

# 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

**D**AS Lösen großer linearer Gleichungssysteme 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 [3] der linearen Algebra, im einzelnen in Tabelle I beschrieben.

TABLE I: Im IDR(s) verwendete Operationen

Operation	Zusammenhang	Bemerkung
Addition	$c = a + b$	
Skalarprodukt	$s = a \cdot b$	
Norm	$ a  = \sqrt{a \cdot a}$	
Matrix-Vektorprodukt	$c = A_{full} \cdot b$	$A_{full}$ vollbesetzt
Matrix-Vektorprodukt	$c = A_s \cdot b$	$A_s$ dünnbesetzt
Löser nach Gauß	$M \cdot m = c$	LGS aus Zeile 36 [3]

$A_s$  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 Beschränkung des CUDA-Blocksizes maximum 512 Threads enthält, muss ein Block iterativ oder mehrere Blocks verwendet werden.

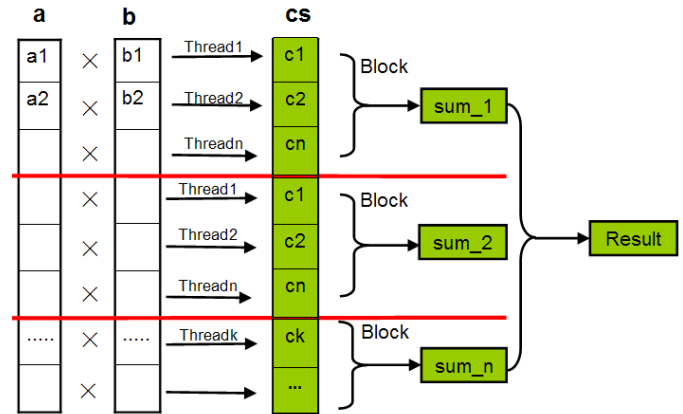


Fig. 1: Vektor Multiplikation.  $A$  : Erst Vektor;  $b$ : Zweiter Vektor;  $cs$ : Produktvektor

2) *Multiplikation von Matrix mit Vektor in CUDA Implementierung*: Beim Wissenschaftlichen Rechnen trifft man häufig die Multiplikation von Matrizen mit Vektor. In CUDA Implementierung wird die Operation als unterschiedliche Vektor-Multiplikationen zerlegt. Mit ähnlicher Methode werden auch Matrizen 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 Index, deren die zur Spalte  $i$  gehörten Nonzero-Elemente und Zeileninformation aufweisen.

