Ausarbeitung zur Projektarbeit „Wissenschaftliches Programmieren mit Cuda"

Guanhua Bai, Achim Grolms und Buyu Xiao

***Abstract*—Dieser Text beschreibt unsere Erfahrungen und Ergebnisse bei der Implementierung des IDR(s) [1] Algorithmus auf CUDA-fähigen Grafikkarten des Herstellers Nvidia mit doppelter Fließkommagenauigkeit im Rahmen einer vierteljährlichen Projektarbeit an der Universität Paderborn im Winter 2009/2010**

I. Einleitung

DAS Lösen großer linearer Gleichungsysteme tritt häufig auf im Zusammenhang mit Feldtheoretischen Problemen. Grafikkarten (GPU) mit vielen parallelen Prozessoren sind geeignet als Rechenhardware um die rechnergestützte Lösung dieser LGS zu beschleunigen. Die Hersteller ATI und NVIDIA bieten SDK an speziell für die Entwicklung mit diesem Einsatzzweck. CUDA [2] ist ein API für GPU von Nvidia für die Entwicklung in C und C++.

Matlab bietet zum Einbinden eigenen C-Codes das API 'Mex' an.

II. „Theorie"

A. Operationen

Der iterative Löser IDR(s) [1] besteht aus Operationen [3] der linearen Algebra, im einzelnen in Tabelle I beschrieben.

TABLE I: Im IDR(s) verwendete Operationen

Operation	Zusammenhang	Bemerkung
add	$\mathbf{c} = \mathbf{a} + \mathbf{b}$	
dotmul	$s = \mathbf{a} \cdot \mathbf{b}$	
norm	$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$	
matrixmul	$\mathbf{c} = \mathbf{A_{full}} \cdot \mathbf{b}$	$\mathbf{A_{full}}$ vollbesetzt
sparsemul	$\mathbf{c} = \mathbf{A_s} \cdot \mathbf{b}$	$\mathbf{A_s}$ dünnbesetzt
solvergauss	$\mathbf{M} \cdot \mathbf{m} = \mathbf{c}$	LGS aus Zeile 36 [3]

$\mathbf{A_s}$ wird im Speicher dargestellt durch das „Sparse Matrix"-Speicherformat aus Matlab. [4]

B. Implementierung der Operationen

Blocks sind Gruppen von Threads die auf einem einzigen Multiprozessor laufen und gemeinsam das schnelle on-chip „shared Memory" des Multiprozessors nutzen können. [2] Die Operationen sind in doppelter Fließkommagenauigkeit implementiert.

Guanhua Bai, Achim Grolms und Buyu Xiao sind Studenten an der Universität Paderborn im Fachgebiet „Theoretische Elektrotechnik"

1) Punktprodukt: Im Abb. 1 wird das Implementierungsmethode in CUDA gezeigt. A und b sind Vektoren, die sich miteinander in jeweiligem Thread multipliziert werden müssen. cs sind erzeugte Produktvektoren, die allen zu einem Wert summiert werden. Da eine Beschrankung des CUDA-Blocksizes maximum 512 Threads enthält, muss ein Block iterativ oder mehrere Blocks verwendet werden.

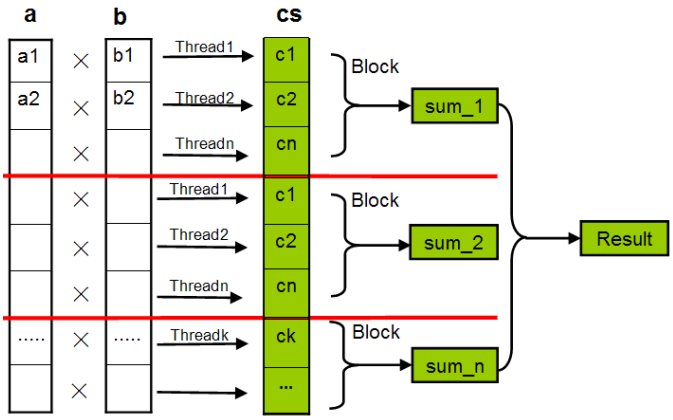


Fig. 1: Vektor Multiplikation. A : Erst Vektor; b: Zweiter Vektor; cs: Produktvektor

2) Multiplikation von Matrix mit Vektor in CUDA Implementierung: Beim Wissenschaftlichen Rechen trifft man häufig die Multiplikation von Matritze mit Vektor. In CUDA Implementierung wird die Oparation als unterschiedliche Vektor-Multiplicationen zerlegt.Mit änlichem Methode werden auch Matizen mit Vektoren multipliziert. Im folgendem Bild Abb.2.zeigt, dass jede zerlegte Vektor von A mit Vektor b in einem Block multipliziert wird.

3) Sparse Matrix und Vektormultiplikation: Sparsematrix, oder dünnbesetzte Matrix, bezeichnet man als eine Matrix, bei der so viele Einträge aus Nullen bestehen. Im Abb.3 wird ein einfaches Beispiel gezeigt. Da Sparsematrix mit Vollmatrix genau umgekehrt ist, hat man dafür auch eine andere Speicherweise. Unter dem Zusammenhang zwischen Abb.3a., Abb.3b. und Abb.3c. versteht man, dass bei der Sparsematrizen wird nun nur die Nonzero-Elemente und die zugehörigen Stelleinformationen(Zeilen und Spalte) gespeichert. Vektor pr enthält alle Nonzero-Elemente. Die Vektoren ir und jc enthalten die Zeileninformation und die Spalteinformation. Im Abb.3c. bezeichnet man, wie die Informationen gepackt werden. Die Werte von $jc[i]$ und $jc[i+1]-1$ zeigen den Indexe, deren die zur Spalte i gehörteten Nonzero-Elemente und Zeileinformation aufweisen.

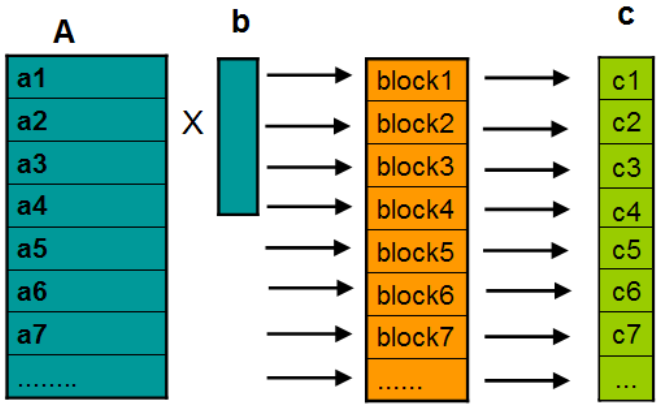
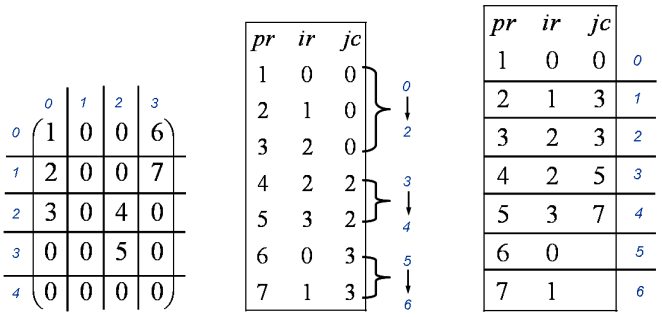


Fig. 2: Matrix mal Vektor. A: Matrix; b: Vektor; c: Produktvektor



(a) Original-Matrize (b) Sparsame Struktur (c) "compressed column structure"

Fig. 3: Sparse Matrix

Mit obengenanntem Methode werden Spasematrizen in Spaltfolg gespeichert. Bei unser Implementierung verwenden wir es als Zeilfolg. Im folgendem Abb.4. zeigt, wie die Multiplikation ausgeführt wird

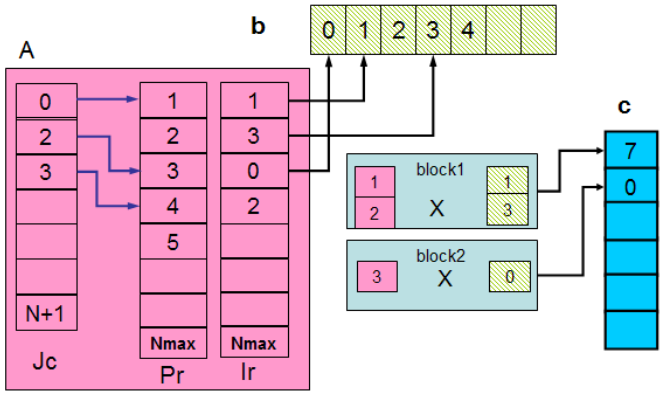


Fig. 4: Sparse Matrizenmultiplikation. A:Sparsematrize, Jc:Vektor der Zeileinfo, Pr: Vecktor der Nonzeroelements, Ir: Vektor der Spalteninfo ; b: Vektor; c: produkt Vektor

4) Löser Gauss: Basis für die CUDA Implementierung ist ein Algoritumus aus der Standardliteratur [8] mit Pivotisierung. Da Teile des Algorithmus (Rückwärtssubstitution, Maximum suchen bei der Povitisierung) nur sequentiell ausgeführt werden können fehlt die volle Ausnutzung der Parallelität.

III. Typische Probleme bei der GPU-Programmierung

Die besondere Architektur der GPU führt zu besonderen Problemen und Ansätzen zur Problemlösung.

A. Dreieckförmige Summation

Ein typisches Problem ist Bloksummation. Aus der Beschreibungen der Operationen Multiplikationen der Matrix mal Vektor und Sparsematrize mal Vektor beruhen obige Operationen auf Vektormultiplikationen, die schließlich ein Summierungsverfahren in jedem Block enthalten. Blocksummation in einzigen Thread ist nicht effizient. Die einführende Algorithmus: Dreieckförmige Summation lautet wie Fig.5 . In erst Schritte werden 2n und 2n+1 Elements des Produktvektors cs in jeweilig Threads summiert. In zweiter Schritte werden 4n und 4n+2 Elements summiert. Bis BlockSize/2 Schritte erhalt man endlich Ergebnisse.

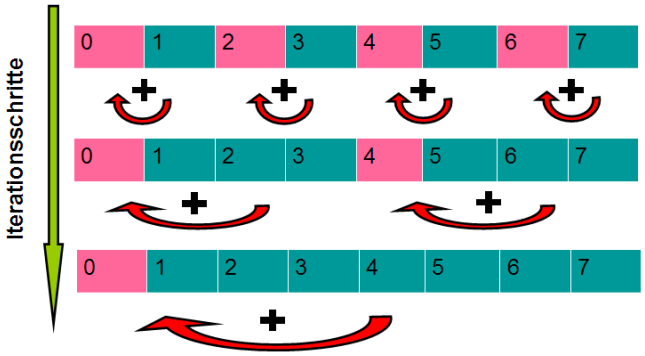


Fig. 5: Dreieckförmige Summation. Von Oben nach Unter zeigt

Beispiel Code kann man in [10] finden.

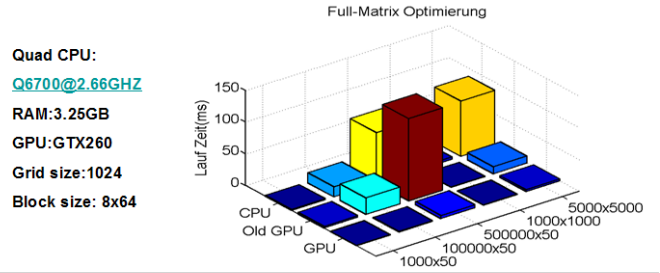
B. Minimierung leer laufende Thread

In CUDA bearbeitet jeder Multiprozessor gleichzeitig mit 32 Threads [2], die allen zum selben Block gehören. Bei der Sparematrix-Multiplikation sind viele Threads am leerlaufen. Um die Leerlaufenden Threads zu minimieren, müssen mehrere Punktprodukte in einem Block bearbeitet werden. Dazu verwendet man 2 dimensionierte Blocksizes. Die Definition und Anwendung von 2-D Blocksize findet man in der Referenz[2] und [6].

