

Seminar Multicore Architectures and Programming 08
am Lehrstuhl Informatik 12, Hardware-Software-Co-Design

Efficient Matrix Inversion in CUDA

Robert Grimm, Matthias Schneider

Friedrich-Alexander Universität
Erlangen-Nürnberg

11.07.2008

Outline

- 1 Motivation
 - Problemstellung

- 2 Ansätze
 - Gauss-Elimination
 - Bordering
 - Cholesky-Zerlegung
 - QR-Zerlegung
 - SART

- 3 Fazit
 - Zusammenfassung
 - Alternativen

Outline

- 1 Motivation
 - Problemstellung
- 2 Ansätze
- 3 Fazit

Matrixinversion

Das Problem

- Zu lösen: Mehrere Gleichungssysteme der Form $\mathbf{A} \cdot \vec{x} = \vec{b}$ mit verschiedenen rechten Seiten
- Idee: Berechne Inverse \mathbf{A}^{-1} : danach nur noch Matrix-Vektor-Multiplikation: $\vec{x} = \mathbf{A}^{-1} \cdot \vec{b}$.
- Allgemein: Berechnung von Matrix-Inversen **numerisch sehr bedenklich** - für uns aber nebensächlich.
- Beschränkung auf relativ kleine Matrizen ($A \in \mathbb{C}^{64 \times 64 \dots 256 \times 256}$).

Outline

1 Motivation

2 Ansätze

- Gauss-Elimination
- Bordering
- Cholesky-Zerlegung
- QR-Zerlegung
- SART

3 Fazit

Gauss-Eliminationsverfahren

Eliminationsverfahren von Gauss-Jordan

- Schema zu Beginn: $(\mathbf{A}|\mathbf{I})$, nach Gauss-Verfahren: $(\mathbf{I}|\mathbf{A}^{-1})$.

- $\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \vdots \\ \vdots & & \ddots & \\ a_{n,1} & \dots & & a_{n,n} \end{pmatrix}$

- Durch Operationen auf ganzen Zeilen: Für alle Spalten: jeweils nur Diagonaleintrag belassen, in allen anderen Zeilen 0en erzeugen
- Rechenaufwand $O(n^3)$, Operationen auf Zeilen der Breite 128 - sollte sich doch super eignen?
- keine Pivotisierung (verbessert zwar numerische Stabilität, aber schwerer parallelisierbar)

Gauss-Eliminationsverfahren

Naive Implementierung

- Nur global memory
- Nur ein thread block - jeder der 128 threads für eine Spalte zuständig:
 - 1 Iteriere über alle Zeilen:
 - 2 Teile alle Einträge der aktuellen Zeile i durch das Diagonalelement $a_{i,i}$ (Normalisierung)
 - 3 Iteriere über alle Zeilen $j \neq i$:
 - 4 Subtrahiere das $a_{j,i}$ -fache der Zeile i von Zeile j , um in der Spalte i eine 0 zu erzeugen

Naive Implementierung

Performance

- Natürlich extrem langsam
- Speicherzugriffe zwar coalesced, aber pro Spaltenupdate muss die komplette Matrix eingelesen und wieder in den Speicher geschrieben werden.
- Lösung: na klar, Shared Memory!

Gauss-Eliminationsverfahren

Bessere Implementierung

- Implementierung mit Shared Memory
- Speicherbedarf: 128×128 Komplexzahlen (2×4 Byte). **viel zu viel :(**
- Dann laden wir eben nur eine "Kachel" oder wenige Zeilen/Spalten und diagonalisieren erstmal die.
- **Und wann werden die 0en in den anderen Zeilen erzeugt?**
- Dafür braucht man die (normalisierte) Referenz-Zeile bzw. Koeffizienten der Subtraktion.
- **Keine Möglichkeit zu inter-block Synchronisation bzw. "Broadcast"**

Gauss-Eliminationsverfahren

Bessere Implementierung

- Implementierung mit Shared Memory
- Speicherbedarf: 128×128 Komplexzahlen (2×4 Byte). **viel zu viel :(**
- Dann laden wir eben nur eine “Kachel” oder wenige Zeilen/Spalten und diagonalisieren erstmal die.
- Und wann werden die 0en in den anderen Zeilen erzeugt?
- Dafür braucht man die (normalisierte) Referenz-Zeile bzw. Koeffizienten der Subtraktion.
- Keine Möglichkeit zu inter-block Synchronisation bzw. “Broadcast”

Gauss-Eliminationsverfahren

Bessere Implementierung

- Implementierung mit Shared Memory
- Speicherbedarf: 128×128 Komplexzahlen (2×4 Byte). **viel zu viel :(**
- Dann laden wir eben nur eine “Kachel” oder wenige Zeilen/Spalten und diagonalisieren erstmal die.
- **Und wann werden die 0en in den anderen Zeilen erzeugt?**
- Dafür braucht man die (normalisierte) Referenz-Zeile bzw. Koeffizienten der Subtraktion.
- Keine Möglichkeit zu inter-block Synchronisation bzw. “Broadcast”

Gauss-Eliminationsverfahren

Bessere Implementierung

- Implementierung mit Shared Memory
- Speicherbedarf: 128×128 Komplexzahlen (2×4 Byte). **viel zu viel :(**
- Dann laden wir eben nur eine “Kachel” oder wenige Zeilen/Spalten und diagonalisieren erstmal die.
- **Und wann werden die 0en in den anderen Zeilen erzeugt?**
- Dafür braucht man die (normalisierte) Referenz-Zeile bzw. Koeffizienten der Subtraktion.
- **Keine Möglichkeit zu inter-block Synchronisation bzw. “Broadcast”**

Gauss-Eliminationsverfahren

Bessere Implementierung

- Shared Memory nur eingeschränkt nutzbar:
 - 1 Lade "Source"-Zeile i in Shared Memory \mathbf{R} (1 KB)
 - 2 Teile Zeile i durch $a_{i,i}$
 - 3 Lade nacheinander alle "Target"-Zeilen j , $j \neq i$ in Shared Memory \mathbf{S} (1 KB)
 - 4 Subtrahiere $a_{j,i}$ -faches der Zeile \mathbf{R} von \mathbf{S}
 - 5 Schreibe \mathbf{S} wieder zurück in globalen Speicher
- Optimierung: "in-place", daher Speicherbedarf für eine Matrix-Zeile nur 128, aber benötigt aber eine zusätzliche Spalte mit Koeffizienten im Shared Memory.

Elimination mit Shared Memory

Performance

- Nur geringfügig besser als naiver Ansatz
- Für 100 parallele Inversionen gleichzeitig: Knapp 5mal langsamer als unoptimierte Elimination auf CPU.
- Praktisch keine Kommunikation zwischen den Threads innerhalb eines Blocks
- Berechnung von $\frac{1}{a_{i,i}}$, mit dem jeder Thread "seinen" Wert multipliziert: Müsste eigentlich nicht jeder Thread selbst machen, aber branch würde sonst divergieren.

Bordering Methode

Konzept

- Angenommen, die Inverse einer $(k - 1) \times (k - 1)$ Matrix \mathbf{A}_{k-1} ist bekannt
- Dann lässt sich daraus die Inverse der $k \times k$ Matrix \mathbf{A}_k berechnen, die man erhält, wenn man \mathbf{A}_{k-1} um eine Zeile/Spalte ergänzt.
- Beginne mit $k = 1$:
$$\mathbf{A}_1 = (a_{1,1}); \mathbf{A}_1^{-1} = \left(\frac{1}{a_{1,1}} \right).$$
- Konstruiere daraus \mathbf{A}_2^{-1} , usw. bis \mathbf{A}_k^{-1} .

Bordering Schema

Rekursive bzw. iterative Rechenvorschrift

$$\mathbf{A}_k = \begin{pmatrix} \mathbf{A}_{k-1} & \vec{u} \\ \vec{v}^T & a_{k,k} \end{pmatrix};$$

$$\mathbf{A}_k^{-1} = \begin{pmatrix} \mathbf{A}_{k-1}^{-1} & \vec{0} \\ \vec{0}^T & 0 \end{pmatrix} + p \cdot \begin{pmatrix} \mathbf{A}_{k-1}^{-1} \cdot \vec{u} \\ -1 \end{pmatrix} \cdot (\vec{v}^T \cdot \mathbf{A}_{k-1}^{-1} \quad -1),$$

mit

$$p = p_k = (a_{kk} - \vec{v}^T \mathbf{A}_{k-1}^{-1} \vec{u})^{-1}.$$

Bordering

Analyse

$$\bullet \mathbf{A}_k^{-1} = \begin{pmatrix} \mathbf{A}_{k-1}^{-1} & \vec{0} \\ \vec{0}^T & 0 \end{pmatrix} + p \cdot \begin{pmatrix} \mathbf{A}_{k-1}^{-1} \cdot \vec{u} \\ -1 \end{pmatrix} \cdot (\vec{v}^T \cdot \mathbf{A}_{k-1}^{-1} \quad -1)$$

$$p = p_k = (a_{kk} - \vec{v}^T \mathbf{A}_{k-1}^{-1} \vec{u})^{-1}.$$

- Eigentlich nur lineare Algebra (Matrix-Addition, Matrix-Vektor / Vektor-Vektor Multiplikationen) - lässt sich i.A. gut parallelisieren
- Problem: Dimensionen zwischen 1 und 128 - da lohnt sich kein *CUBLAS*-Aufruf.
- Operationen mit Vektoren, deren Länge nicht ein Vielfaches von 32 ist, meist ineffizient.

Cholesky-Zerlegung

Übersicht

- Spezialfall der LU-Zerlegung
- Faktorisierungsmethode für symmetrisch positiv definite Matrizen

$$A = L \cdot L^H$$

L : untere Dreiecksmatrix

- Lösung von $Ax = b$ erhält man durch je einmal vorwärts (1) und rückwärts (2) einsetzen

① $L \cdot y = b$

② $L^H \cdot x = y$

(ohne explizite Berechnung der Inversen!)

Cholesky-Zerlegung

Input: $X \in \mathbb{C}^{n \times n}$ transponiert komplexe Matrix

```

// init L with lower left part of X
L=zeros(n);
for i=1:n
    L(i:n,i)=X(i:n,i);
end

for k=1:n
    // normalize kth column
    L(k:n,k)=L(k:n,k)/sqrt(L(k,k));

    // outer product iteration
    for i=(k+1):n
        L(i:n,i)=L(i:n,i) - L(i,k)*L(i:n,k);
    end
end

```

Output: $L \in \mathbb{C}^{n \times n}$ untere Dreiecksmatrix mit $X = L \cdot L^H$

Cholesky-Zerlegung

Feststellungen & Probleme:

- Äußeres Produkt in k -ter Iteration verändert alle Einträge des unteren Dreiecks in den verbleibenden Spalte $k + 1 - n$
- Berechnung des äußeren Produkts benötigt normalisierte Einträge der k -ten Spalte
- Äußere for-Schleife lässt sich also nicht parallelisieren

Lösung:

- Parallelisierung der Normalisierung und des äußeren Produkts mit Kernel auf 1×1 -Grid
- Multiprozessoren der Hardware alles andere als ausgelastet

⇒ Schlechte Lösung

Cholesky-Zerlegung

Feststellungen & Probleme:

- Äußeres Produkt in k -ter Iteration verändert alle Einträge des unteren Dreiecks in den verbleibenden Spalte $k + 1 - n$
- Berechnung des äußeren Produkts benötigt normalisierte Einträge der k -ten Spalte
- Äußere for-Schleife lässt sich also nicht parallelisieren

Lösung:

- Parallelisierung der Normalisierung und des äußeren Produkts mit Kernel auf 1×1 -Grid
- Multiprozessoren der Hardware alles andere als ausgelastet

⇒ Schlechte Lösung

QR-Zerlegung

Übersicht

- Faktorisierungsmethode

$$A = Q \cdot R$$

Q : orthonormale Matrix, R : obere Dreiecksmatrix

- Lösung von $Ax = b$ erhält man durch rückwärts einsetzen

$$R x = Q^T \cdot b$$

(ohne explizite Berechnung der Inversen!)