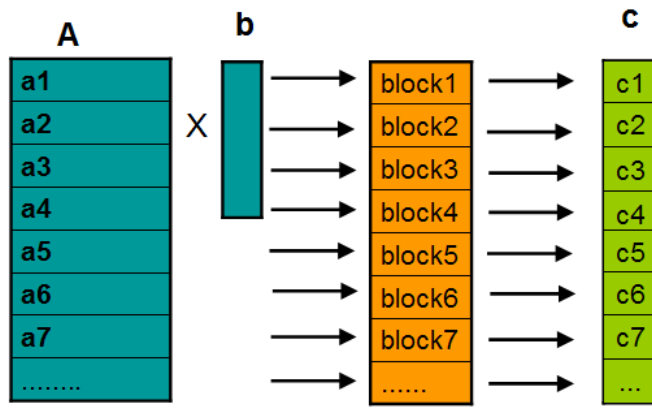


Fig. 2: Matrix mal Vektor. A: Matrix; b: Vektor; c: Produktvektor

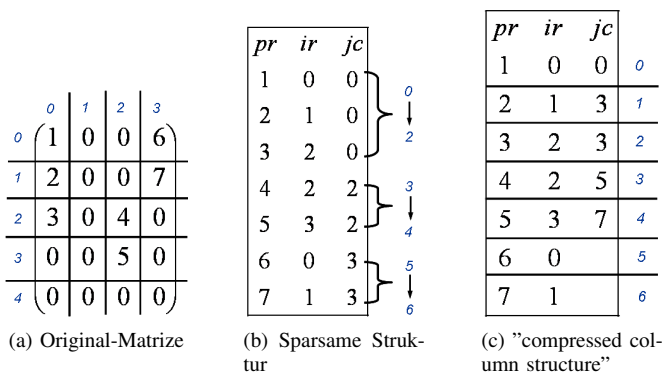


Fig. 3: Sparse Matrix

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

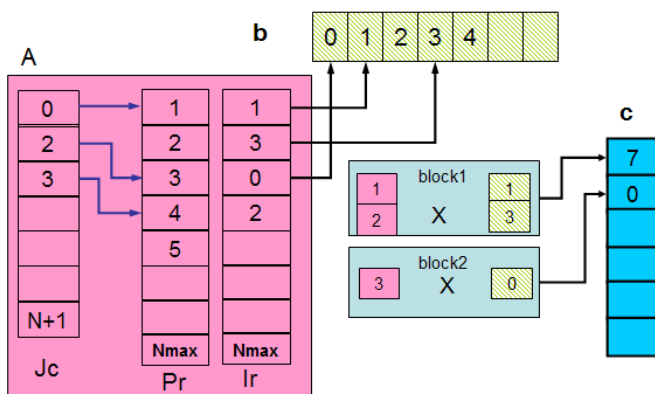


Fig. 4: Sparse Matrizenmultiplikation. A: Sparsematrix, B: Vektor der Zeileinfo, C: Vektor der Nonzeroelements, D: Vektor der Spalteninfo ; E: Vektor; F: produkt Vektor

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

leität.

### 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 Bloksumation. Aus der Beschreibungen der Operationen Multiplikationen der Matrix mal Vektor und Sparsematrix mal Vektor beruhen obige Operationen auf Vektormultiplikationen, die schließlich ein Summierungsverfahren in jedem Block enthalten. Blocksumation 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 erhält man endlich Ergebnisse.

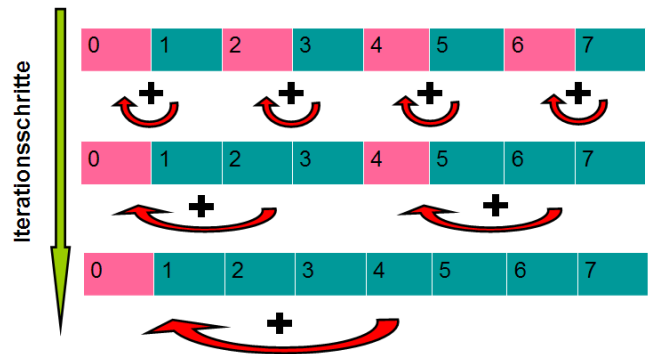


Fig. 5: Dreieckförmige Summation. Von Oben nach Unter zeigt

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 Sparsematrix-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 Grafik 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-Implementierung für  $M \times N$  Vollmatrizen).

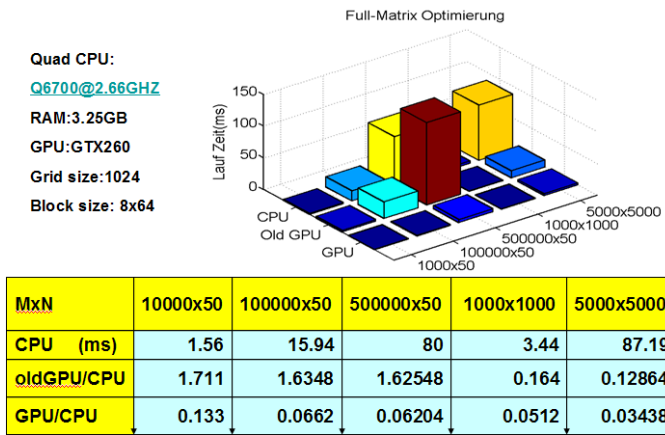


Fig. 6: Vergleich von optimierte Vollmatrixmultiplikation mit C-Implementierung und alte GPU-Implementierung für  $M \times N$  Vollmatrix

Die optimierte GPU-Implementierung ist immer schneller 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 Blockübergreifen

Für große Vektormultiplikation müssen mehrere Blöcke verwendet werden. Das Endergebnis 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. 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

#### A. sparsemul

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

#### B. Skalarprodukt

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

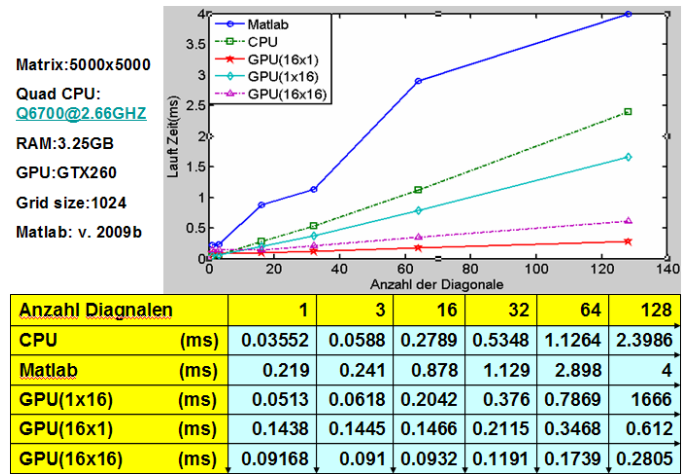


Fig. 7: Vergleichung der sparser Matrizenmultiplikation von matlab, CPU-und GPU-Implementierung.