C. Shared Memory

In der Grafikkate ist der Zugriff auf den globalen Speicher langsamer als auf den On-Chip-Speicher. Wie Beispiele in [2] gezeigt, kann man mehrer mal verwendete Daten zunächst in shared Memory schreiben, dann für die entsprechenden Operationen benutzen. In der Multiplikation der Vollmatrize mal Vektor wird jede Vektorelement mehr mal gebraut. Nach Untersuchungen wählen wir 1-Dimensionblock,die 64 beträgt und jede Vektorelement 8 mal gebraucht in einem Block, d.h.

in jedem Block 8 zerlegende Vektormultiplikation bearbeitet werden. Aus den Ergebnisse von Abb.6.(Vergleich von optimierte Vollmatrixmultiplikation mit C-Implementierung und alte GPU-Implementirung für MxN Vollmatrizen).



MxN	10000x50	100000x50	500000x50	1000x1000	5000x5000
CPU (ms)	1.56	15.94	80	3.44	87.19
oldGPU/CPU	1.711	1.6348	1.62548	0.164	0.12864
GPU/CPU	0.133	0.0662	0.06204	0.0512	0.03438

Fig. 6: Vergleich von optimierte Vollmatrixmultiplikation mit C-Implementierung und alte GPU-Implementirung für MxN Vollmatrix

Die optimierte GPU-Implementierung ist immer schnelle als die CPU-Implementierung und die Alte. Für Matrix 5000x5000 kann die CUDA-Programm 30 mal schneller als CPU

D. Schwierigkeit bei der Synchronisation vom Blocküebergreifen

Für größe Vektormultiplikation müssen mehrere Blöcke verwendet werden. Das Endeergebniss bekommt man durch die Blocksummation. Da wir keine Synchronisationsmaßnahme für Blocksummation haben, kann der Vektor nicht im selben Kernel bearbeitet werden. Die möglichkeiten wären, entweder die Blocksummation in CPU bearbeitet wird, oder die Blocksummation in einem neuen Kernel bearbeitet wird.

E. Texture Memory

Die Karteneinheit für „Texture Memory” kann nicht benutzt werden zur Beschleunigung weil diese nur float, aber nich double unterstützt.

F. Fehlersuche im laufenden Algorithmus

Der auf der Karte ablaufende Prozess ist zum Debuggen und automatisierten Testen zunächst schwer zugänglich weil die klassischen Instrumente des Debuggings, angefangen beim klassischen ’printf()’ darauf aufsetzen daß alle benötigten Daten im Speicher zugänglich sind.

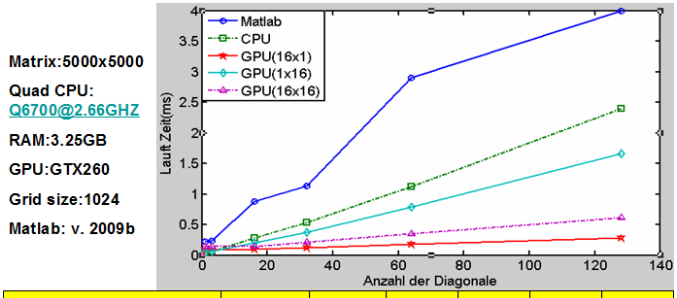
Diese Probleme werden in unserer Implementierung adressiert durch eine Zwischenschicht die im Debugging Mode die jeweils für das Debugging relevanten Daten aus der Karte extrahiert und in leicht verarbeitbarer Form ins Memory spiegelt:

Die Einzelnen Operationen werden je in ein Command-Muster [7] gewrappt und je einmal in CUDA und auf der Host-CPU implementiert. Die Host-Implemetierung wird dabei als Referenz und Sollwertgeber für das erwartete Ergebnis benutzt. Der Algorithmus wurde als Template-Muster ausgeführt so daß im Testbetrieb wahlweise CUDA-und CPU Operationen auf die Algorithmus-Instanz aufkonfiguriert werden können oder zwei Algorithmus-Instanzen im Parallelbetrieb gefahren werden können. Dadurch können wir im Testbetrieb alle Algorithmusschritte mit einem Soll-Ist-Vergleich durchfahren und so automatisiert jene Abweichungen zwischen CPU-und CUDA-Implementierung lokalisieren die manuell schwer oder garnicht aufgefunden werden können.

IV. Testergebnisse

A. sparsemul

Versuch der Vergleichung von matlab,CPU-und GPU-Implimentierung wird in Abb.7. ausgewiesen. Für 128-Diagonalmatix kann CUDA-Implementierung gegen CPU zu Faktor 9 erreichen.



Anzahl Diagnalen	1	3	16	32	64	128
CPU (ms)	0.03552	0.0588	0.2789	0.5348	1.1264	2.3986
Matlab (ms)	0.219	0.241	0.878	1.129	2.898	4
GPU(1x16) (ms)	0.0513	0.0618	0.2042	0.376	0.7869	1666
GPU(16x1) (ms)	0.1438	0.1445	0.1466	0.2115	0.3468	0.612
GPU(16x16) (ms)	0.09168	0.091	0.0932	0.1191	0.1739	0.2805

Fig. 7: Vergleichung der sparser Matrizemultiplikation von matlab, CPU-und GPU-Implementierung.

B. Skalarprodukt

TABLE II: Ausführungszeiten der Operation Skalarprodukt, gemessen auf GT 9500 in single precision

N	t_{CPU}/ms	t_{GPU}/ms	Speedup
1000	0.033824	0.006368	-
10000	0.044928	0.048160	1.1
100000	0.184160	0.495968	2.7
1000000	1.406144	5.045984	3.5
9000000	12.563456	45.246208	3.7

Tabelle II zeigt die Ausführungszeiten des Skalarproduktes in Abhängigkeit der Vektorgröße N, je für eine Host-CPU Implementierung im Vergleich zu einer GPU-Implementierung.

C. Löser Gauss

Tabelle III zeigt daß die GPU-Implementierung des Gauss-Lösers langsamer läuft als die CPU-Implementierung. Viele

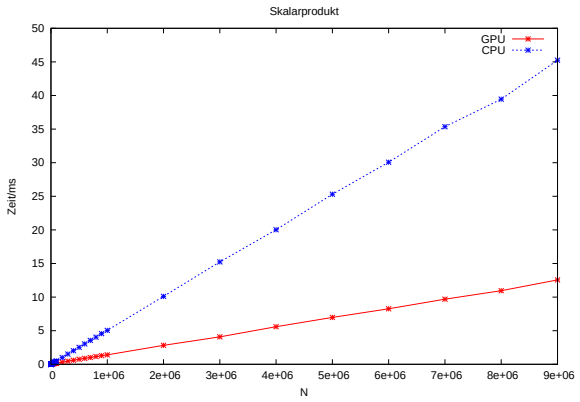


Fig. 8: Skalarprodukt: Rechenzeit in Abhängigkeit von der Problemgröße N

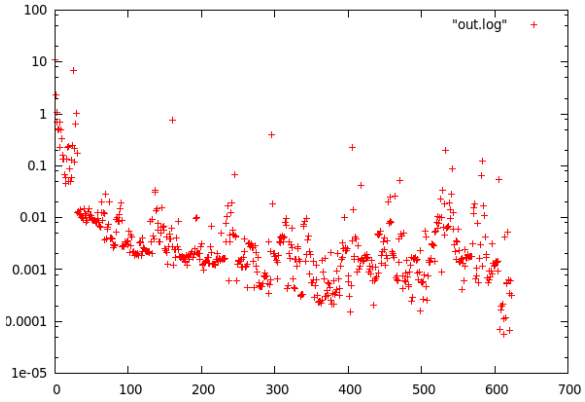


Fig. 9: Residuenverlauf des CUDA-IDR(4) bei einer Toleranz von 10^{-4}

Schritte des Gauss-Algorithmus können nur sequentiell ausgeführt werden. Im rein sequentiellen Betrieb (bspw. im substitute-Teil des Algorithmus) ist die langsameere Taktrate der CUDA-Karte der Größte Einfluss auf das Verhältnis der Ausführungszeiten. Für den IDR(s) relevant sind die $N \in [3, 6]$

TABLE III: Ausführungszeiten der Operation Löser Gauss

N	t_{CPU}/ms	t_{GPU}/ms	Speedup
3	0.037280	0.003168	-
4	0.030752	0.003104	-
5	0.033824	0.003104	-
6	0.043776	0.003040	-
10	0.073088	0.014432	-
19	0.176288	0.071456	-

D. IDR(s) Gesamtalgorithmus

Testproblem für das Messen ist das LGS für ein 1D-Laplaceproblem mit Randwerten. Die Zeilenzahl des Test-LGS wird N genannt.

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix} \cdot \vec{\mathbf{x}} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

Pro Iterationsschritt i wird das Residuum $r_i = |\mathbf{A} \cdot \tilde{\mathbf{x}}_\mathbf{i} - \tilde{\mathbf{b}}|$ aufgezeichnet und im Abb. 9 gegen i aufgetragen.

Der CUDA-IDR(s) konvergiert bis zu einer Genauigkeit von 10^{-4} bei $N = 700$

V. Möglichkeiten für die weitere Optimierung der IDR(s)-Implementierung

- Spezialkernel die angepasst sind auf bestimmte Matrixgrößen, denn der optimale Kernel für $A_{Ns} \cdot B_{s1}$ muß anders implementiert werden als $A_{sN} \cdot B_{N1}$. Hier wählt das Operation-Command selbstätigt den passenden Kernel in Abhängigkeit von s und N.
- Einfügen von Instrumentation-Code analog dem Tuning-Interface des Oracle-RDBMS [9]. Die Idee besteht darin automatisiert jene Operationen zu identifizieren die in Summe den größten Beitrag zur Gesamtlaufzeit beitragen. Das Codeskelett dieses Instrumentation-Codes wurde bereits erstellt, aber noch nicht im Gesamtsystem verbaut.
- Operationen-Commands per Regelsatz in Abhängigkeit von Problemgröße und Struktur die Größe von Block und Grid wählen

VI. Zusammenfassung

References

[1] Peter Sonneveld and Martin B. van Gijzen, *IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems.* SIAM J. Sci. Comput. Vol. 31, No. 2, pp. 1035-1062 (2008)
[2] NVIDIA Corporation. (2009) *NVIDIA CUDA Programming Guide Version 2.3* [Online] Available: http://www.nvidia.de/object/cuda_develop_emeai.html
[3] Peter Sonneveld and Martin B. van Gijzen, (December 2008) *idrs.m* [Online] Available: http://ta.twi.tudelft.nl/NW/users/gijzen/idrs.m
[4] The Math works *Matlab data* [Online] Available: http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f21585.html
[5] NVIDIA Corporation. (2009) *CUDA CUBLAS in CUDA Toolkit v2.3* [Online] Available: http://www.nvidia.de/object/cuda_develop_emeai.html
[6] NVIDIA Corporation. (2009) *NVIDIA CUDA C Programming Best Practices Guide* CUDA Toolkit 2.3 [Online] Available: http://www.nvidia.de/object/cuda_develop_emeai.html
[7] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995
[8] Robert Sedgewick *Algorithmen in C* Addison Wesley, 1992
[9] Mogens Norgaard *You probably dont't tune right* in *Oracle Insights: Tales of the Oak Table* New York, Apress, 2004, ch.2, pp 71-94
[10] Mark Harris *Optimizing Parallel Reduction in CUDA* in *CUDA SDK* Nvidia Corporation [Online] Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/r