QR-Zerlegung

Der Weg zur Faktorisierung (Numerical Recipes)

- Sukzessive Erzeugung von Nulleinträgen unterhalb der Diagonale durch geeignete Householderspiegelungen
- $A^{(0)} = A \in \mathbb{C}^{m \times n}$
 $A^{(k+1)} = Q^{(k)} \cdot A^{(k)}$
- $A^{(k)}$ besitzt in den ersten k Spalten nur Nulleinträge unterhalb der Diagonale

$$\forall (i, j) : j < k \wedge j < i \implies a_{ij}^{(k)} = 0$$

- $A^{(n)}$ ist also obere Dreiecksmatrix mit

$$\underbrace{A^{(n)}}_R = \underbrace{Q^{(n-1)} \dots Q^{(1)} Q^{(0)}}_{Q^T} \cdot A$$

QR-Zerlegung

Die Q -Matrix erhält man also als Produkt von insgesamt n Matrizen $Q^{(i)}$:

$$\underbrace{A^{(n)}}_R = \underbrace{Q^{(n-1)} \dots Q^{(1)} Q^{(0)}}_{Q^T} \cdot A$$

Problem:

- Berechnung der Matrixprodukte übersteigt Berechnungszeit für vollständige Invertierung auf CPU
- Berechnungszeiten für das Aufstellen der $Q^{(i)}$ und das rückwärts einsetzen dabei noch vollkommen unberücksichtigt

Simultaneous Algebraic Reconstruction (SART)

Gleichungssysteme mal anders ...

- Verfahren zum Lösen von Gleichungssystem

$$A \cdot x = b$$

- Jede Gleichung definiert Hyperebene H_i mit Normale a_i :

$$a_i \cdot x = b_i$$

- Lösung = Schnitt der Hyperebenen

$$\hat{x} = \bigcap_{i=0}^{N_a-1} H_i$$

Simultaneous Algebraic Reconstruction (SART)

Lösungsansatz

- 1 Initialisierung des Lösungsvektors $\hat{x}^{(0)}$
- 2 Orthogonale Projektion auf alle Hyperebenen
- 3 Berechnung der neuen Schätzung aus projizierten Vektoren (z.B. Mittelwert)

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \lambda_k \sum_{i=0}^{N_a-1} u_{k,i} \cdot \frac{b_i - a_i^T \hat{x}^{(k)}}{a_i^T a_i} a_i$$

$$\sum_{i=0}^{N_a-1} u_{k,i} = 1$$

λ_k : Relaxationsparameter

Simultaneous Algebraic Reconstruction (SART)

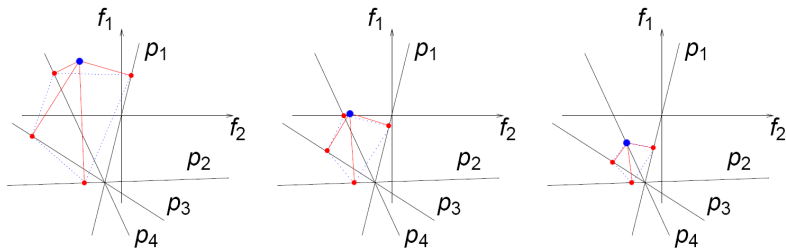


Abbildung: SART-Iterationschema mit $u_{i,k} = 1/N_a$ und $\lambda = 1$.

Quelle: Skript zur Vorlesung "Digital Medical Image Processing" (LME)

Simultaneous Algebraic Reconstruction (SART)

Vereinfachtes Schema:

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \frac{1}{N_a} \sum_{i=0}^{N_a-1} \frac{b_i - a_i^T \hat{x}^{(k)}}{a_i^T a_i} a_i$$

Parallelisierung (Versuch 1)

- 1 Jeder Thread berechnet die Projektion von $x^{(k)}$ auf eine Hyperebene H_{TID} .
- 2 Synchronisation
- 3 Berechnung des neuen Schätzwertes $x^{(k+1)}$ aus Projektionen
- 4 Wiederholen der Schritte 1 bis 3 bis Konvergenz bzw. maximale Iterationsanzahl erreicht

