Calcul Ştiinţific

Capitolul 2: Algoritmi la nivel de bloc

Bogdan Dumitrescu

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica Bucureşti

Cuprins

- Partiţionarea unei matrice în blocuri, operaţii
- Înmulţirea de matrice
- Rezolvarea sistemelor triunghiulare cu parte dreaptă multiplă
- Factorizare LU la nivel de bloc
  - eliminare gaussiană
  - algoritmul Crout
- Triangularizare ortogonală la nivel de bloc

Partiţionare în blocuri

- Fie A o matrice de dimensiune $M \times N$
- O partiţionăm în blocuri $r \times r$ (blocurile ar putea fi şi dreptunghiulare, dar în practică e mai util să fie pătrate)

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix}$$

- Dimensiunea în blocuri este $m \times n$, cu $m = \lceil M/r \rceil$, $n = \lceil N/r \rceil$
- Blocurile de pe ultima linie/coloană pot fi mai mici de $r \times r$

Operaţii cu matrice partiţionate

- Partiţionarea în blocuri pătrate permite scrierea simplă a operaţiilor uzuale (adunare, înmulţire, dar şi altele) cu blocuri în loc de elemente
- Adunarea $C \leftarrow A + B$, cu matrice de dimensiune $m \times n$, partiţionate în blocuri $r \times r$, se efectuează adunând blocurile: $C_{ij} \leftarrow A_{ij} + B_{ij}$
- Blocurile se adună prin operaţii cu elemente
- Produsul $C \leftarrow AB$ (presupunem $m = n$) se efectuează prin

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

- Exerciţiu: detaliaţi cazul în care blocurile de pe ultima linie/coloană sunt incomplete, iar matricele sunt dreptunghiulare

Înmulţirea la nivel de bloc

- Vrem să implementăm înmulţirea de matrice pe un calculator cu memorie ierarhică (implementăm BLAS-3 !)
- Scop: minimizarea traficului între memoria principală MP şi memoria rapidă MR
- Matricele A, B, C sunt stocate în MP
- Alegem r astfel încât 3 blocuri de matrice să poată fi stocate simultan în MR, $3r^2 < \dim MR$
- Variabilele X, Y şi Z se găsesc în MR

1. `Pentru` $i = 1 : n$
    1. `Pentru` $j = 1 : n$
        1. $Z \leftarrow 0$
        2. `Pentru` $k = 1 : n$
            1. $X \leftarrow A_{ik}$, $Y \leftarrow B_{kj}$
            2. $Z \leftarrow Z + X \cdot Y$
        3. $C_{ij} \leftarrow Z$

Evaluarea performanţei

- Operaţiile de genul $X \leftarrow A_{ik}$ reprezintă transferuri (lente) între MR şi MP
- Înmulţirea a două blocuri $Z \leftarrow Z + X \cdot Y$ are loc cu viteză maximă (ideal, un flop/tact)
- Numărul total de operaţii: $2n^3r^3 = 2N^3$
- Accese la memorie: $2n^3r^2 = 2N^3/r$
- Dacă s-ar lucra tot timpul la nivel de element, numărul de accese la memorie ar fi $2N^3$ (c_{ij} stă în MR)
- Implementare ideală: transferul în MR al blocurilor următoare are loc în timpul înmulţirii blocurilor curente
- Traficul MR $\leftrightarrow$ MP poate fi de obicei făcut în paralel cu operaţiile: avem $2r^2$ elemente de transferat în timpul în care se fac $2r^3$ flopi

Cum alegem dimensiunea blocurilor ?

- Regula evidentă: $3r^2 < \dim MR$
- Dacă vrem ca transferurile să aibă loc în paralel cu operaţiile, atunci $6r^2 < \dim MR$
- Trebuie avută în vedere şi ocuparea MR cu alte variabile, cu instrucţiuni, etc.
- Ajustarea dimensiunii blocurilor se poate face şi experimental
- Pe un calculator paralel trebuie avută în vedere şi păstrarea paralelismului, care în general creşte pe măsură ce r scade (la înmulţirea de matrice efectul e mic)

Algoritmul Strassen

- Este cel mai simplu algoritm rapid de înmulţire de matrice
- Natural structurat pe blocuri
- Partiţionăm A, B şi $C = AB$ în blocuri de dimensiune $N/2 \times N/2$, adică

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- Cu formulele de pe pagina următoare, C se calculează cu 7 înmulţiri şi 18 adunări de matrice de dimensiune $N/2 \times N/2$, deci aproximativ $\frac{14N^3}{8} + \frac{18N^2}{4}$ flopi
- Algoritmul standard: 8 înmulţiri şi 4 adunări de blocuri, adică aproximativ $2N^3$
- Pentru N suficient de mare, algoritmul Strassen e mai rapid

Algoritmul Strassen—formule

- Produsul se calculează cu formulele

$$C = \begin{bmatrix} M_1 + M_2 + M_3 - M_4 & M_4 + M_6 \\ M_3 + M_5 & M_1 - M_5 + M_6 + M_7 \end{bmatrix}$$

$$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \qquad M_5 = (A_{21} + A_{22}) \cdot B_{11}$$
$$M_2 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \qquad M_6 = A_{11} \cdot (B_{12} - B_{22})$$
$$M_3 = A_{22} \cdot (B_{21} - B_{11}) \qquad M_7 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$M_4 = (A_{11} + A_{12}) \cdot B_{22}$$

- Exerciţiu: verificare
- Exerciţiu de imaginaţie: cum i-a venit ideea lui Strassen ?

Recursia Strassen

- Cheia unei viteze net superioare este aplicarea recursivă a formulelor Strassen pentru fiecare înmulţire de blocuri
- La fiecare nivel de recursie dimensiunea problemei se înjumătăţeşte
- Recursia are loc până când se atinge o dimensiune r suficient de mică pentru a avea un număr mic de operaţii, dar suficient de mare pentru a beneficia de stocarea în MR
- Numărul de operaţii $N_{op}(n)$ este definit de recurenţa

$$N_{op}(n) = 7N_{op}(\frac{n}{2}) + 18\frac{n^2}{4}, \qquad N_{op}(r) = 2r^3$$

- Soluţie asimptotică $N_{op}(n) = O(n^{\log_2 7})$, $(\log_2 7 \approx 2.807)$
- Dezavantaje Strassen: memorie ceva mai multă, potenţială instabilitate numerică

Alţi algoritmi rapizi

- Winograd: idee similară cu a lui Strassen
- Aceeaşi complexitate asimptotică cu Strassen, dar mai puţine adunări (15 în formulele de înjumătăţire)
- Recordul de complexitate este $O(N^{2.37})$, dar nu poate fi folosit algoritmic datorită dimensiunilor uriaşe la care ar fi eficient
- Se ştie că produsul de matrice are complexitate cel puţin $O(N^2)$, dar nu se ştie care e complexitatea teoretică precisă
- Cercetarea continuă, mai ales în privinţa implementării şi a obţinerii stabilităţii numerice
- Lectură posibilă: http://www.ics.uci.edu/∼fastmm/

Rezolvarea sistemelor triunghiulare

- Fie A o matrice inferior triunghiulară: $a_{ij} = 0, \forall i < j$
- Algoritm pentru rezolvarea $Ax = b$, pe linii

$$
\begin{aligned}
&x \leftarrow b \\
&\texttt{Pentru } i = 1 : N \\
&\qquad \texttt{Pentru } j = 1 : i - 1 \\
&\qquad\qquad x_i \leftarrow x_i - a_{ij} x_j \\
&\qquad x_i \leftarrow x_i / a_{ii}
\end{aligned}
$$

- Complexitate: N^2
- Fiecare element al matricei este accesat o singură dată, deci nu se poate reutiliza în MR

Sist. triungh. cu parte dreaptă multiplă

- Sistem cu parte dreaptă multiplă:

$$AX = B, \;\; A \in \mathbb{R}^{N \times N}, \; B \in \mathbb{R}^{N \times M}$$

- Se poate aplica algoritmul precedent pentru fiecare coloană: $Ax_j = b_j$, $j = 1 : M$
- Complexitate: MN^2
- Rezolvarea pentru fiecare coloană separat nu este eficientă pe calculatoare cu memorie ierarhică dacă A nu încape integral în MR
- Algoritmul următor reprezintă o schiţă de implementare pentru rutina TRSM din BLAS-3

Algoritm la nivel de bloc (1)

- Partiţionăm A în blocuri $r \times r$

$$\begin{bmatrix} A_{11} & 0 & \dots & 0 \\ A_{21} & A_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix}$$

- X şi B pot fi partiţionate şi pe coloane, generalizarea este imediată
- Blocurile A_{ii} sunt inferior triunghiulare
- Ecuaţia corespunzătoare bloc liniei i

$$\sum_{j=1}^{i} A_{ij} X_j = B_i \implies A_{ii} X_i = B_i - \sum_{j=1}^{i-1} A_{ij} X_j$$

Algoritm la nivel de bloc (2)

- Variabilele C, D, Z sunt memorate în MR
    1. `Pentru` $i = 1 : n$
        1. $D \leftarrow B_i$
        2. `Pentru` $j = 1 : i - 1$
            1. $C \leftarrow A_{ij}$, $Z \leftarrow X_j$
            2. $D \leftarrow D - CZ$
        3. $C \leftarrow A_{ii}$
        4. rezolvă sistemul triunghiular cu p.d.m. $CZ = D$
        5. $X_i \leftarrow Z$
- Sistemul $CZ = D$ se rezolvă folosind algoritmul scalar, pe coloane: $Cz_l = D_l$, $l = 1 : M$
- Exerciţiu: ce se întâmplă dacă ultima bloc linie a lui A are mai puţin de r linii ?

Analiza algoritmului

- Numărul de operaţii aritmetice este acelaşi ca în algoritmul la nivel de element
- Numărul transferurilor $MP \leftrightarrow MR$ este

$$\sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} (r^2 + 2rM) + r^2 + rM \right) \approx \frac{MN^2}{r}$$

- Numărul de accese la MP este de aproximativ r ori mai mic decât numărul de operaţii aritmetice
- Dacă se partiţionează X şi B şi pe coloane (cazul uzual), raportul rămâne acelaşi

Eliminare gaussiană

- Fie A o matrice $N \times N$
- Algoritmul de eliminare gaussiană calculează transformările inferior triunghiulare elementare M_k, $k = 1 : N-1$, astfel încât

$$M_{N-1} \dots M_2 M_1 A = U$$

cu U inferior triunghiulară

- Reamintire: matrice inferior triunghiulară elementară

$$M_k = I_n - m_k e_k^T = \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & 0 \\ & & \ddots & & \dots & \\ 0 & 0 & \dots & 1 & \dots & 0 \\ 0 & 0 & \dots & -\mu_{k+1,k} & \dots & 0 \\ & & \dots & \dots & \ddots & 0 \\ 0 & 0 & \dots & -\mu_{nk} & \dots & 1 \end{bmatrix}$$

Algoritmul de eliminare gaussiană

- Algoritmul se efectuează pe loc în matricea A
- Algoritmul calculează implicit şi factorizarea $A = LU$
- În final, U ocupă triunghiul superior (inclusiv diagonala), iar L triunghiul inferior (elementele diagonale ale lui L sunt egale cu 1 şi deci nu trebuie memorate)

1. `Pentru` $k = 1 : N - 1$
    1. `Pentru` $i = k + 1 : N$
        1. $a_{ik} \leftarrow \mu_{ik} = a_{ik}/a_{kk}$
    2. `Pentru` $j = k + 1 : N$
        1. `Pentru` $i = k + 1 : N$
            1. $a_{ij} \leftarrow a_{ij} - \mu_{ik} a_{kj}$

- Număr de operaţii: $2N^3/3$

Algoritm vectorial

- Algoritmul e adecvat în mod natural calculatoarelor vectoriale
- Bucla 1.1 reprezintă împărţirea unui vector cu un scalar, deci o operaţie SSCAL (vezi BLAS-1)
- Bucla 1.2.1 este o operaţie de tip SAXPY

1. `Pentru` $k = 1 : N - 1$
    1. $A(k+1:N, k) \leftarrow m_k = A(k+1:N, k)/A(k,k)$
    2. `Pentru` $j = k + 1 : N$
        1. $A(k+1:N, j) \leftarrow A(k+1:N, j) - A(k, j)m_k$

- Toate operaţiile sunt vectoriale
- Lungimea vectorilor scade de la $N - 1$ (la început, $k = 1$) până la 1
- Vectorul m_k este refolosit

Factorizare LU la nivel de bloc

- Partiţionăm matricea A

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} \} r \\ \} N-r \end{matrix}$$
$$\underbrace{\phantom{A_{11}}}_{r} \quad \underbrace{\phantom{A_{12}}}_{N-r}$$

- Începem prin a calcula prima bloc linie şi coloană a factorizării, adică L_{11}, L_{21}, U_{11}, U_{12}, astfel încât

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \cdot \begin{bmatrix} I_r & 0 \\ 0 & B \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix} \quad (1)$$

- Matricea B va fi utilizată ulterior pentru continuarea factorizării

Detaliile primului pas

- $A_{11} = L_{11}U_{11}$, deci L_{11} şi U_{11} provin din factorizarea LU la nivel de element a matricei A_{11}
- $A_{21} = L_{21}U_{11} \Rightarrow L_{21} = A_{21}U_{11}^{-1}$. Deci, U_{11} fiind cunoscut de la pasul anterior, L_{21} poate fi calculat prin rezolvarea unui sistem superior triunghiular cu parte dreaptă multiplă
- $A_{12} = L_{11}U_{12} \Rightarrow U_{12} = L_{11}^{-1}A_{12}$. Deci, U_{12} este soluţia unui sistem inferior triunghiular cu parte dreaptă multiplă
- $A_{22} = L_{21}U_{12} + B \Rightarrow B = A_{22} - L_{21}U_{12}$; blocul "restant" B depinde doar de matrice cunoscute sau deja calculate
- Astfel se calculează prima linie şi coloană a factorizării

Continuare

- Continuăm prin a calcula factorizarea LU a matricei B

$$B = L_{22}U_{22}$$

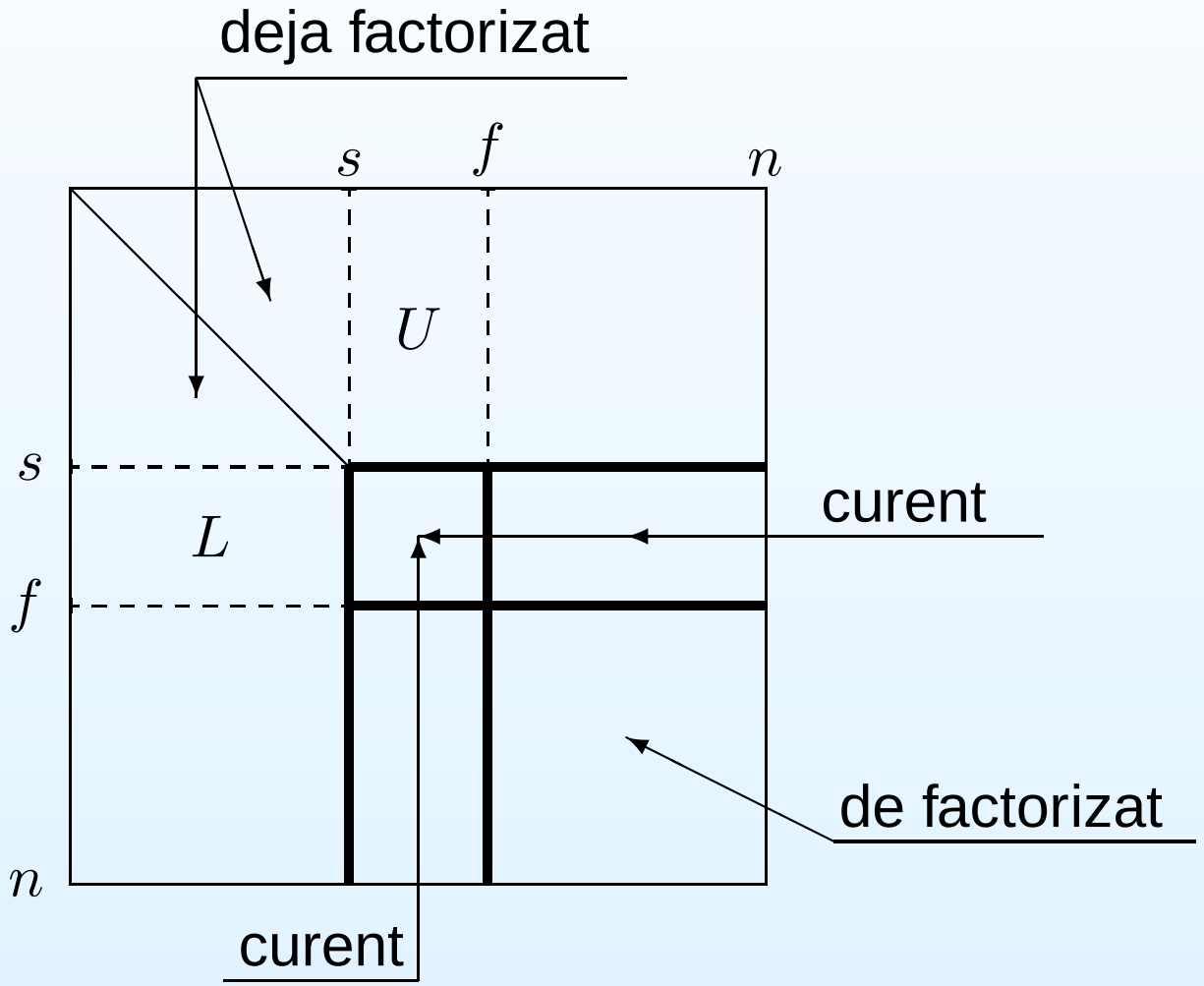
- În acest caz, egalitatea (1) devine o factorizare LU a matricei A, cu

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}, \qquad U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

- Aplicând în mod repetat cei patru paşi de mai sus, dimensiunea problemei se reduce de la N la $N - r$, $N - 2r$, etc.

Structura intermediară a matricei A

- Înaintea unui pas al algoritmului, conţinutul matricei A are structura



Algoritmul de factorizare

- Algoritmul complet ($n = \lceil N/r \rceil$):
  1. `Pentru` $k = 1 : n$
     1. $s \leftarrow (k-1)r + 1$
     2. $f \leftarrow \min(kr, N)$
     3. Se calculează factorizarea LU
        $$A(s:f, s:f) = L(s:f, s:f) \cdot U(s:f, s:f)$$
     4. Se rezolvă sistemul superior triunghiular
        $$Z \cdot U(s:f, s:f) = A(f+1:N, s:f)$$
     5. $L(f+1:n, s:f) \leftarrow Z$
     6. Se rezolvă sistemul inferior triunghiular
        $$L(s:f, s:f) \cdot Z = A(s:f, f+1:N)$$
     7. $U(s:f, f+1:n) \leftarrow Z$
     8. $A(f+1:n, f+1:n) \leftarrow A(f+1:n, f+1:n) -$
        $$-L(f+1:n, s:f)U(s:f, f+1:n)$$

Comentarii

- Apeluri la BLAS-3:
  - TRSM: instrucţiunile 1.4, 1.6
  - GEMM: instrucţiunea 1.8
- Într-o implementare dedicată pe calculatoare cu memorie ierarhică
  - Z e variabilă în MR
  - Factorizarea LU din 1.3 se poate implementa eficient, având $2r^3/3$ operaţii pentru r^2 accese la memorie
- De obicei LAPACK este scris în limbaj de nivel înalt, deci pondere operaţiilor de nivel 3 este

$$P_3(N,r) = \frac{2N^3/3 - 2nr^3/3}{2N^3/3} = 1 - \frac{r^2}{N^2}$$

- Dacă N e mare, $P_3(N,r) \approx 1$

Cum se alege r ?

- Numărul de operaţii în algoritmul la nivel de bloc este $2N^3/3$, deci nu e influenţat de r
- Compromis: r mare favorizează BLAS-3, dar scade $P_3(N, r)$
- Paralelismul creşte pe măsură ce r scade
- Explicaţii mai multe pe cale orală . . .

Introducerea pivotării parţiale

- La fiecare pas al algoritmului scalar, elementul maxim în valoare absolută de pe coloana curentă este adus în poziţie diagonală prin permutare de linii
- Algoritmul de eliminare gaussiană calculează

$$M_{N-1}P_{N-1}\ldots M_2P_2M_1P_1A = U$$

  unde P_k sunt matrice elementare de permutare
- Aceasta este echivalent cu calculul unei factorizări $PA = LU$, cu $P = P_{N-1}\ldots P_2P_1$
- Exerciţiu: scrierea algoritmului (GPP)
- Atenţie la calculul lui L: permutările se aplică pe întreaga linie a lui A, nu doar pe blocul dreapta-jos

Primul pas

- Introducerea pivotării în primul pas revine la

$$\tilde{P}_1 \cdot \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \cdot \begin{bmatrix} I_r & 0 \\ 0 & B \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix}$$

- $\tilde{P}_1 = P_r \ldots P_1$, deci matricea $\tilde{P}_1$ cumulează pivotările din primii r paşi ai algoritmului scalar

Detaliile primului pas (1)

- Se calculează prin eliminare gaussiană factorizarea LU pentru cele două blocuri din stânga:

$$\tilde{P}_1 \cdot \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \cdot U_{11}$$

  Se aplică algoritmul GPP, chiar dacă matricea în cauză este $n \times r$; căutarea pivotului se face pe toată porţiunea subdiagonală a unei coloane

- Se aplică permutarea restului matricei A (cele două blocuri din dreapta), obţinându-se

$$\begin{bmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{bmatrix} = \tilde{P}_1 \cdot \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}$$

Detaliile primului pas (2)

- Din $\tilde{A}_{12} = L_{11}U_{12}$ se poate calcula $U_{12} = L_{11}^{-1}\tilde{A}_{12}$, prin rezolvarea unui sistem inferior triunghiular cu parte dreaptă multiplă (se apelează TRSM)
- Mai rămâne $B = \tilde{A}_{22} - L_{21}U_{12}$, termenii din dreapta fiind toţi deja calculaţi; deci B se poate obţine în urma unui apel la GEMM
- Exerciţiu: algoritm detaliat !
- Ponderea operaţiilor de nivel 3 este mai mică decât în lipsa pivotării, datorită necesităţii lucrului cu operaţii scalare pe o matrice $N \times r$

Factorizare LU, versiunea Crout

- Partiţionăm toate matricele în blocuri $r \times r$

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & \dots & L_{nn} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & \dots & U_{1n} \\ 0 & U_{22} & \dots & U_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & U_{nn} \end{bmatrix}$$

- Rezultă egalităţile

$$A_{ij} = \sum_{t=1}^{\min(i,j)} L_{it} U_{tj}$$

Egalitatea $A = LU$ detaliată

- Distingem următoarele cazuri

$$i = j = k \;\Rightarrow\; A_{kk} = \sum_{t=1}^{k} L_{kt}U_{tk} \;\Rightarrow\; L_{kk}U_{kk} = \left(A_{kk} - \sum_{t=1}^{k-1} L_{kt}U_{tk} \right) \tag{2}$$

$$i > j = k \;\Rightarrow\; A_{ik} = \sum_{t=1}^{k} L_{it}U_{tk} \;\Rightarrow\; L_{ik} = \left(A_{ik} - \sum_{t=1}^{k-1} L_{it}U_{tk} \right) U_{kk}^{-1} \tag{3}$$

$$k = i < j \;\Rightarrow\; A_{kj} = \sum_{t=1}^{k} L_{kt}U_{tj} \;\Rightarrow\; U_{kj} = L_{kk}^{-1} \left(A_{kj} - \sum_{t=1}^{k-1} L_{kt}U_{tj} \right) \tag{4}$$

