Równoległa implementacja algorytmu Gaussa-Jordana w CUDA

Seweryn Tasior, WI

29 grudnia 2025

1 Wprowadzenie

1.1 Cel ćwiczenia

Celem ćwiczenia jest implementacja równoległego algorytmu eliminacji Gaussa-Jordana do rozwiązywania układów równań liniowych na platformie GPU. Kluczowym aspektem zadania jest dekompozycja algorytmu na elementarne operacje macierzowe,zdefniowanie alfabetu, znalezienie relacji zależności, grafu Diekierta i klasy Foaty co pozwala na ich niezależne wykonanie przez wiele wątków.

1.2 Środowisko testowe

Implementację oraz testy wydajnościowe przeprowadzono na komputerze przenośnym Gigabyte Gaming A16 3VH, wyposażonym w:

- Procesor: 8-rdzeniowy, taktowanie 3.8 GHz.
- Karta graficzna: NVIDIA GeForce RTX 5060
- System operacyjny: Windows 11 Home.

2 Analiza teoretyczna

Rozważamy układ równań liniowych reprezentowany przez macierz rozszerzoną M o wymiarach $n \times (n+1)$, powstałą przez dołączenie wektora wyrazów wolnych b do macierzy współczynników A:

$$M = \left[\begin{array}{cccc|c} M_{11} & M_{12} & \cdots & M_{1n} & b_1 = M_{1(n+1)} \\ M_{21} & M_{22} & \cdots & A_{2n} & b_2 = M_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ M_{n1} & M_{n2} & \cdots & M_{nn} & b_n = M_{n(n+1)} \end{array}\right]$$

2.1 Metoda Gaussa a Gaussa-Jordana

Klasyczna eliminacja Gaussa sprowadza macierz do postaci górnotrójkątnej, co wymaga późniejszego "podstawiania wstecz"w celu obliczenia niewiadomych. Metoda Gaussa-Jordana kontynuuje eliminację, zerując również elementy powyżej głównej przekątnej oraz normalizując przekątną do jedynek. Wynikiem jest macierz jednostkowa w części A oraz wektor rozwiązań w ostatniej kolumnie, co eliminuje etap podstawiania i lepiej nadaje się do zrównoleglenia.

2.2 Niepodzielne zadania obliczeniowe

W celu zrównoleglenia algorytmu, proces obliczeniowy podzielono na pięć typów operacji :

- $A_{i,j}$ - obliczenie mnożnika wykorzystywanego dla wierszy z kolumny j względem pivota w wierszu i, $a_{j,i} = M_{i,i}/M_{j,i}$
- $B_{i,j,k}$ - pomnożenie każdego elementu z k-tego wiersza przez mnożnik, $M_{j,k} = M_{j,k}/a_{j,i}$
- $C_{i,j,k}$ - właściwa eliminacja - odjęcie od każdego elementu z wiersz k-tego elementu wiersza z pivotem tak by zerować j-tą kolumnę , $M_{j,k} = M_{j,k} - M_{i,k}$

- D_i - obliczenie odwrotności elementu na przekątnej macierzy w celu normalizacji (po zakończeniu eliminacji),
  $d_i = 1/M_{i,i}$
- $E_{i,k}$ - doprowadzenie do postaci jednostkowej i otrzymanie wyniku w kolumnie z wyrazami wolnymi
  $M_{i,k} = M_{i,k} * d_i$

2.3 Algorytm sekwencyjny

Eliminację przeprowadzamy tak, aby zerować elementy (oprócz tych leżących na przekątnej inaczej pivotów) w kolejnych kolumnach. W każdym z tych kroków dla każdego wiersza znajdujemy współczynnik $a_j i$ (operacja **A**), następnie mnożymy go przez każdy element w wierszu (operacja **B**) i odejmujemy od niego element z tej samej kolumny (operacja **C**), ale z wiersza zawierającego pivot. Otrzymujemy postać macierzy diagonalnej, którą należy znormalizować. Dla każdego wiersza obliczamy odwrotność elementu na przekątnej d_i (operacja **D**), następnie kolejne elementy z wiersza mnożymy przez d_i (operacja **E**).

Poniższy kod przedstawia logikę algorytmu, która posłużyła do analizy zależności danych.

```
def gauss_jordan(A, b):
    n = len(A)
    for i in range(n):
        # Doklejenie wektora b do macierzy
        A[i].append(b[i])
    for i in range(n):
        for j in range(n):
            if i==j:
                continue
            # Operacja A:
            a_ji = A[i][i] / A[j][i]
            for k in range(n+1):
                # Operacja B:
                A[j][k] = A[j][k] * a_ji
                # Operacja C:
                A[j][k] = A[j][k] - A[i][k]
    for i in range(n):
        # Operacja D:
        d_i = 1.0 / A[i][i]
        for k in range(i, n + 1):
            # Operacja E:
            A[i][k] = A[i][k] * d_i
    return A
```

2.4 Alfabet w sensie teorii śladów

Alfabet zadań Σ dla problemu o rozmiarze N definiujemy jako sumę zbiorów operacji:

$$\Sigma = A \cup B \cup C \cup D \cup E$$

Gdzie każdą z tych operacji definiujemy jako:

- $A = \{A_{i,j} : i, j \in \{1, \ldots, N\} \wedge i \neq j\}$
- $B = \{B_{i,j,k} : i, j \in \{1, \ldots, N\} \wedge k \in \{1, \ldots, N+1\} \wedge i \neq j\}$
- $C = \{C_{i,j,k} : i, j \in \{1, \ldots, N\} \wedge k \in \{1, \ldots, N+1\} \wedge i \neq j\}$
- $D = \{D_i : i \in \{1, \ldots, N\}\}$
- $E = \{E_{i,k} : i \in \{1, \ldots, N\} \wedge k \in \{i, \ldots, N+1\}\}$

2.5 Relacje zależności

Relacja zależności $\mathcal{D}$ określa, które operacje muszą być wykonane sekwencyjnie.

$$\mathcal{D} = \text{sym}\{Z_1 \cup Z_2 \cup Z_3 \cup Z_4 \cup Z_5 \cup Z_6\}^+ \cup I_\Sigma \tag{1}$$

- **Zależności na podstawie przebiegu danych w algorytmie**
  - Zbiór Z_1 Zależność między obliczeniem mnożnika a jego użyciem

$$Z_1 = \{(A_{i,j}, B_{i,j,k}) : i, j \in \{1, \ldots, N\} \wedge k \in \{i, \ldots, N+1\} \wedge i \neq j\} \tag{2}$$

– Zbiór Z_2 Zależność między przygotowaniem wartości a odejmowaniem

$$Z_2 = \{(B_{i,j,k}, C_{i,j,k}) : i, j \in \{1, \ldots, N\} \wedge k \in \{i, \ldots, N+1\} \wedge i \neq j\} \tag{3}$$

- **Zależności wynikające z operacji na danych między różnymi wierszami**
  – Zbiór Z_3 , i Z_4 zależności między kolejnymi krokami eliminacji dla tej samej komórki macierzy (krok i musi zakończyć się przed krokiem $i+1$)

$$Z_3 = \{(C_{i-1,j,i}, A_{i,j}) : i, j \in \{2, \ldots, N\} \wedge i < j\} \tag{4}$$

$$Z_4 = \{(C_{i-1,i,i}, A_{i,j}) : i, j \in \{2, \ldots, N\} \wedge i \neq j\} \tag{5}$$

- **Zależności pojawiające się na etapie normalizacji**
  – Zbiór Z_5, Z_6 obliczenia odwrotności pivota i doprowadzenie do postaci jedynek na przekątnej, uzyskanie wyniku w ostatniej kolumnie

$$Z_5 = \{(D_i, E_{i,k}) : i, k \in \{i, \ldots, N\}\} \tag{6}$$

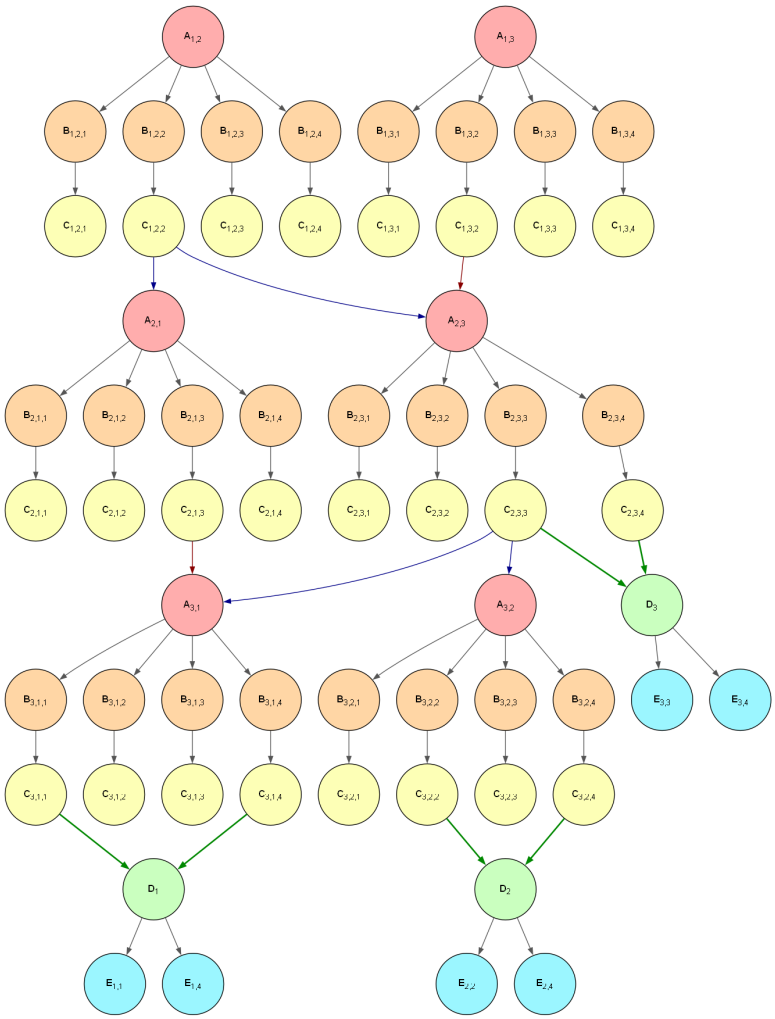
$$Z_6 = \{(C_{n,i,k}, D_i) : i, j \in \{1, \ldots, N\} \wedge k \in \{i, \ldots, N+1\}\} \tag{7}$$

W celu ułatwienia tworzenia grafu Diekierta zmniejszono zbiór, bo macierz jednostkowa (w innych miejscach zera)

$$Z_6 = \{(C_{n,i,i}, D_i), (C_{n,i,n+1}, D_i) : i \in \{1, \ldots, N\}\} \tag{8}$$

2.6 Graf zależności Diekierta

Ogólny graf dla $N = 3$ przedstawiający zależności pomiędzy operacji



2.7 Postać Normalna Foaty (FNF)

Postać normalna Foaty przedstawia klasy operacji, które mogą być wykonywane równolegle. Dla algorytmu Gaussa-Jordana poszczególne warstwy odpowiadają kolejnym kolumnom pivota:

$$
\begin{aligned}
FNF = \quad & [\{A_{1,2}, A_{1,3}, ..., A_{1,n}\}]_{\equiv_I^+} \frown \\
& [\{B_{1,2,1}, B_{1,2,2}, ..., B_{1,2,n+1}, B_{1,3,1}, B_{1,3,2}, ..., B_{1,3,n+1}, ..., B_{1,n,1}, B_{1,n,2}, ..., B_{1,n,n+1}\}]_{\equiv_I^+} \frown \\
& [\{C_{1,2,1}, C_{1,2,2}, ..., C_{1,2,n+1}, C_{1,3,1}, C_{1,3,2}, ..., C_{1,3,n+1}, ..., C_{1,n,1}, C_{1,n,2}, ..., C_{1,n,n+1}\}]_{\equiv_I^+} \frown \\
& [\{A_{2,1}, A_{2,3}, ..., A_{2,n}\}]_{\equiv_I^+} \frown \\
& [\{B_{2,1,1}, ..., B_{2,2,n+1}, B_{2,3,2}, ..., B_{2,3,n+1}, ..., B_{2,n,2}, ..., B_{2,n,n+1}\}]_{\equiv_I^+} \frown \\
& [\{C_{2,1,1}, ..., C_{2,2,n+1}, C_{2,3,2}, ..., C_{2,3,n+1}, ..., C_{2,n,2}, ..., C_{2,n,n+1}\}]_{\equiv_I^+} \frown \\
& ... \frown \\
& [\{A_{n,1}, A_{n,2}, ..., A_{n,n-1}\}]_{\equiv_I^+} \frown \\
& [\{B_{n,1,1}, B_{n,1,n}, B_{n,1,n+1}, B_{n,2,2}, B_{n,2,n}, B_{n,2,n+1}, ..., B_{n,n-1,n-1}, B_{n,n-1,n}, B_{n,n-1,n+1}\}]_{\equiv_I^+} \frown \\
& [\{C_{n,1,1}, C_{n,1,n}, C_{n,1,n+1}, C_{n,2,2}, C_{n,2,n}, C_{n,2,n+1}, ..., C_{n,n-1,n-1}, C_{n,n-1,n}, C_{n,n-1,n+1}\}]_{\equiv_I^+} \frown \\
& [\{D_1, D_2, ..., D_n\}]_{\equiv_I^+} \frown \\
& [\{E_{1,1}, E_{1,n+1}, E_{2,2}, E_{2,n+1}, ..., E_{n,n}, E_{n,n+1}\}]_{\equiv_I^+}
\end{aligned}
$$

3 Realizacja implementacyjna ćwiczenia

3.1 Środowisko programistyczne

Solver zaimplementowano w języku C++ z wykorzystaniem CUDA API. Obliczenia podwójnej precyzji (`double`) wykonywane są na GPU. Weryfikacja poprawności (porównanie wyników, obliczanie normy błędu) realizowana jest przez zewnętrzny moduł w języku Java (`MatrixTW`). Całość procesu budowania i testowania automatyzują skrypty Makefile i PowerShell.

Cały proces kompilacji i testowania wykonano dla różnych rozmiarów problemu ($N \in \{5, 10, 20, 50, 100, 400, 1000\}$). W celu wyeliminowania niestabilności numerycznych (np. wartości `NaN` lub błędów maszynowych rzędu 10^{-17}) zastosowano mechanizmy progowania (thresholding) oraz "twardego zerowania"kolumn pivota.

3.2 Wersja 1: Implementacja naiwna (Niestabilna)

Pierwsza wersja opierała się na bezpośrednim odwzorowaniu modelu teoretycznego z podziałem na osobne kernele dla operacji A, B, C, D i E.

Problem: W tej wersji mnożnik zdefiniowano jako $a_{ji} = \frac{A_{ii}}{A_{ji}}$ (odwrotność standardowego mnożnika), co prowadziło do mnożenia wiersza przez bardzo duże wartości, gdy A_{ji} było bliskie zeru. Skutkowało to błędami numerycznymi i wynikami `NaN` dla macierzy o rozmiarze $N > 20$.

```
#define IDX(row, col, N) ((row) * ((N) + 1) + (col))
// Kernel A: Dzielenie przez element eliminowany (niebezpieczne numerycznie)
__global__ void operation_A(double *A, double *a_ji, int i, int N) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < N && j != i) {
        if (abs(A[IDX(j, i, N)]) > 1e-15) {
            a_ji[j] = A[IDX(i, i, N)] / A[IDX(j, i, N)];
        } else {
            a_ji[j] = 0.0;
        }
    }
}

__global__ void operation_B(double *A, double *a_ji, int i, int N) {
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (j < N && k < N + 1 && j != i) {
        if (abs(a_ji[j]) > 1e-20) {
```

```
            A[IDX(j, k, N)] = A[IDX(j, k, N)] * a_ji[j];
        }
    }
}


__global__ void operation_C(double *A, double *a_ji, int i, int N) {
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (j < N && k < N+1 && j != i ) {
        if (abs(a_ji[j]) > 1e-20) {
            A[IDX(j,k,N)] = A[IDX(j,k,N)] - A[IDX(i,k,N)];
        }

        if(abs(A[IDX(j,k,N)]) < 1e-10)
             A[IDX(j,k,N)] = 0.0;
    }

}

__global__ void clean_pivot_column(double *A, int i, int N) {
    int j = blockIdx.x * blockDim.x + threadIdx.x; // Wiersz

    if (j < N && j != i) {
        A[IDX(j, i, N)] = 0.0;
    }
}

__global__ void operation_D(double *A, double* d_i, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        if (abs(A[IDX(i,i,N)]) > 1e-20)
            d_i[i] = 1.0 / A[IDX(i,i,N)];
        else
            d_i[i] = 1.0;
    }
}


__global__ void operation_E(double *A,  double* d_i, int N) {
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N && k < N +1)
        A[IDX(i,k,N)] = A[IDX(i,k,N)] *  d_i[i];
}


// Macierz wraz z doklejnonym wektorem wyrazow wolnych
void gauss_jordan(double * A, int N){
    ...

    for(int i=0;i<N;i++){
        operation_A<<<blocksPerGrid1D, threadsPerBlock1D>>>(d_A,d_multipliers,i,N);
        cudaDeviceSynchronize();

        operation_B<<<blocksPerGrid2D, threadsPerBlock2D>>>(d_A,d_multipliers,i,N);
        cudaDeviceSynchronize();

        operation_C<<<blocksPerGrid2D, threadsPerBlock2D>>>(d_A,d_multipliers,i,N);
        cudaDeviceSynchronize();

        clean_pivot_column<<<blocksPerGrid1D, threadsPerBlock1D>>>(d_A,i,N);
        cudaDeviceSynchronize();

    }

    operation_D<<<blocksPerGrid1D, threadsPerBlock1D>>>(d_A,d_multipliers,N);
    cudaDeviceSynchronize();

    operation_E<<<blocksPerGrid2D, threadsPerBlock2D>>>(d_A,d_multipliers,N);
    cudaDeviceSynchronize();
    ....
}
```

3.3 Wersja 2: Implementacja stabilna

W celu rozwiązania problemów ze stabilnością wprowadzono dwie kluczowe zmiany:

1. **Zmiana definicji mnożnika:** Zastosowano standardową definicję $a_{ji} = \frac{A_{ji}}{A_{ii}}$, co jest numerycznie bezpieczne (zakładając, że na przekątnej nie ma zera).
2. **Połączenie kerneli:** Połączono operacje B i C w jeden kernel `operation_BC`. Zmniejsza to nałożenie pracy na kernele, jednocześnie zachowując poprawność logiczną.
3. **Progowanie:** Dodano warunki ignorujące wartości bliskie zeru ($< 10^{-20}$) oraz "twarde zerowanie"kolumny pivota, aby uniknąć akumulacji błędów zaokrągleń.

Dzięki tym zmianom program poprawnie rozwiązuje układy równań dla $N = 400$ i większych.

```
__global__ void operation_A(double *A, double *a_ji, int i, int N) {
    ....
            a_ji[j] = A[IDX(j, i, N)] / A[IDX(i, i, N)];
    ....
}
// Polaczone operacje B i C
__global__ void operation_BC(double *A, double *a_ji, int i, int N) {
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (j < N && k < N + 1 && j != i) {
         if (abs(a_ji[j]) > 1e-20) {
            A[IDX(j, k, N)] = A[IDX(j, k, N)] - A[IDX(i, k, N)] * a_ji[j];
        }
    }
}
...

void gauss_jordan(double * A, int N){
    ...
    for(int i=0;i<N;i++){
        operation_A<<<blocksPerGrid1D, threadsPerBlock1D>>>(d_A,d_multipliers,i,N);
        cudaDeviceSynchronize();

        operation_BC<<<blocksPerGrid2D, threadsPerBlock2D>>>(d_A,d_multipliers,i,N);
        cudaDeviceSynchronize();

        clean_pivot_column<<<blocksPerGrid1D, threadsPerBlock1D>>>(d_A,i,N);
        cudaDeviceSynchronize();
    }
    ...

}
```

4 Podsumowanie

Zrealizowano implementację algorytmu Gaussa-Jordana na GPU, która dzięki optymalizacjom numerycznym (zmiana mnożnika) i strukturalnym (łączenie kerneli) osiąga stabilność dla dużych układów równań. Analiza śladowa pozwoliła na poprawne zidentyfikowanie zależności, co umożliwiło efektywne zrównoleglenie procesu eliminacji.