TABLE II: Ausführungszeiten der Operation Skalarprodukt, gemessen auf Tesla C1060 in double precision

N	t <sub>CPU</sub> /ms	t <sub>GPU</sub> /ms	Speedup
1000	0.041344	0.006848	-
8000	0.044288	0.043040	1.0
10 <sup>5</sup>	0.086080	0.551808	6.8
10 <sup>6</sup>	0.332928	6.685600	20.2
10 <sup>7</sup>	2.675264	67.924385	26.1

In Abb 8 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 III 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 Taktzahl

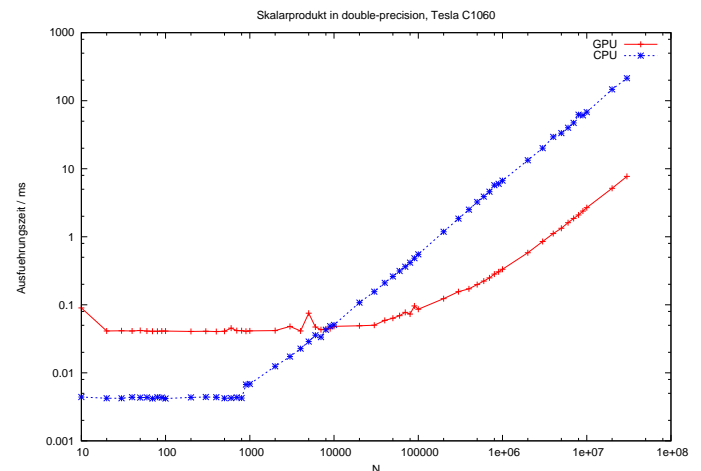


Fig. 8: Skalarprodukt: Ausführungszeit in Abhängigkeit von der Problemgröße  $N$ , double precision, Tesla C1060

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 <sub>GPU</sub> /ms	t <sub>CPU</sub> /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.

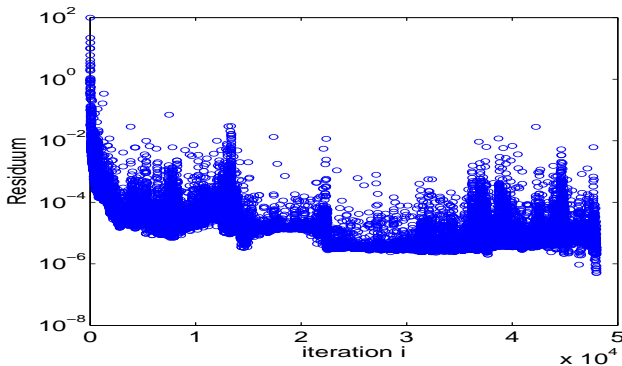
$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

1) *Konvergenzverhalten:* Der Ergebnisvektor  $\mathbf{x}$  wird mit Zufallswerten  $x_n \in [-0.5, 0.5]$  vorgelegt. Pro Iterationsschritt  $i$  wird das Residuum  $r_i = |\mathbf{A} \cdot \mathbf{x}_i - \mathbf{b}|$  aufgezeichnet und in Abb. 9 gegen  $i$  aufgetragen.

Zu diesem Testproblem ergibt sich bei hinreichend genauer Toleranz  $\epsilon$  als Lösung für  $\mathbf{x}$  eine Gerade. (Abb. 10)

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-Minimums klingt das Residuum auf gegen Unendlich.

Fig. 9: Residuenverlauf des CUDA-IDR(4) bei  $N = 20000$  und Toleranz  $\epsilon = 10^{-8}$ 

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

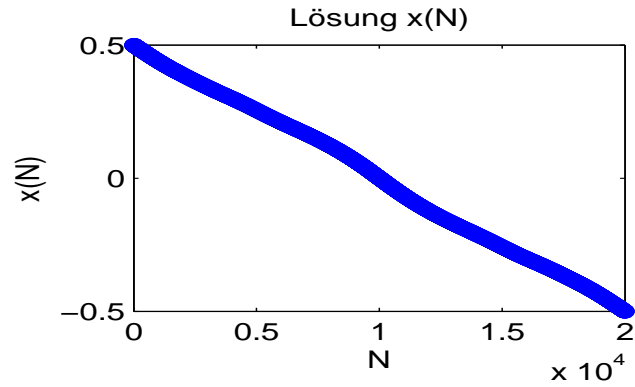
Fig. 10: Lösung  $\mathbf{x}$  des CUDA-IDR(4) bei  $N = 20000$  und Toleranz  $\epsilon = 10^{-8}$ 

TABLE IV: Ausführungszeiten des IDR(4), gemessen je auf Matlab v.2009b und auf Tesla C1060 in double precision

N	t <sub>Matlab</sub> /s	t <sub>GPU</sub> /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 selbstständig 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.

#### 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] 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](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](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](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](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 don'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/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf)