Mátrix szorzás gyorsítása

A mátrixszorzás gyorsítása C-ben többféle módon lehetséges. Az alábbiakban bemutatunk néhány optimalizálási technikát:

Optimális sorrend

A hagyományos mátrixszorzás három beágyazott ciklusból áll (i, j, k sorrendben). Azonban a memóriaelérés optimalizálásával (pl. oszlopok helyett sorok szerint haladás) jelentősen növelhetjük a sebességet.

```
void multiplyMatrices(int **A, int **B, int **C, int N) {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {  // Megváltoztatott sorrend
            int r = A[i][k];
            for (int j = 0; j < N; j++) {
                C[i][j] += r * B[k][j];  // Csökkentett memóriahozzáférés
            }
        }
    }
}
```

Miért gyorsabb?

- Az **A** mátrix sorait gyorsabban beolvassuk a CPU gyorsítótárba (cache).
- Az előre kiszámított **r** változó csökkenti az A[i][k] memória-hozzáférését.

Blokkosítás

A blokkosított mátrixszorzás során a mátrixokat kisebb blokkokra bontjuk, és ezeket a blokkokat külön-külön számoljuk ki, így kihasználva a CPU gyorsítótárát.

$$ A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$ B = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

A fenti mátrixokat 2×2-es blokkokra bontjuk, tehát így:

$$ A = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \\ \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} & \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} \end{bmatrix}$$

$$ B = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0

\end{bmatrix} \end{bmatrix}$$

Mátrix szorzása blokkonként

$$ C_{ij} = \sum_{k} A_{ik} \cdot B_{kj} $$

Ahol minden C blokk a megfelelő A és B blokkok szorzataként adódik.

Első blokk (C_{11}): $$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$ $$ \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Számoljuk ki az egyes részmátrixok szorzatát:

$$ \begin{bmatrix} 1\cdot1 + 2\cdot0 & 1\cdot0 + 2\cdot1 \\ 5\cdot1 + 6\cdot0 & 5\cdot0 + 6\cdot1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 3\cdot1 + 4\cdot0 & 3\cdot0 + 4\cdot1 \\ 7\cdot1 + 8\cdot0 & 7\cdot0 + 8\cdot1 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$$

Összeadva: $$ \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ 12 & 14 \end{bmatrix}$$

Végeredmény

Hasonlóan számolva az összes blokkra ez lesz a végeredmény:

$$ C = \begin{bmatrix} 4 & 6 & 4 & 6 \\ 12 & 14 & 12 & 14 \\ 20 & 22 & 22 & 20 \\ 28 & 30 & 30 & 28 \end{bmatrix} $$

Miért gyorsabb ez a módszer?

- Cache-hatékonyság: Egy blokk adatai könnyebben belefférnek a CPU gyorsítótárába.
- Kevesebb memória-hozzáférés: A kisebb méretű részmátrixok többször felhasználhatók anélkül, hogy újra és újra be kellene tölteni a RAM-ból.
- Jól párhuzamosítható: Az egyes blokkok párhuzamosan is számolhatók több CPU magon vagy GPU-n.

C implementáció:

```
void multiplyMatricesBlocked(int A[N][N], int B[N][N], int C[N][N]) {
    // Eredménymátrix nullázása
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
        }
    }
```

```
    // Blokkosított szorzás
    for (int bi = 0; bi < N; bi += BLOCK_SIZE) {
        for (int bj = 0; bj < N; bj += BLOCK_SIZE) {
            for (int bk = 0; bk < N; bk += BLOCK_SIZE) {

                // Blokkon belüli műveletek
                for (int i = bi; i < bi + BLOCK_SIZE; i++) {
                    for (int j = bj; j < bj + BLOCK_SIZE; j++) {
                        for (int k = bk; k < bk + BLOCK_SIZE; k++) {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
                }
            }
        }
    }
}
```