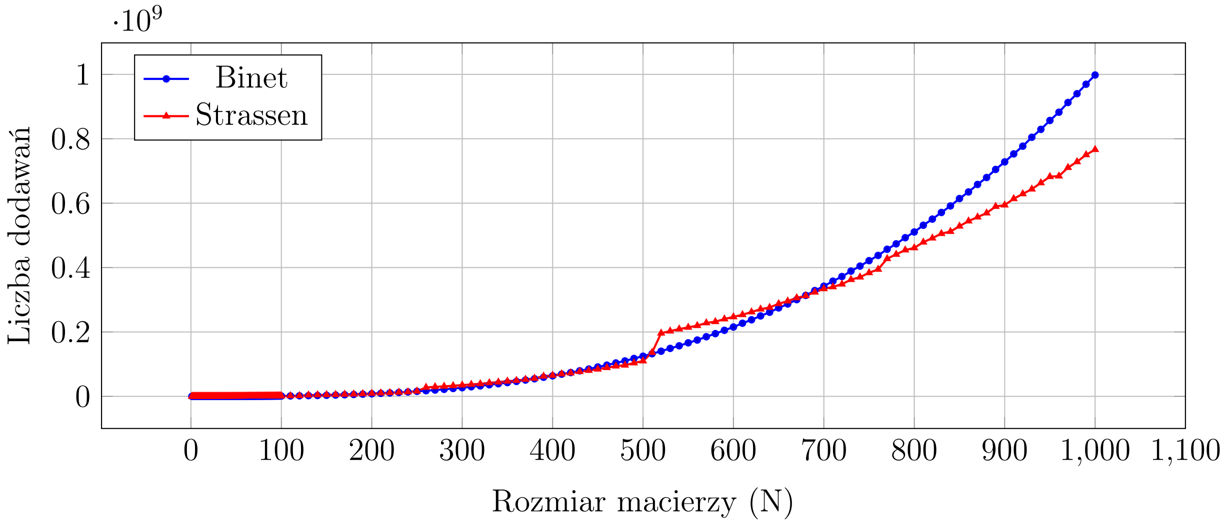

Figure 1.2.2: Porównanie liczby operacji dodawania

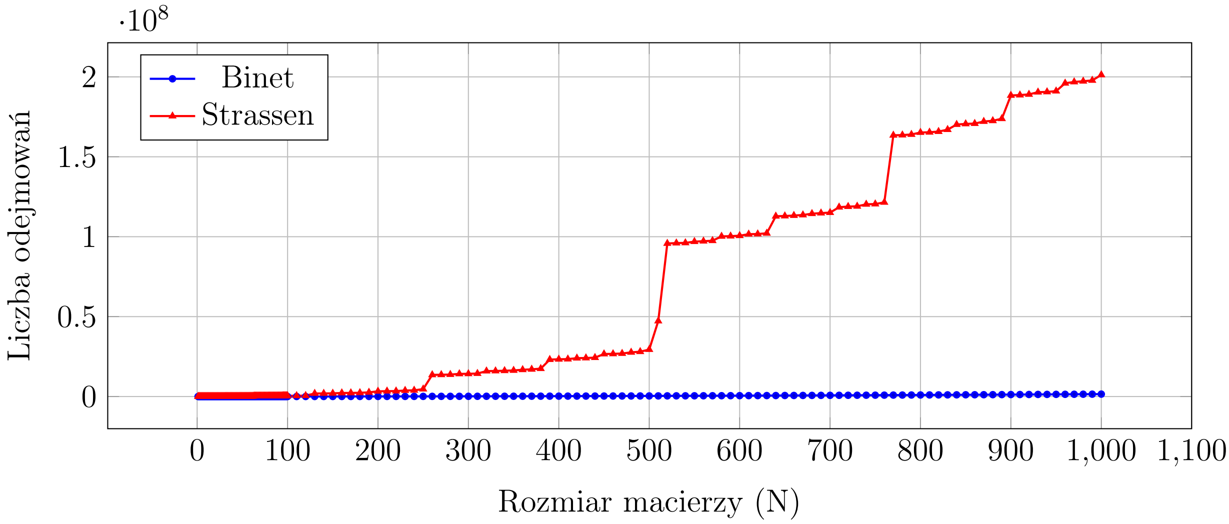

Figure 1.2.3: Porównanie liczby operacji odejmowania

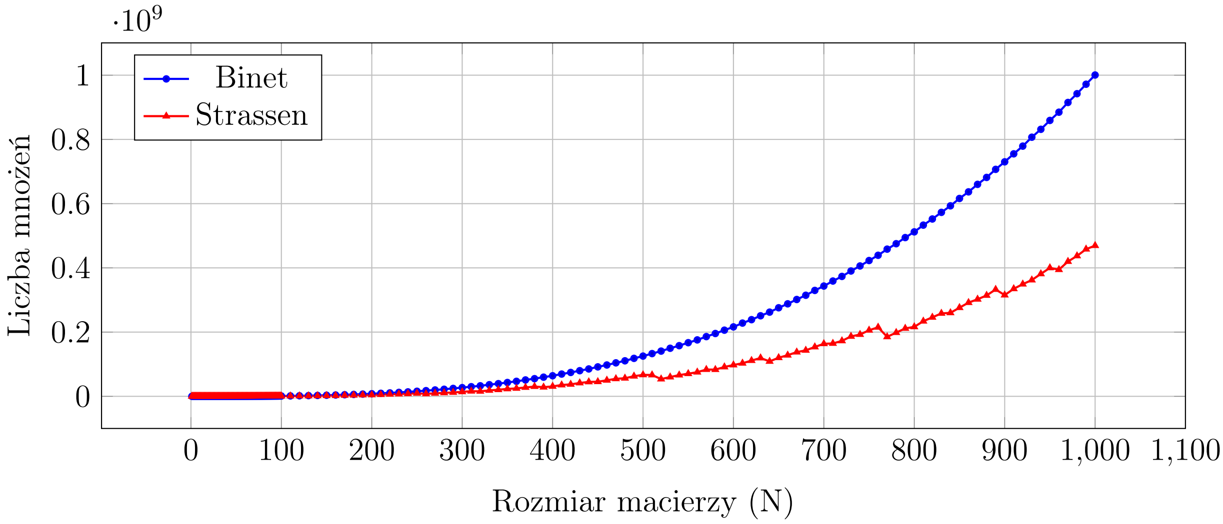

