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. Arithmetische Operationen

A. Operationen

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

TABLE I: Im IDR(s) verwendete Operationen

Operation	Zusammenhang	Bemerkung
Addition	$\mathbf{c} = \mathbf{a} + \mathbf{b}$	
Skalarprodukt	$s = \mathbf{a} \cdot \mathbf{b}$	
Norm	$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$	
Matrix-Vektorprodukt	$\mathbf{c} = \mathbf{A_{full}} \cdot \mathbf{b}$	$\mathbf{A_{full}}$ vollbesetzt
Matrix-Vektorprodukt	$\mathbf{c} = \mathbf{A_s} \cdot \mathbf{b}$	$\mathbf{A_s}$ dünnbesetzt
Löser nach Gauß	$\mathbf{M} \cdot \mathbf{m} = \mathbf{c}$	Zeile 36 idrs.m [1]

$\mathbf{A_s}$ wird im Speicher dargestellt durch das „Sparse Matrix”-Speicherformat aus Matlab. [3]

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) Skalarprodukt: Abb. 1 zeigt schematisch eine Skalarprodukt-Implementierung in CUDA:

Der $n$te Thread berechnet das Vektorelement $c_n = a_n \cdot b_n$ Die Elemente des Ergebnisvektors $\mathbf{c}$ werden zu einem Skalar aufsummiert. Überschreitet die Vektorgröße die maximale Blockgröße 512 iteriert der Thread über mehrere Elemente, oder mehrere Blöcke müssen abgestimmt auf den großen Vektoren arbeiten.

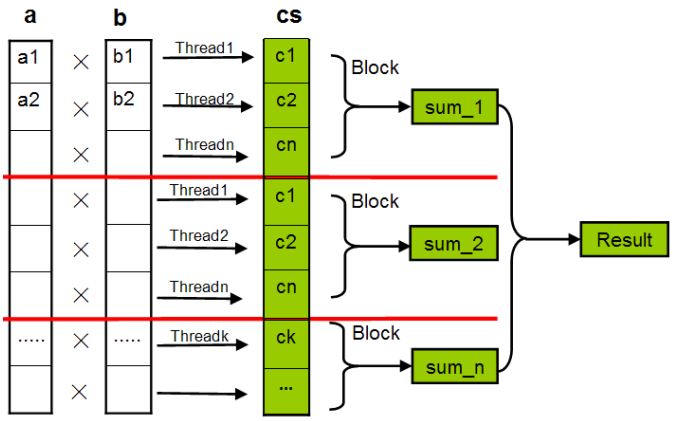


Fig. 1: Skalarprodukt: Multiplikand $\mathbf{a}$, Multiplikator $\mathbf{b}$, (Zwischen-)Ergebnisvektor $\mathbf{c_s}$

2) Matrix-Vektorprodukt: Die CUDA-Implementierung zerlegt die Matrix-Vektormultiplikation in einzelne Verktorprodukte. Abb.2 zeigt wie die Teilvektoren von $\mathbf{A}$ mit dem Vektor $\mathbf{b}$ innerhalb eines Blockes multipliziert werden.

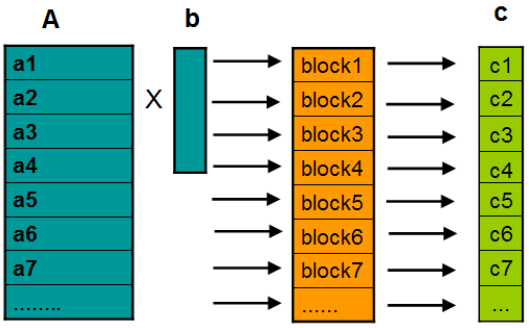
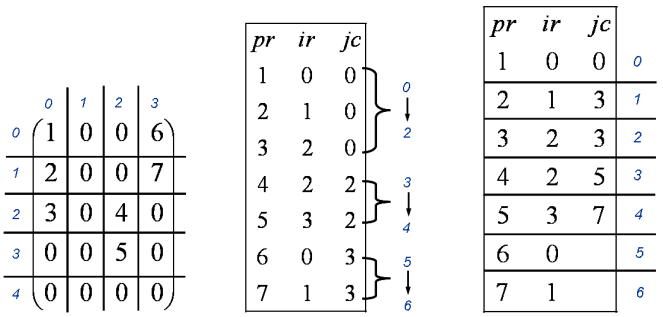


Fig. 2: Matrix-Vektorprodukt: Multiplikand $\mathbf{A}$, Multiplikator $\mathbf{b}$, Produkt $\mathbf{c}$

3) Matrix-Vektorprodukt mit Sparsematrix: Als Sparsematrix oder dünnbesetzte Matrix bezeichnet man eine Matrix bei der die Mehrheit der Elemente aus Nullen besteht. Abb.3

zeigt von links nach rechts die Reduktion von einer vollbesetzten Struktur zu einer platzsparenden Struktur. Aus Platzspargründen nutzt Matlab ein spezielles Memorylayout [3] mit dem nur die nonzero-Elemente im Speicher gehalten werden.



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

Fig. 3: Sparse Matrix

Unsere CUDA-Implementierung verwendet zur Speicherung ein abgewandeltes Matlab-Format. Statt in Spaltenfolge speichern wir die Matrix in Zeilenfolge, um bei der Multiplikation mit einem Spaltenvektor direkt die Multiplikationsschleife „über die Zeile treiben" zu können.

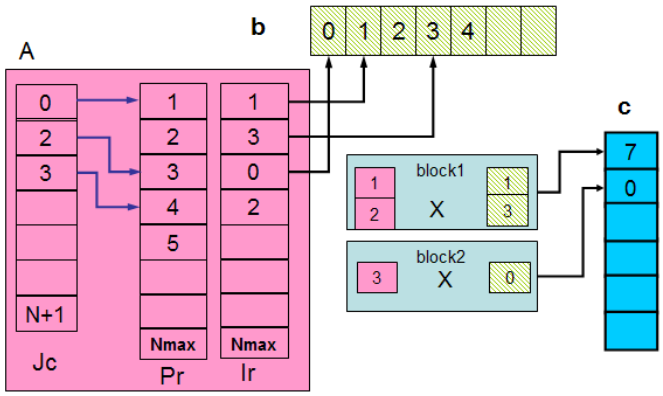


Fig. 4: Matrix-Vektorprodukt mit Sparsematrix: Sparsematrix $\mathbf{A}$, Zeilenindex $\mathbf{j_c}$, Nonzero-Elemente $\mathbf{P_r}$, Spalteninformation $\mathbf{I_r}$, Multiplikator $\mathbf{b}$, Produkt $\mathbf{c}$

4) Löser Gauß : Basis für unsere CUDA-Implementierung ist ein Algorithmus aus der Standardliteratur [4] mit Pivotisierung. Da Teile des Algorithmus (z.B. Rückwärtssubstitution, Maximumssuche 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. Summation längs eines Vektors

Als typisches Teilproblem tritt die Summation der Vektorelemente zu einem Skalar auf, bspw. beim Skalarprodukt und bei der Matrizenmultiplikation. Der klassische Ansatz, die Addition als Schleife mit einem Akkumulator auszuführen, wäre ineffizient weil nur ein einziger Thread für die Arbeit benutzt werden kann. Abb. 5 zeigt schematisch ein Verfahren mit mehreren Threads welche Teilergebnisse aufsummieren. Die Grundidee: im ersten Schritt werden die Elemente cs_{2n} und cs_{2n+1} summiert. Im zweiten Schritt cs_{4n} und cs_{4n+1}. Für die Errechnug des Ergebnisses benötigt man $\log_2(blocksize)$ Iterationsschritte.

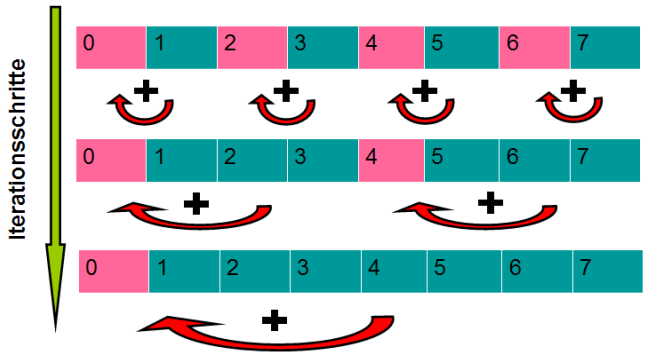


Fig. 5: Dreieckförmige Summation, Iterationsschritte von oben nach unten.