Algoritm Crout, prima versiune

- Calculăm în ordine, coloanele lui L şi liniile lui U
    1. `Pentru` $k = 1 : n$
        1. Calculează L_{kk} şi U_{kk} factorizând LU termenul drept din (2)
        2. `Pentru` $i = k + 1 : n$
            1. Calculează L_{ik} ca în (3)
        3. `Pentru` $j = k + 1 : n$
            1. Calculează U_{kj} ca în (4)
- Suma din (3) se poate calcula printr-un singur GEMM
- Mai mult, toată bucla 1.2 poate fi compactată
- Notând $s = (k-1)r + 1$, $f = kr$, bucla 1.2 se scrie

$$L(f{+}1 : N, s : f) \leftarrow [A(f{+}1 : N, s : f) - L(f{+}1 : N, 1 : s{-}1) \cdot U(1 : s{-}1, s : f)] \cdot U_{kk}^{-1}$$

- Se procedează similar cu bucla 1.3

Algoritm Crout de factorizare LU la nivel de bloc

- Algoritmul complet

1. `Pentru` $k = 1 : n$
    1. $s \leftarrow (k-1)r + 1$
    2. $f \leftarrow \min(kr, N)$
    3. $A(s:N, s:f) \leftarrow A(s:N, s:f) - L(s:N, 1:s-1) \cdot U(1:s-1, s:f)$
    4. Se calculează factorizarea LU
    $$A(s:f, s:f) = L(s:f, s:f) \cdot U(s:f, s:f)$$
    5. Se rezolvă sistemul superior triunghiular
    $$Z \cdot U(s:f, s:f) = A(f+1:N, s:f)$$
    6. $L(f+1:N, s:f) \leftarrow Z$ (o bloc coloană din L)
    7. $A(s:f, f+1:N) \leftarrow A(s:f, f+1:N) -$
    $$-L(s:f, 1:s-1) \cdot U(1:s-1, f+1:N)$$
    8. Se rezolvă sistemul inferior triunghiular
    $$L(s:f, s:f) \cdot Z = A(s:f, f+1:N)$$
    9. $U(s:f, f+1:N) \leftarrow Z$ (o bloc linie din U)

Comparaţii cu eliminarea gaussiană

- Şi aici L şi U se pot calcula pe loc în A
- Acelaşi număr de operaţii
- Aceeaşi pondere a operaţiilor de nivel 3
- Mai multe apeluri la GEMM (două în fiecare iteraţie), deci timp de execuţie eventual mai mic, datorită dimensiunilor mai mici ale matricelor
- Aici se actualizează blocurile curente din L şi U, după care acestea se calculează (left-looking)
- În eliminarea gaussiană, actualizările se făceau la dreapta, după calcularea blocurilor curente din L şi U

Triangularizare ortogonală

- Fie $A \in \mathbb{R}^{M \times N}$ o matrice dreptunghiulară, cu $M \geq N$
- Triangularizarea ortogonală

$$U_N \ldots U_2 U_1 A = R$$

- $R \in \mathbb{R}^{M \times N}$ este superior triunghiulară
- U_i sunt reflectori Householder

$$U_i = I - \tau_i u_i u_i^T$$

- $\tau_i = 2/\|u_i\|$, $u_i = [0 \ \ldots \ 0 \ u_{ii} \ \ldots \ u_{Mi}]^T$
- Reflectorul U_i se calculează astfel încât să introducă zerouri subdiagonale pe coloana i
- Pentru calculul reflectorilor în algoritmul de triangularizare, vezi cursul de Metode Numerice

Strategie pe blocuri

- Considerăm triangularizarea primelor r coloane ale lui A, utilizând transformarea $Q = U_r \ldots U_1$, numită reflector bloc
- Partiţionăm $A = [A_1\ B]$, unde $A_1 = A(1:M, 1:r)$, $B = A(1:M, r+1:N)$
- Structura primului pas
  1. Se generează Q a.î. $QA_1 = R_1$ este superior triunghiulară (calculând fiecare reflector U_i)
  2. Se formează $Q = U_r \ldots U_1$
  3. Se calculează $B \leftarrow QB$
- Se continuă similar pe blocul B

Dificultăţi şi soluţie: reflector bloc structurat

- Matricea Q are dimensiune $M \times M$
- Dacă o formăm explicit şi apoi calculăm QB, numărul de operaţii va fi mult prea mare
- Reamintire: complexitatea triangularizării ortogonale este $O(MN^2)$
- Produsul explicit QB ar necesita $O(M^2N)$ operaţii doar pentru primul pas
- Ideea de eficientizare: exprimarea matricei Q într-o formă structurată
  - Reprezentarea WY: $Q = I - WY^T$, cu $W, Y \in \mathbb{R}^{M \times r}$
  - Reprezentarea W^{2}T: $Q = I - WTW^T$, cu T inferior triunghiulară

De ce e bine ?

- Pasul 3 se calculează prin

$$B \leftarrow QB = (I - WY^T)B = B - W(Y^T B)$$

- Produsul $Y^T B$ necesită $Mr(N - r)$ operaţii
- La fel produsul $W(Y^T B)$
- În acest fel algoritmul de triangularizare ortogonală pe blocuri necesită doar un număr neglijabil de operaţii în plus faţă de cel la nivel de element
- Grosul operaţiilor este în înmulţirile de matrice
- Reprezentarea W^{2}T necesită mai puţină memorie (dar operaţii în plus pentru înmuţirea cu T)

Reprezentarea WY (1)

- Matricele W şi Y se construiesc iterativ
- Iniţial avem $Q = U_1 = I - \tau_1 u_1 u_1^T$, deci putem lua

$$W = u_1, \quad Y = \tau_1 u_1$$

- Presupunând Q deja în formă WY, înmulţirea cu un reflector la stânga se face prin

$$\begin{aligned} Q_+ = U_i Q &= (I - \tau_i u_i u_i^T)(I - WY^T) = \\ &= I - WY^T - \tau_i u_i u_i^T (I - WY^T) = \\ &= I - \begin{bmatrix} W & u_i \end{bmatrix} \begin{bmatrix} Y^T \\ z_i^T \end{bmatrix} = \\ &= I - W_+ Y_+^T \end{aligned}$$

unde $z_i = \tau_i (I - YW^T) u_i$

Reprezentarea WY (2)

- În concluzie, matricele W şi Y se construiesc prin

$$W \leftarrow [W\ u_i], \quad Y \leftarrow [Y\ z_i]$$

- Matricea W se obţine prin alăturarea vectorilor Householder (deja disponibili ca atare)
- Vectorii z_i care formează matricea Y se construiesc rapid, grupând $z_i = \tau_i[u_i - (YW^T)u_i]$ (matricele W şi Y au puţine coloane)
- Chiar dacă prin calculul lui z_i se fac mai multe operaţii decât în triangularizarea la nivel de element, grosul operaţiilor este în actualizările $B \leftarrow QB$, care se pot face acum apelând GEMM
- Matricea W se memorează esenţialmente în A
- Pentru Y e necesară memorie separată

Reprezentarea W^{2}T (1)

- Matricele W şi T se calculează iterativ
- Iniţial avem $Q = I - \tau_1 u_1 u_1^T$, deci putem lua

$$W = u_1, \quad T = \tau_1$$

- Presupunând Q deja în forma dorită, înmulţirea cu un reflector are forma

$$\begin{aligned} Q_+ = U_i Q &= (I - \tau_i u_i u_i^T)(I - WTW^T) = \\ &= I - WTW^T - \tau_i u_i u_i^T + u_i(\tau_i u_i^T WT)W^T = \\ &= I - \begin{bmatrix} W & u_i \end{bmatrix} \begin{bmatrix} T & 0 \\ t_i^T & \tau_i \end{bmatrix} \begin{bmatrix} W^T \\ u_i^T \end{bmatrix} = \\ &= I - W_+ T_+ W_+^T \end{aligned}$$

unde $t_i^T = -\tau_i u_i^T WT$

Reprezentarea W^{2}T (2)

- Matricele W şi Y se construiesc prin

$$W \leftarrow [W\ u_i], \quad T \leftarrow \begin{bmatrix} T & 0 \\ t_i^T & \tau_i \end{bmatrix}$$

- Matrice W are aceeaşi formă ca la reprezentarea WY
- Necesarul de memorie suplimentară este de r^2, pentru T
- Exerciţiu: scrieţi cât mai detaliat algoritmul de triangularizare ortogonală utilizând reprezentarea W^{2}T
- Exerciţiu: algoritm pentru factorizarea QR (în plus, se înmulţesc transformările ortogonale bloc)