Figure 1.2.4: Porównanie liczby operacji mnożenia

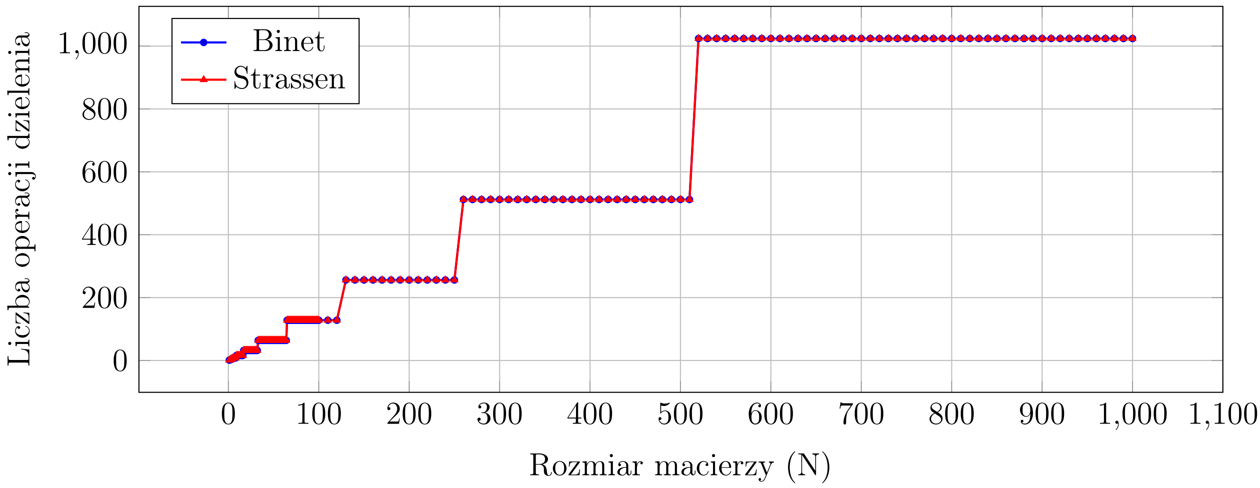

Figure 1.2.5: Porównanie liczby operacji dzielenia

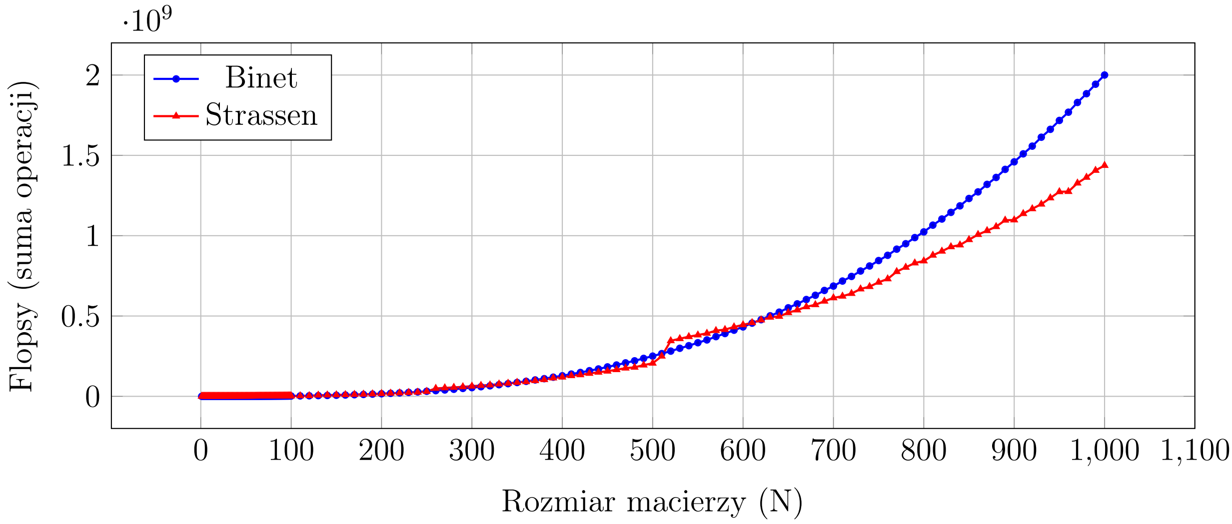

Figure 1.2.6: Porównanie liczby operacji zmiennoprzecinkowych (flops)

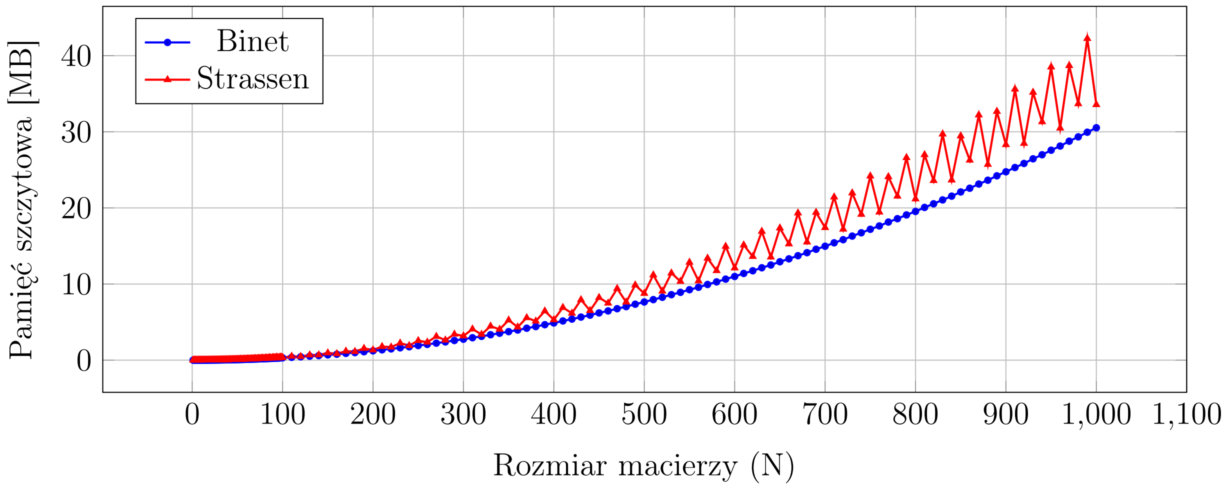

Figure 1.2.7: Porównanie zużycia pamięci

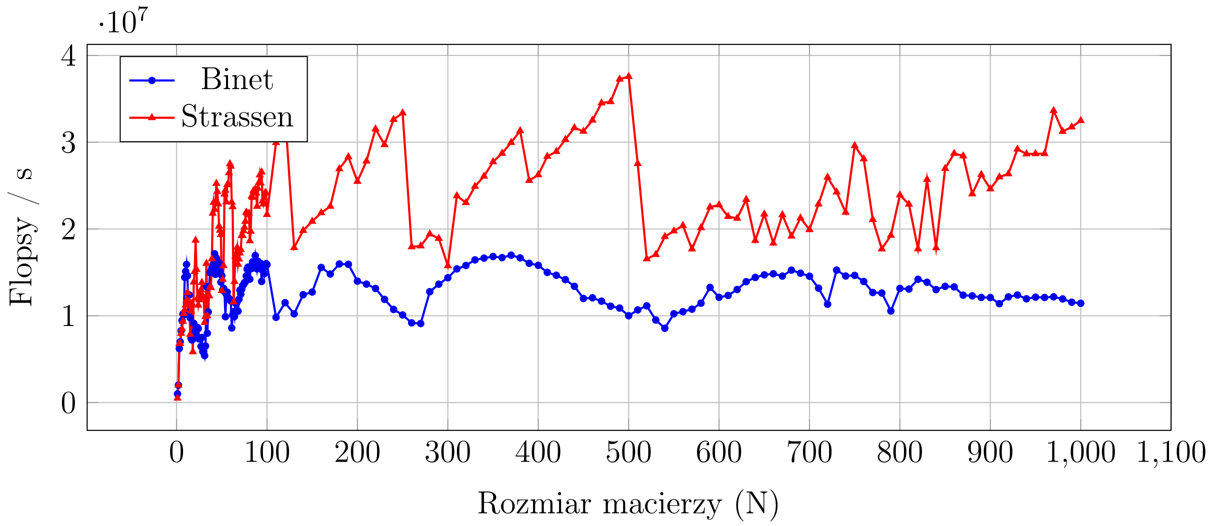

Figure 1.2.8: Przepustowość (flops / s) porównanie

2 Eliminacja Gaussa

Dane A, b:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{3}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5}$$

$$A_{11}x_1 + A_{12}x_2 = b_1 \tag{6}$$

$$A_{21}x_1 + A_{22}x_2 = b_2 \tag{7}$$

$$A_{11} = L_{11}U_{11} \tag{8}$$

$$S = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12} = L_S U_S \tag{9}$$

$$C = \begin{bmatrix} U_{11} & L_{11}A_{12} \\ 0 & U_S \end{bmatrix} \tag{10}$$

$$c = \begin{bmatrix} L_{11}^{-1}b_1 \\ L_S^{-1}(b_2 - A_{21}U_{11}^{-1}L_{11}^{-1}b_1) \end{bmatrix} \tag{11}$$

Zwróć C, c.

2.1 Implementacja

```
std::pair<Matrix, Matrix> GaussElimination(const Matrix &A, const
    Matrix &b, std::unique_ptr<IMnozenie> &multImpl) {
    if (rows(A) == 1) {
        return {A, b};
```

```
    }

    if (rows(A) % 2 == 0) {
        memCounterEnterCall(rows(A), cols(A), 4);

        int halfSize = rows(A) / 2;

        Matrix A11 = subMatrix(A, 0, 0, halfSize, halfSize);
        Matrix A12 = subMatrix(A, 0, halfSize, halfSize, halfSize);
        Matrix A21 = subMatrix(A, halfSize, 0, halfSize, halfSize);
        Matrix A22 = subMatrix(A, halfSize, halfSize, halfSize, halfSize);

        Matrix b1 = subMatrix(b, 0, 0, halfSize, 1);
        Matrix b2 = subMatrix(b, halfSize, 0, halfSize, 1);

        auto [L11, U11] = LUfactorization(A11, multImpl);

        Matrix L11_inv = inverse(L11, multImpl);
        Matrix U11_inv = inverse(U11, multImpl);

        Matrix S1 = multImpl->multiply(A21, U11_inv);
        Matrix S2 = multImpl->multiply(L11_inv, A12);
        Matrix S3 = L11_inv * b1;

        auto [LS, US] = LUfactorization(A22 - multImpl->multiply(S1, S2), multImpl);

        Matrix LS_inv = inverse(LS, multImpl);

        Matrix c1 = S3;
        Matrix c2 = LS_inv * b2 - multImpl->multiply(LS_inv, S1) * S3;

        Matrix C11 = U11;
        Matrix C12 = multImpl->multiply(L11, A12);
        Matrix C21 = zeroMatrix(halfSize, halfSize);
        Matrix C22 = US;

        Matrix C = combine(C11, C12,
                           C21, C22);
        Matrix c = combine(c1, {},
                           c2, {});

        memCounterExitCall(rows(A), cols(A), 4);
        return {C, c};
    } else {
```

```
            Matrix A_padded = pad(A, rows(A) + 1, rows(A) + 1);
            Matrix b_padded = pad(b, rows(b) + 1, 1);
            A_padded[rows(A)][rows(A)] = 1.0;
            b_padded[rows(b)][0] = 0.0;
            auto [C_padded, c_padded] = GaussElimination(A_padded,
               b_padded, multImpl);
            Matrix C = trim(C_padded, rows(A), rows(A));
            Matrix c = trim(c_padded, rows(b), 1);
            return {C, c};
      }
}
```

2.2 Wykresy

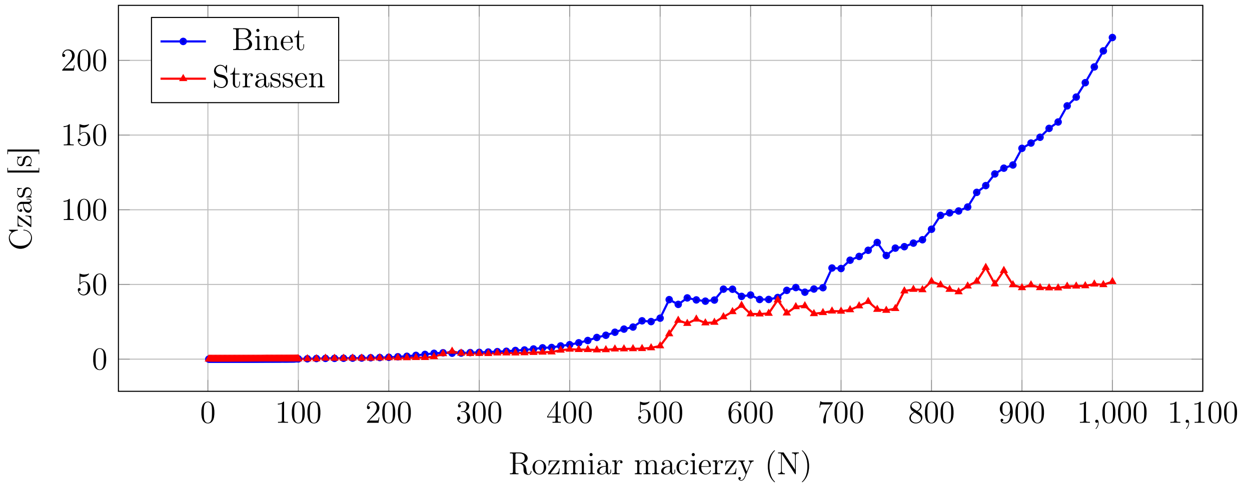

Figure 2.2.1: Czas działania (Binet vs Strassen)

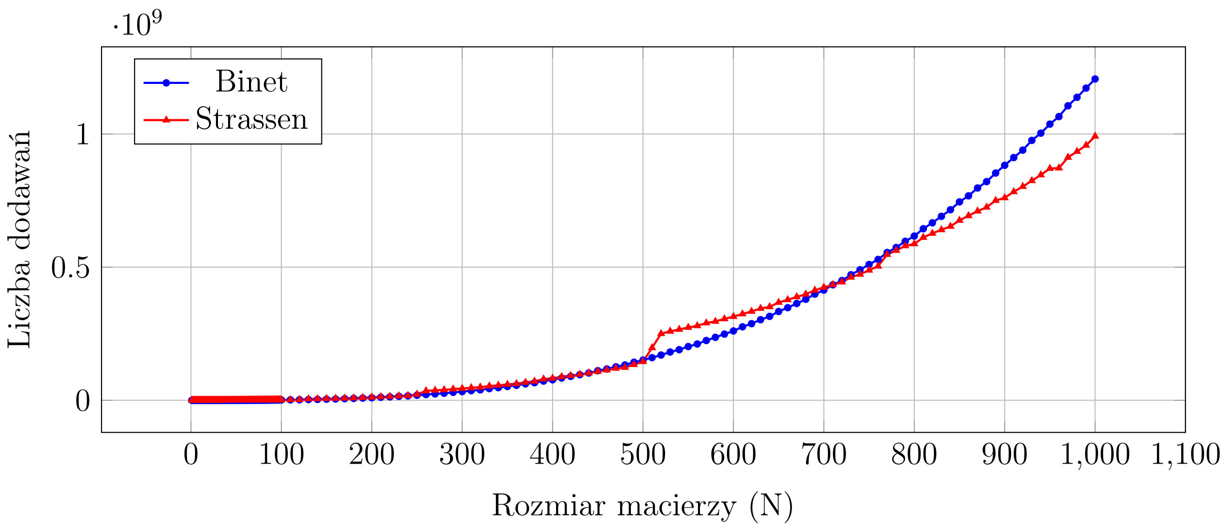

Figure 2.2.2: Porównanie liczby operacji dodawania

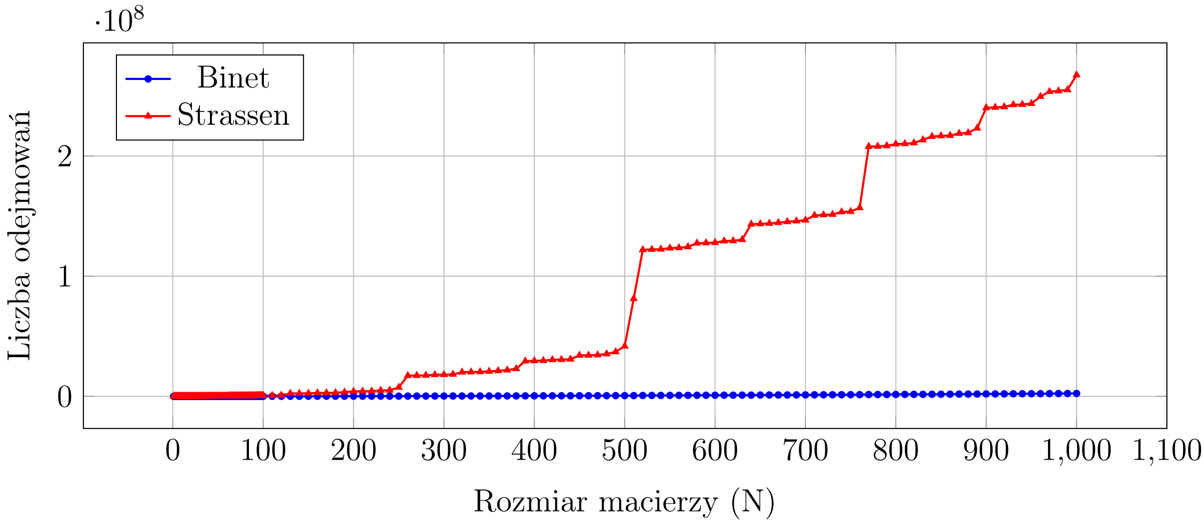

Figure 2.2.3: Porównanie liczby operacji odejmowania

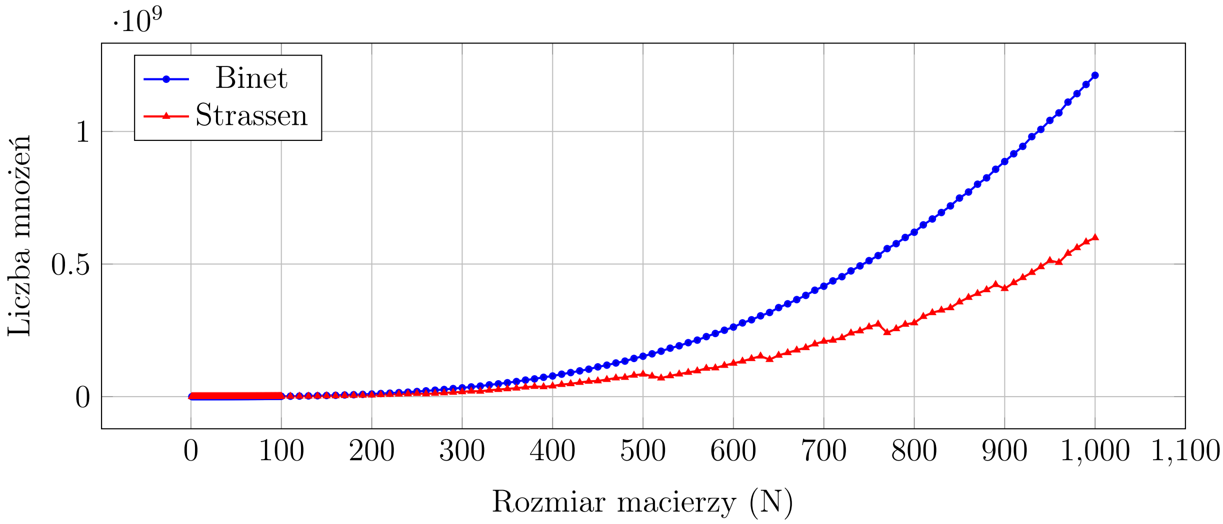

Figure 2.2.4: Porównanie liczby operacji mnożenia

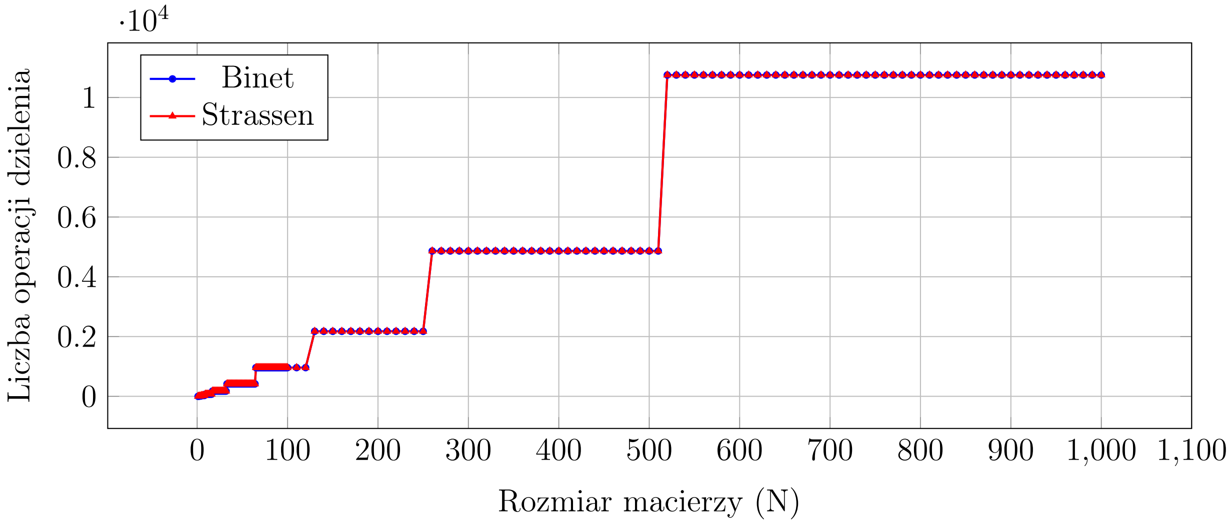

Figure 2.2.5: Porównanie liczby operacji dzielenia

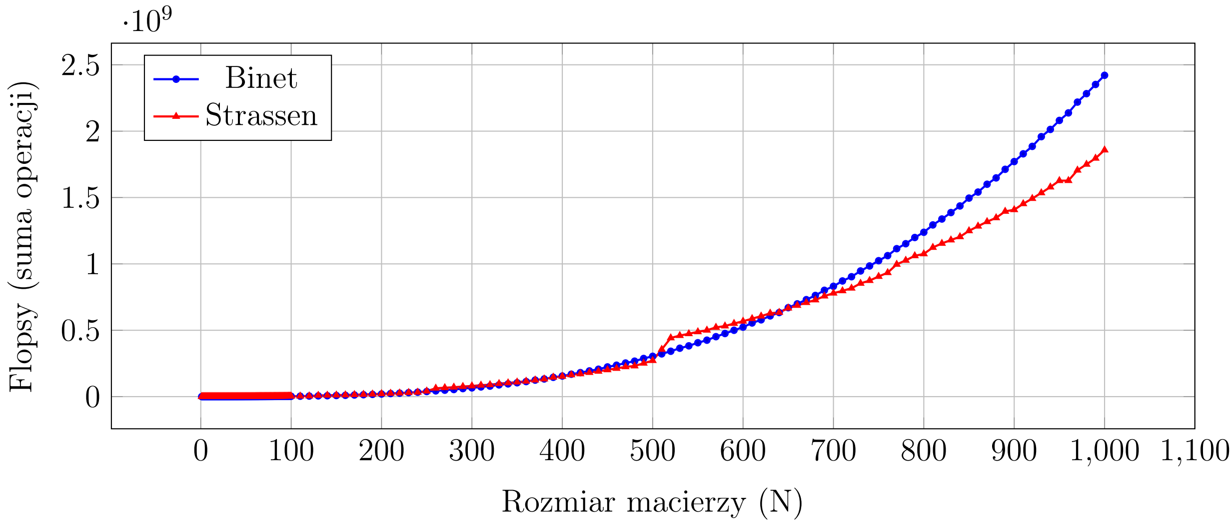

Figure 2.2.6: Porównanie liczby operacji zmiennoprzecinkowych (flops)

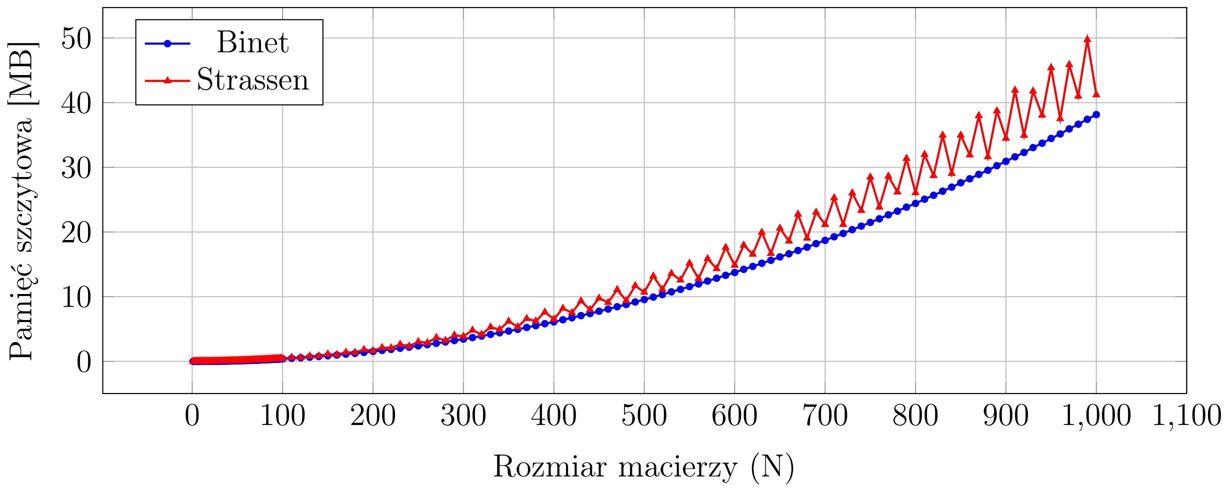

Figure 2.2.7: Porównanie zużycia pamięci

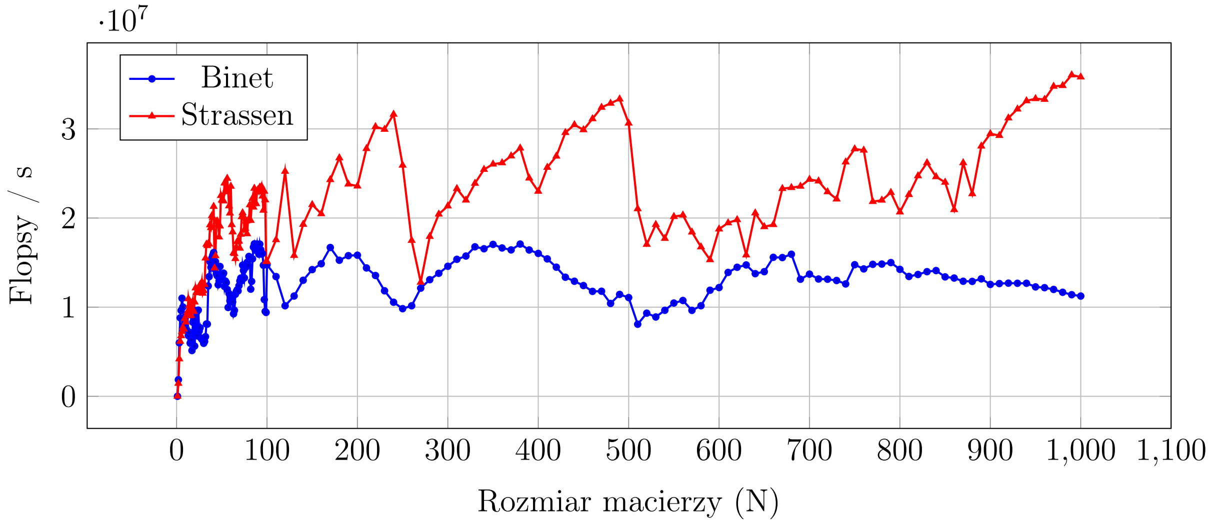

Figure 2.2.8: Przepustowość (flops / s) porównanie

3 LU faktoryzacja

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{12}$$

$$L_{11}U_{11} = LU(A_{11}) \tag{13}$$

$$S = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12} = L_S U_S \tag{14}$$

$$LU(A) = \begin{bmatrix} L_{11} & 0 \\ A_{21}U_{11}^{-1} & L_S \end{bmatrix} \begin{bmatrix} U_{11} & L_{11}^{-1}A_{12} \\ 0 & U_S \end{bmatrix} \tag{15}$$

Wyznacznik $A = LU$ to $\det(A) = \prod_1^n u_{nn}$.

3.1 Implementacja

```
std::pair<Matrix, Matrix> LUfactorization(const Matrix &A, std::
    unique_ptr<IMnozenie> &multImpl) {
     if (rows(A) == 1) {
         Matrix L = identityMatrix(1);
         Matrix U = A;
         return {L, U};
     }

     if (rows(A) % 2 == 0) {
         memCounterEnterCall(rows(A), cols(A), 3);

         int halfSize = rows(A) / 2;

         Matrix A11 = subMatrix(A, 0, 0, halfSize, halfSize);
         Matrix A12 = subMatrix(A, 0, halfSize, halfSize, halfSize)
            ;
         Matrix A21 = subMatrix(A, halfSize, 0, halfSize, halfSize)
            ;
         Matrix A22 = subMatrix(A, halfSize, halfSize, halfSize,
            halfSize);

         auto [L11, U11] = LUfactorization(A11, multImpl);

         Matrix L11_inv = inverse(L11, multImpl);
         Matrix U11_inv = inverse(U11, multImpl);

         Matrix U12 = multImpl->multiply(L11_inv, A12);
         Matrix L21 = multImpl->multiply(A21, U11_inv);

         Matrix S = A22 - multImpl->multiply(L21, U12);

         auto [L22, U22] = LUfactorization(S, multImpl);

         Matrix L = combine(L11, zeroMatrix(halfSize, halfSize),
                            L21, L22);
         Matrix U = combine(U11, U12,
                            zeroMatrix(halfSize, halfSize), U22);

         memCounterExitCall(rows(A), cols(A), 3);
         return {L, U};
     } else {
         Matrix A_padded = pad(A, rows(A) + 1, rows(A) + 1);
         auto [L_padded, U_padded] = LUfactorization(A_padded,
            multImpl);
         Matrix L = trim(L_padded, rows(A), rows(A));
         Matrix U = trim(U_padded, rows(A), rows(A));
         return {L, U};
```

```
    }
}

double determinantLU(const Matrix &A, std::unique_ptr<IMnozenie> &
   multImpl) {
    auto [_, U] = LUfactorization(A, multImpl);
    double det = 1.0;
    for (int i = 0; i < rows(U); ++i) {
        det *= U[i][i];
    }
    return det;
}
```

3.2 Wykresy

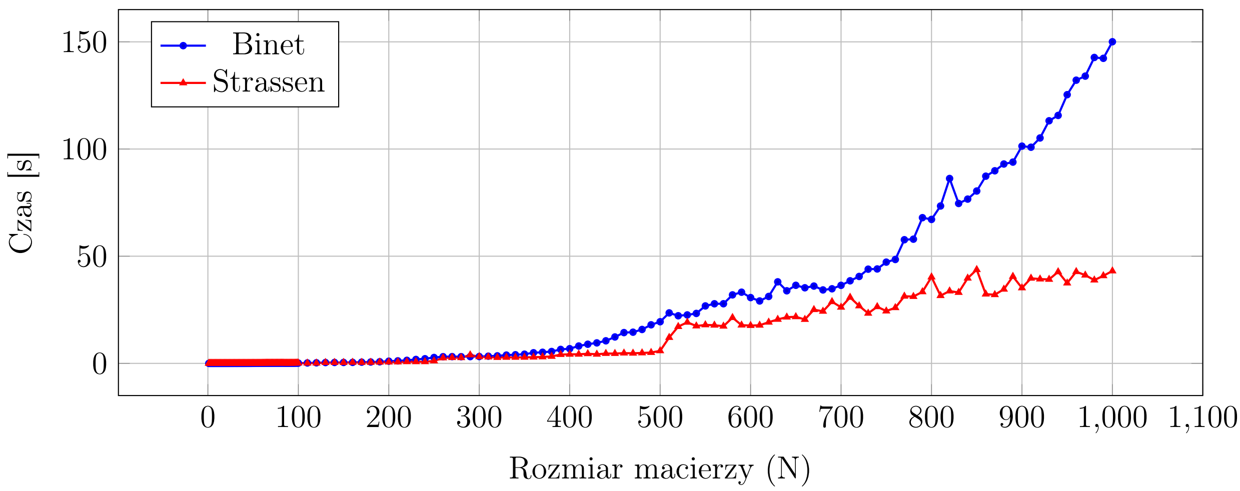

Figure 3.2.1: Czas działania (Binet vs Strassen)

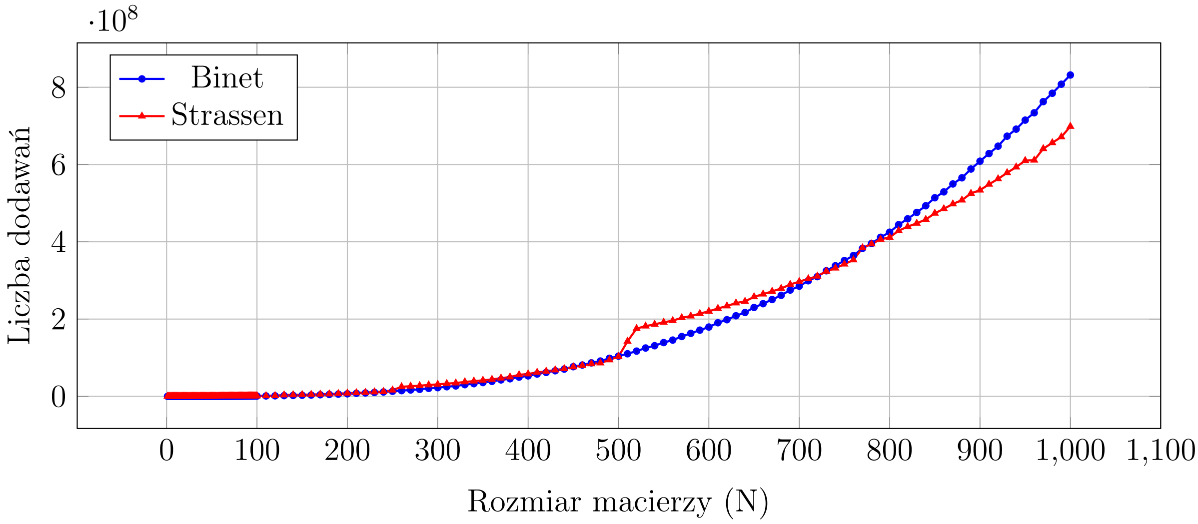

Figure 3.2.2: Porównanie liczby operacji dodawania

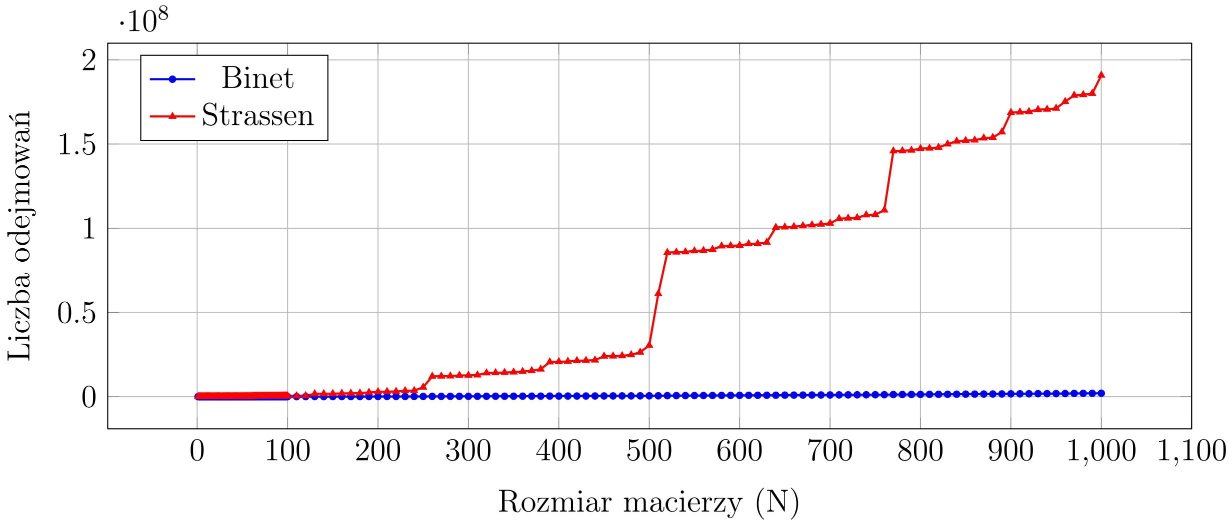

Figure 3.2.3: Porównanie liczby operacji odejmowania

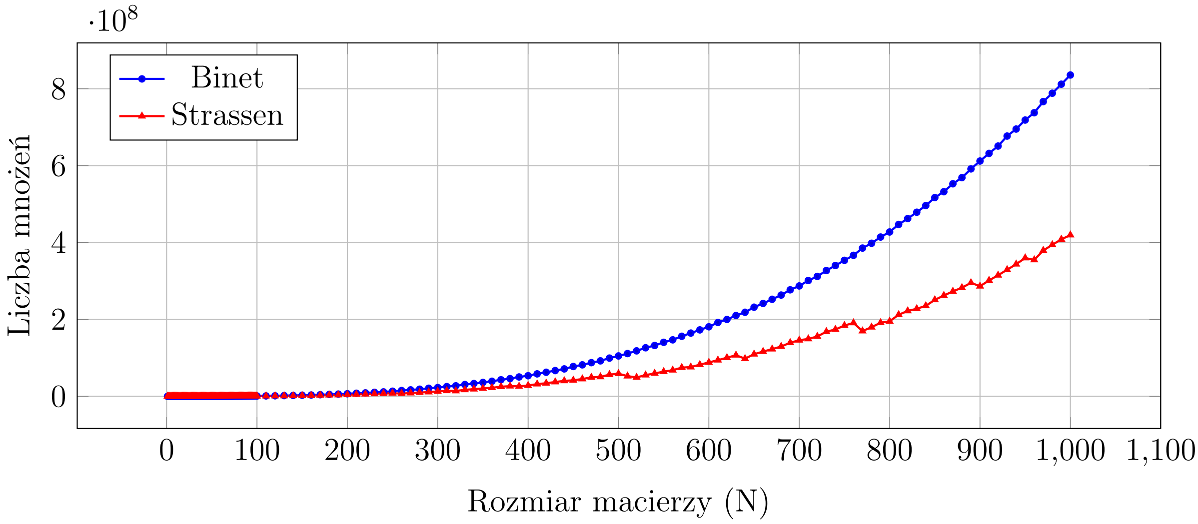

Figure 3.2.4: Porównanie liczby operacji mnożenia

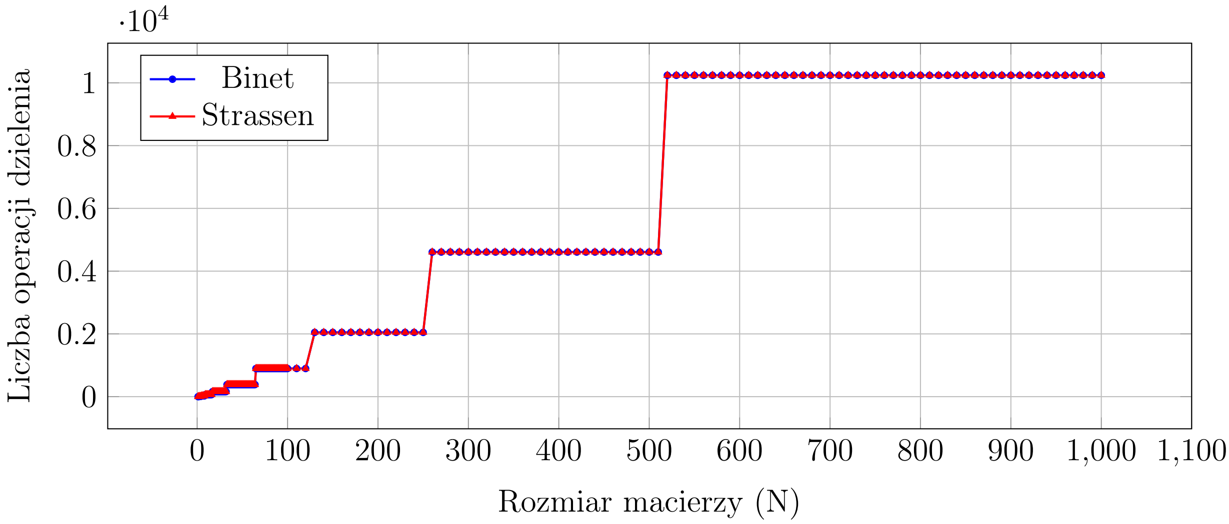

Figure 3.2.5: Porównanie liczby operacji dzielenia

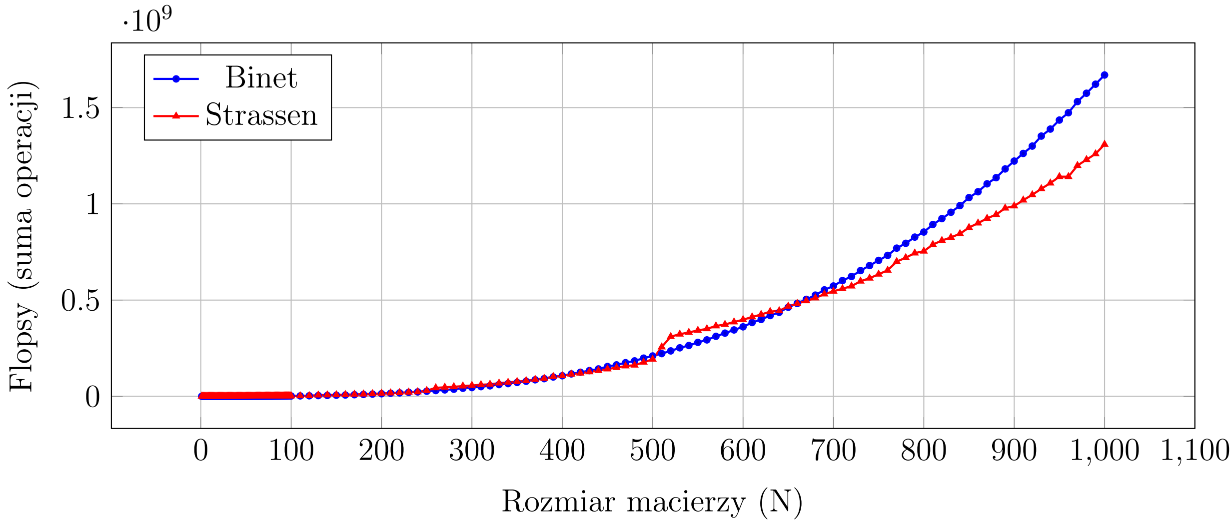

Figure 3.2.6: Porównanie liczby operacji zmiennoprzecinkowych (flops)

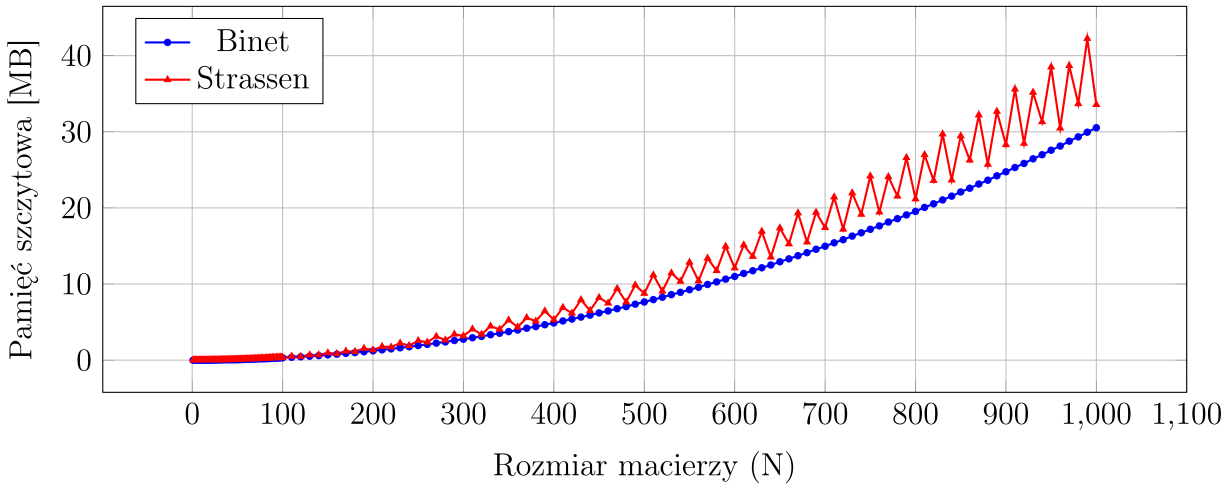

Figure 3.2.7: Porównanie zużycia pamięci

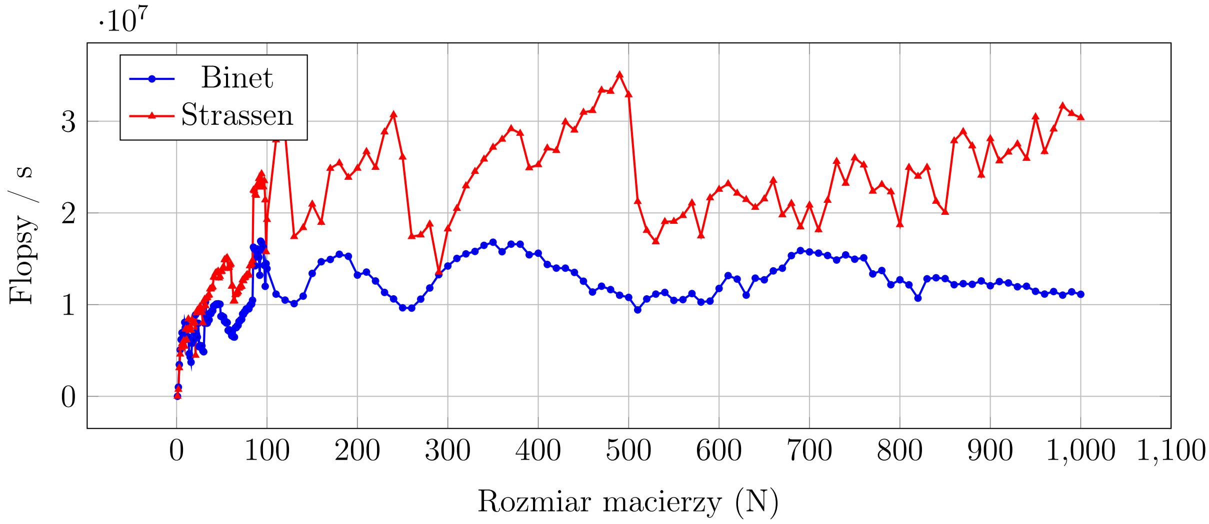

Figure 3.2.8: Przepustowość (flops / s) porównanie