Beispielcode und eine detaillierte Beschreibung findet man in [5].

B. Minimierung leerlaufender Threads

In CUDA bearbeitet jeder Multiprozessor gleichzeitig mit 32 Threads [2]. Diese gehören alle zum selben Block. Bei der Sparsematrix-Multiplikation sind viele Threads im Leerlauf. Um die Anzahl der leerlaufenden Threads zu minimieren, müssen mehrere Punktprodukte in einem Block bearbeitet werden. Dazu verwendet man zweidimensionale Blocksizes. Die Definition und Anwendung zweidimensionaler Blocksizes findet man in der Referenz[2] und [6].

C. Shared Memory

In der Grafikkarte ist der Zugriff auf den globalen Speicher langsamer als auf den On-Chip-Speicher. Nach dem Beispiel in [2] kann man mehrmals verwendete Daten zunächst in das shared Memory kopieren und dann für die entsprechenden Operationen benutzen. In der Multiplikation Vollmatrix mal Vektor wird jedes Vektorelement mehrmals gebraucht. Nach Untersuchungen wählen wir einen eindimensionalen Block der Größe 64 Jedes Vektorelement wird 8 mal bnutzt in einem Block, d.h. in jeder Block bearbeitet 8 zerlegte Vektormultiplikationen. Die TabelleII zeigt die Laufzeitenunterschiede der unterschiedlicher Implementierungen.

Die optimierte GPU-Implementierung ist immer schnelle als die CPU-Implementierung und die nichtoptimierte Version. Für eine Matrixgröße 5000x5000 kann läuft das CUDA-Programm 30 mal schneller als die CPU-Implementierung.

TABLE II: Ausführungszeiten der Vollmatrix-Vektormultipkation $\mathbf{A_{mn}} \cdot \mathbf{b}$

m	10^4	10^5	50000	10^3	5000
n	50	50	50	10^3	5000
CPU(ms)	1.56	15.94	80	3.44	87.9
old GPU/CPU	1.711	1.6348	1.62548	0.164	0.12854
GPU/CPU	0.133	0.0662	0.06204	0.0512	0.03438

D. Threadsynchronisation über Blockgrenzen hinaus

Für größe Vektormultiplikation benötigt man mehrere Blöcke. Da die Threads nur innerhalb eines Blockes synchronisiert werden können muß man die Blockergebnisse entweder in der CPU oder in einem zweiten Kernel weiter aufsummieren.

E. Fehlersuche im laufenden Algorithmus

Für die automatisierte Fehlersuche im laufenden CUDA-Algorithmus per Soll-Ist-Vergleich sind die Operationen als Command-Muster [7], der IDR(s) als Template-Muster [7] ausgeführt. CPU- oder GPU-Implementierungen der Operationen können auf den Algorithmus aufkonfiguriert werden.

IV. Testergebnisse

Alle Test sind ausgeführt in double precision. Die vewendete GPU ist, wenn nicht anders genannt, eine Tesla C1060. Matlab wird in der Version v.2009b verwendet.

A. Sparsematrix-Vektorprodukt

Der Vergleich von Matlab, CPU- und GPU-Implementierung wird in Abb.6 und Tabelle III gezeigt. Für eine 128-Diagonalmatix kann die CUDA-Implementierung (16x16) einen Speedup-Faktor von 9 gegenüber der CPU erreichen.