Simultaneous Algebraic Reconstruction (SART)

Vereinfachtes Schema:

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \frac{1}{N_a} \sum_{i=0}^{N_a-1} \frac{b_i - a_i^T \hat{x}^{(k)}}{a_i^T a_i} a_i$$

Parallelisierung (Versuch 1)

Probleme:

- nur ein aktiver Block (VIEL zu wenig)
- Threads vollkommen überlastet (Skalarprodukte als for-loop)

Lösung:

- Verteilung der Arbeit auf mehrere Blöcke
- Skalarprodukte parallel mit mehreren Threads berechnen

Simultaneous Algebraic Reconstruction (SART)

Vereinfachtes Schema:

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \frac{1}{N_a} \sum_{i=0}^{N_a-1} \frac{b_i - a_i^T \hat{x}^{(k)}}{a_i^T a_i} a_i$$

Parallelisierung (Versuch 2)

- Jeder Block berechnet die Projektion von $x^{(k)}$ auf eine Hyperebene H_{BID} .
- **Synchronisation**
- Berechnung des neuen Schätzwerts $x^{(k+1)}$ aus Projektionen
- Wiederholen der Schritte 1 bis 3 bis Konvergenz bzw. maximale Iterationsanzahl erreicht

Simultaneous Algebraic Reconstruction (SART)

Vereinfachtes Schema:

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \frac{1}{N_a} \sum_{i=0}^{N_a-1} \frac{b_i - a_i^T \hat{x}^{(k)}}{a_i^T a_i} a_i$$

Parallelisierung (Versuch 2)

Problem:

- Synchronisation über Blockgrenzen hinweg nicht möglich

Lösung:

- “Manuelle” Synchronisation über Kernelaufrufe

Simultaneous Algebraic Reconstruction (SART)

Zu früh gefreut: Auch nichts tun kostet Zeit...

Problem:

- Jede Iteration erfordert mindestens zwei Kernelaufrufe
- Mehrere hundert Iterationen benötigt für akzeptable Lösung
- Aufrufzeit von sinnfreien, leeren Kernels (leerer Rumpf, keine Argumente, kein shared memory) übersteigt bereits die Zeit für komplette Invertierung auf CPU

Lösung:

- CPU-Ansatz ⇒)

Outline

- 1 Motivation
- 2 Ansätze
- 3 Fazit
 - Zusammenfassung
 - Alternativen

Das hätten wir gebraucht

- Gauss-Elimination:
 - Schnellerer / breiterer globaler Speicherbus
 - Mehr Shared Memory
 - Globale Synchronisation
- Bordering:
 - Effiziente lineare Algebra mit kleinen Vektoren beliebiger Länge
- SART:
 - Schnellerer / breiterer globaler Speicherbus
 - Globale Synchronisation


Alternativen




Matrix-Invertierung allgemein

- kleinere Matrizen: passen in den Shared Memory
- grössere Matrizen: Aufwand für Zerlegung lohnt sich (Cholesky auf Tesla: ca. 75 GFLOPs bei 5000×5000)
in der Literatur: Matrix-Dimensionen in Timing-Diagrammen beginnen meist erst bei 500-1000... [3, 1]
- LU-Zerlegung: Bei ausreichenden Dimensionen 2x schneller als auf CPU [3], Pivottisierung bremst etwas

Fragen?

Vielen Dank für die Aufmerksamkeit!

-  M. C. Pease, *Matrix Inversion Using Parallel Processing*, Journal of the Association for Computing Machinery, Vol. 13, No. 4, October 1967, pp. 757-764
-  Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press, New York, 1992, pp.98-102.
-  N. Galoppo, N. K. Govindaraju, M. Henson, D. Manocha, *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*, November 2005, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing

-  M. Castillo *et al.*, *Making Programming Synonymous with Programming for Linear Algebra Libraries*, SC08, Austin, Texas, November 2008.
-  G. H. Golub and Ch. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore and London, 3 edition, 1996.
-  A.C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, 1988.