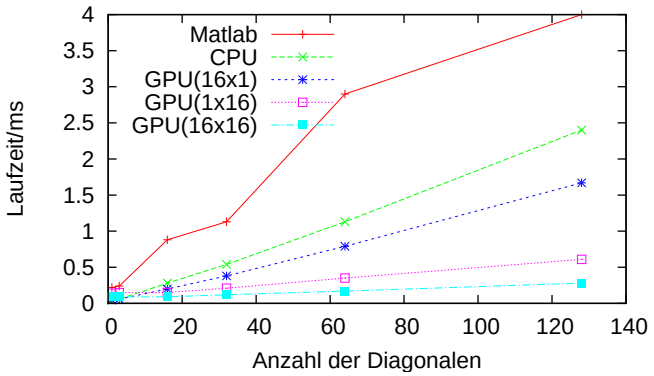


Fig. 6: Sparsematrix-Vektormultiplikation. Matrix 5000x5000, Quad CPU 2.66GHz, RAM 3.25GB, GPU GTX260, Grid size 1024

TABLE III: Ausführungszeiten der Sparsematrix-Vektormultiplikation in ms

Diagonalen	1	3	16	32	64	128
Matlab	0.219	0.241	0.878	1.129	2.898	4
CPU	0.036	0.059	0.279	0.535	1.126	2.399
GPU(1x16)	0.051	0.062	0.204	0.376	0.787	1.666
GPU(16x1)	0.143	0.145	0.147	0.212	0.347	0.612
GPU(16x16)	0.092	0.091	0.0932	0.119	0.174	0.281

TABLE IV: Ausführungszeiten der Operation Skalarprodukt

N	$\mathbf{t_{CPU}/ms}$	$\mathbf{t_{GPU}/ms}$	**Speedup**
1000	0.041344	0.006848	-
8000	0.044288	0.043040	1.0
10^5	0.086080	0.551808	6.8
10^6	0.332928	6.685600	20.2
10^7	2.675264	67.924385	26.1

B. Skalarprodukt

Tabelle IV zeigt die Ausführungszeiten des Skalarproduktes in Abhängigkeit der Vektorgröße N, je für eine Host-CPU Implementierung im Vergleich zu einer GPU-Implementierung. In Abb. 7 ist an der Stelle $N = 8000$ der Schnittpunkt der Ausführungszeitgraphen zu sehen - für große N führt die GPU die Operation schneller aus.

C. Löser Gauß

Tabelle V zeigt daß die GPU-Implementierung des Gauß-Lösers langsamer läuft als die CPU-Implementierung. Viele Schritte des Gauß-Algorithmus können nur sequentiell ausgeführt werden. Im rein sequentiellen Betrieb (bspw. im substitute-Teil des Algorithmus) ist die langsamere Taktrate der CUDA-Karte der größte Einfluss auf das Verhältnis der Ausführungszeiten. Für den IDR(s) relevant sind die $N \in [3, 6]$

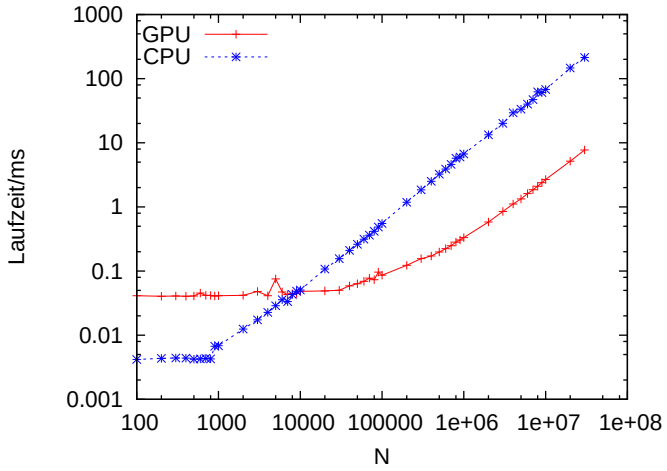


Fig. 7: Skalarprodukt: Ausführungszeit in Abhängigkeit von der Elementanzahl N

TABLE V: Ausführungszeiten der Operation Löser Gauß

N	t_{GPU}/ms	t_{CPU}/ms	Speedup
3	0.037280	0.003168	-
4	0.030752	0.003104	-
5	0.033824	0.003104	-
6	0.043776	0.003040	-

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.

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

1) Konvergenzverhalten: Der Ergebnisvektor $\mathbf{x}$ wird mit Zufallswerten $x_n \in [-0.5, 0.5]$ vorbelegt. Pro Iterationsschritt i wird das Residuum $r_i = |\mathbf{A} \cdot \mathbf{x_i} - \mathbf{b}|$ aufgezeichnet und im Abb. 8 gegen i aufgetragen.

Zu diesem Testproblem ergibt sich bei hinreichend genauer Toleranz ϵ als Lösung für x eine Gerade. (Abb. 9)

Der CUDA-IDR(s) führt zu einer Gerade als Lösung bei einer Genauigkeit von 10^{-8} für N bis zu $N = 20000$.

Für größere N wird keine Gerade mehr errechnet. Für Toleranzen kleiner als 10^{-8} konvergiert der CUDA-IDR(s) nicht mehr, nach Durchschreiten des lokalen Residuen-Minumums klingt das Residuum auf gegen Unendlich.

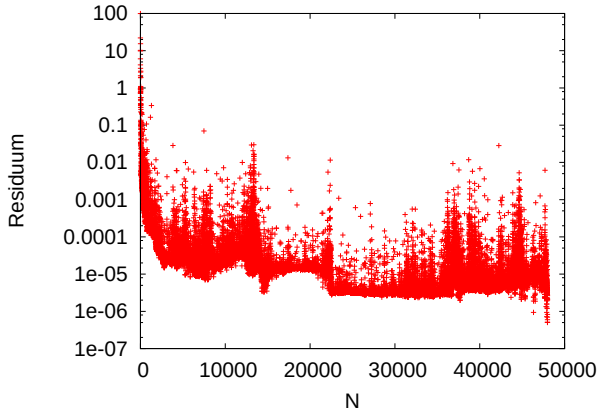


Fig. 8: Residuenverlauf, CUDA-IDR(4), $N = 20000$, $\epsilon = 10^{-8}$

2) Zeitverhalten: Zum Zeitpunkt der Niederschrift läuft der CUDA-IDR(s) langsamer als die Matlab-Implementierung bei gleicher Parametrisierung (Tabelle VI). Das Zusammenspiel der Operationen ist mometan abgestimmt auf „reines Funktionieren" bzgl. des Konvergenzverlaufes, eine Zeitoptimierung des Gesamtalgorthmus fehlt.

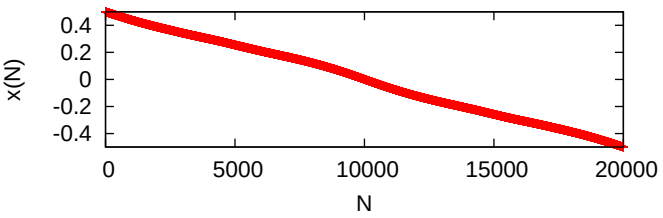


Fig. 9: Lösung $\mathbf{x}$, CUDA-IDR(4), $N = 20000$, $\epsilon = 10^{-8}$

TABLE VI: Ausführungszeiten des CUDA-IDR(4)

N	t_{Matlab}/s	t_{GPU}/s
10000	64	137
20000	415	922

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 [8]. 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.

VI. Zusammenfassung

Der CUDA-IDR(s) liefert dieselben Ergebnisse wie die Matlab-Vorlage. Die Einzeloperationen laufen schneller als ihre CPU/Matlab-Entsprechungen, der CUDA-IDR(s) insgesamt ich noch nicht optimiert bzgl. des Zeitverhaltens.

References

[1] P. Sonneveld and M. 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. [Online]. Available: http://ta.twi.tudelft.nl/nw/users/gijzen/idrs_siam.pdf

[2] (2009) NVIDIA CUDA Programming Guide Version 2.3. Nvidia Corporation. [Online]. Available: http://www.nvidia.de/object/cuda_develop_emeai.html

[3] (2009) Matlab data. The Math works. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f21585.html

[4] R. Sedgewick, *Algorithmen in C*. Addison-Wesley, 1992.

[5] M. Harris. (2009) Optimizing Parallel Reduction in CUDA. Nvidia Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[6] (2009) NVIDIA CUDA C Programming Best Practices Guide. Nvidia Corporation. [Online]. Available: http://www.nvidia.de/object/cuda_develop_emeai.html

[7] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] M. Norgaard, "You probably dont't tune right," in *Oracle Insights: Tales of the Oak Table*. New York: Apress, 2004, ch. 2, pp. 71–94.