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	$\vec{\mathbf{c}} = \vec{\mathbf{a}} + \vec{\mathbf{b}}$	
dotmul	$s = \vec{\mathbf{a}} \cdot \vec{\mathbf{b}}$	
norm	$\lvert\vec{\mathbf{a}}\rvert = \sqrt{\vec{\mathbf{a}} \cdot \vec{\mathbf{a}}}$	
matrixmul	$\vec{\mathbf{c}} = \mathbf{A_{full}} \cdot \vec{\mathbf{b}}$	$\mathbf{A_{full}}$ vollbesetzt
sparsemul	$\vec{\mathbf{c}} = \mathbf{A_s} \cdot \vec{\mathbf{b}}$	$\mathbf{A_s}$ dünnbesetzt
solvergauss	$\mathbf{M} \cdot \vec{\mathbf{m}} = \vec{\mathbf{c}}$	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) dotmul: Im Bild 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.

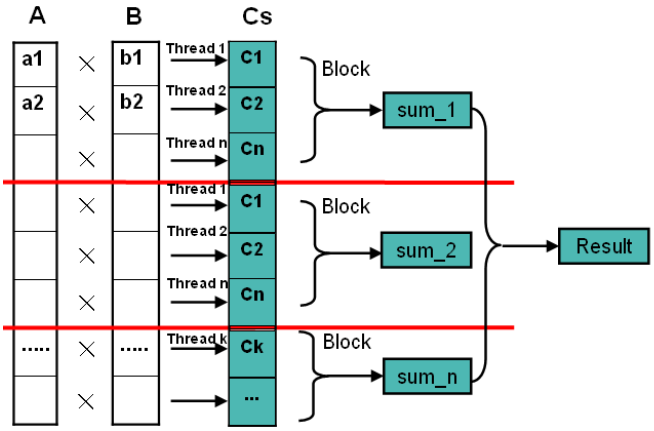


Fig. 1. Vektor Multiplikation. A : Erst Vektor; B: Zweiter Vektor; Cs: Produktvektor

2) Multiplikation von Matrize 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 Fig.2.zeigt, dass jede zerlegte Vektor von A mit Vektor B in einem Block multipliziert wird.

3) sparsemul: Sparse Matrize

Sparsematrizen, oder dünnbesetzte Matrizen, bezeichnet man als eine Matrizen, bei der so viele Einträge aus Nullen bestehen. Im Fig 1 wird ein einfaches Beispiel gezeigt. Da Sparsematrizen mit Vollmatrizen genau umgekehrt ist, hat man dafür auch eine andere Speicherweise. Unter dem Zusammenhang zwischen Fig.3., Fig.4. und Fig.5. versteht man, dass bei der Sparsematrizen wird nun nur die Nonzero-Elementeund 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 Fig.5. 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.

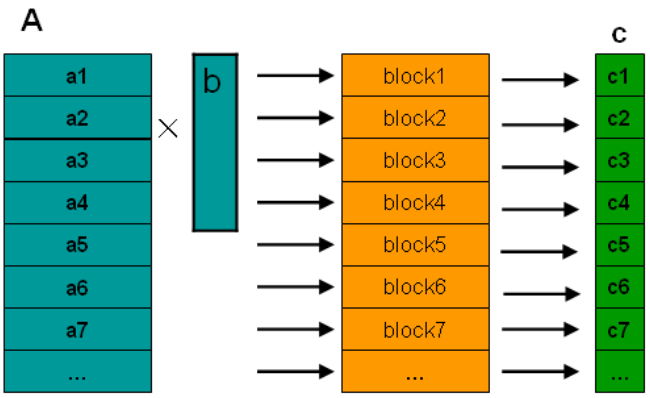


Fig. 2. Matrize mal Vektor. A: Matrize; b: Vektor; c: Produktvektor

	0	1	2	3
0	1	0	0	6
1	2	0	0	7
2	3	0	4	0
3	0	0	5	0
4	0	0	0	0

Fig. 3. Original-Matrize

pr	ir	jc	
1	0	0	0
2	1	0	
3	2	0	2
4	2	2	3
5	3	2	4
6	0	3	5
7	1	3	6

Fig. 4. Sparsame Struktur

pr	ir	jc	
1	0	0	0
2	1	3	1
3	2	3	2
4	2	5	3
5	3	7	4
6	0		5
7	1		6

Fig. 5. ”compressed column structure”

Mit obengenanntem Methode werden Spasematrizen in Spaltfolg gespeichert. Bei unser Implementierung verwenden wir es als Zeilfolg. Im folgendem Fig.6. zeigt, wie die Multiplikation ausgeführt wird

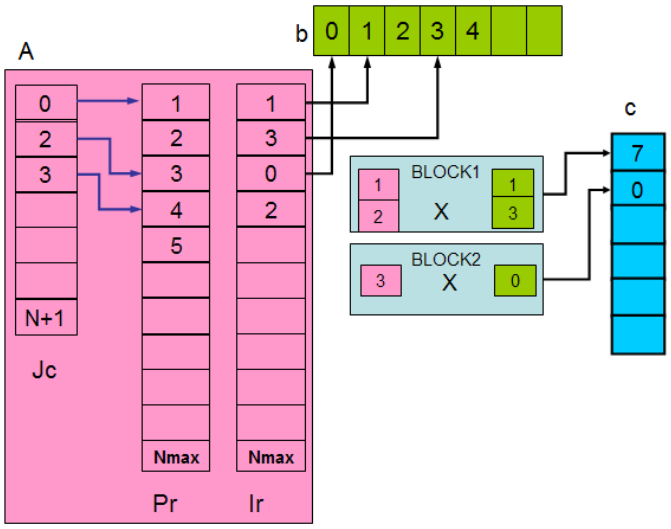


Fig. 6. Sparse Matrizenmultiplikation. A:Sparsematrize, Jc:Vektor der Zeileinfo, Pr: Vecktor der Nonzeroelements, Ir: Vektor der Spalteninfo ; b: Vektor; c: produkt Vektor

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 Matrize 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.7 . 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.

Beispiel Code sieht folgende aus,

```
#define BLOCK_EXP 9
#define DEF_BLOCKSIZE 1 << BLOCK_EXP
short offset = 1;
for (short i = 1;i < BLOCK_EXP; i++) {
    short old = offset;
    offset <<= 1;
    if (threadIdx.x % offset == 0) {
```

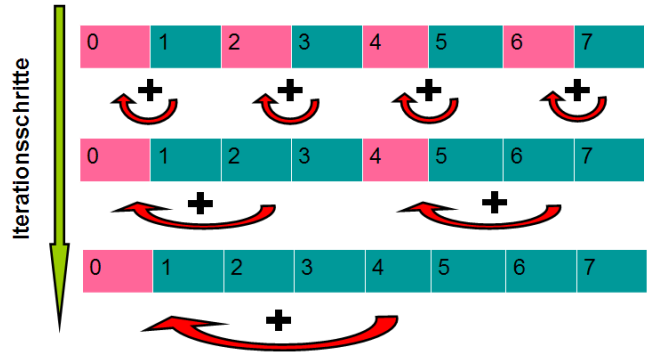


Fig. 7. Dreieckförmige Summation. Von Oben nach Unter zeigt

```
        Vs[threadIdx.x] += Vs[threadIdx.x
         + old];
    }
    __syncthreads();
}
if (threadIdx.x == 0) {
    out[0] = Vs[0] + Vs[offset];
}
```

B. Minimierung leer laufende Thread

Im Cuda, bearbeitet jeder Multiprozessor gleichzeitig ein Warp von 32 Threads [2], die alle zu denselben Blöcken gehören. d.h. Für sehr dumm besetzt Matrix, wie in sparse Matrixmultiplikation, sind viele Threads in Leerlauf. Um die Anzahl der in Leer laufende Thread zu minimieren, werden mehre Zeile in einem Block bearbeitet. Dazu verwendet man 2 dimensionierte Blocksizes. Die Dimension X wird hier für Elementemultiplikationen innerhalb jeweiliger Vektormultiplikationen definiert. Die Dimension Y besorgen Unterschiedlich Vektormultiplikationen innerhalb eines Blockes. Ein optimiertes Beispiel ist sparser Matrizemultiplikation, deren Implementierung ähnlich obiger Darstellung ist. Wie bestimmt man die genaue Größe für beide Dimensionen? Man kann für speziale Anwendungen durch Folgende Versuche die beide dimensionierte Größe auswahlen. Fig[] zeigt die Laufzeit der Multiplikation Ein-Diagonalematrize mal Vektor mit unterschiedliche Blocksize. Es ist offenbar, dass für 1-Diagonalmatize das optimale Blocksize 16x1 oder 32x1 beträgt. 1-Diagonalmatize ist nicht einzige dümbesetzte Matrize in unserer Anwendung. Dickere Sparsermatrizen entstehen auch häufig. Fig[] ist die Messung für 32-Diagonalmatize. Man findet 16x16 eine schlauere Auswahl.

C. Share Memory

Im Grafikkater ist Zugriff der globale Speicher langsamer als ändere Speicher. Wie Beispiele in [2] gezeigt, kann man mehrer mal verwendete Daten zunächst in share 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.

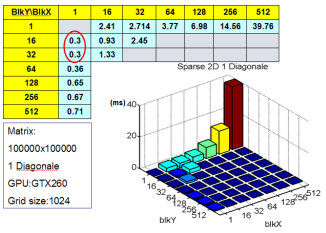


Fig. 8. Multipliktion Ein-Diagonalematrize mal Vektor. BlockY: Anzahl der Y-Dimension von Block; BlockX: Anzahl der X-Dimension von Block

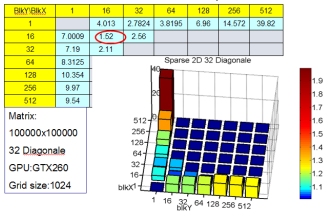
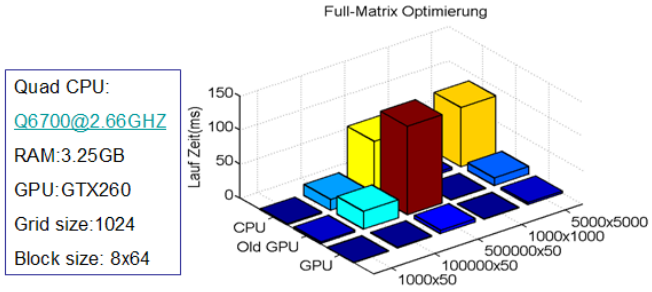


Fig. 9. Multipliktion 32-Diagonalematrize mal Vektor. BlockY: Anzahl der Y-Dimension von Block; BlockX: Anzahl der X-Dimension von Block.

Aus den Ergebnisse von Fig.10.(Vergleich von optimierte Vollmatrixmultiplikation mit C-Implementierung und alte GPU-Implementirung für MxN Vollmatrizen).



M x N	1000 x50	100000 x50	500000 x50	1000 x1000	5000 x5000
CPU	1.56	15.94	80	3.44	87.19
Old GPU	2.669	26.059	130.038	0.564	11.216
GPU	0.207	1.056	4.963	0.176	2.998

Fig. 10. Vergleich von optimierte Vollmatrixmultiplikation mit C-Implementierung und alte GPU-Implementirung für MxN Vollmatrizen

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

D. Blockübergreifende Synchronisation schwierig

Ein Weiter Problem befindet sich auf der Blocksyncronisation. Für größe Vekotrmultiplkation verwendet man wissentlich mehre Blöcke. Schließlich bekomt man ein Vektor von jede Blocksummer. Da Ohne valide Synchronisationsmaßnahme um alle Blocksummer zu synchronisieren, kann man den Vektor nicht direkt im selbes Kernel bearbeiten. Die mögliche Lösungen stellt da, 1st .die gesammte Summier wird in Cpu ausgerechnet.. 2.man leitet den Vektor von jede

Blocksummer zu einem neu Kernel, das sich Summierung beschäftigt.

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.

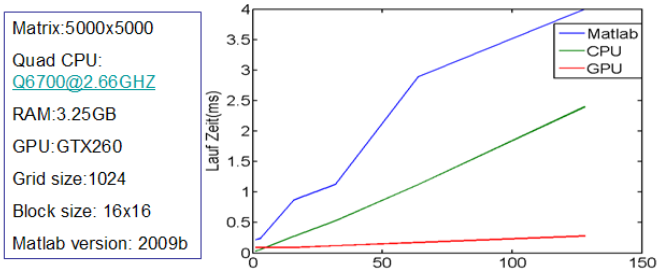
Diese Probleme werden in unserer Implementierung adressiert durch eine Zwischenschicht die im Debugging Mode die jeweils für das Debugging relevanten Daten aus der Karte extrahiert und in leicht verarbeitbarer Form ins Memory spiegelt:

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 Fig.11. ausgewiesen. Für 128-Diagonalmatize kann CUDA-Implementierung gegen CPU zu Faktor 9 erreichen.



Diagonale	1	3	16	32	64	128
matlab	0.219	0.241	0.878	1.129	2.898	4
CPU	0.0355	0.059	0.2789	0.5348	1.126	2.399
GPU	0.0917	0.091	0.0932	0.1191	0.1739	0.2805

Fig. 11. Vergleichung der sparser Matrizemultiplikation von matlab, CPU- und GPU-Implementierung.

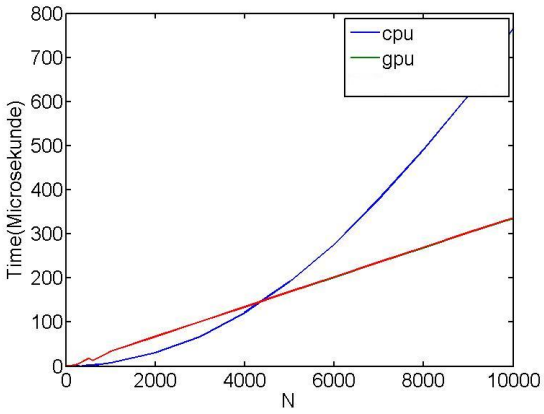


Fig. 12. Skalarprodukt: Rechenzeit in Abhängigkeit von der Problemgröße

B. norm und dotmul

Die Operationen dotmul und Norm laufen für eine Problemgröße $N > 5000$ schneller als die CPU-Implementierung. (Bild 12)

Trotz der Standardmaßnahmen [10] zur Summierung auf ein Skalar wurde die Ausführungsszeit nicht schneller als die der CUBLAS-Operationen [5].

C. Löser Gauss

(tbd Achim) Bislang langsamer als CPU-Implementierung

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 $r_i = |\mathbf{A} \cdot \tilde{\mathbf{x}_i} - \tilde{\mathbf{b}}|$ aufgezeichnet und im Diagramm 13 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

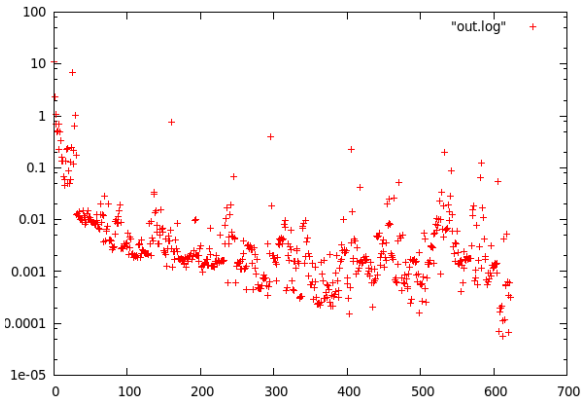


Fig. 13. Residuenverlauf des CUDA-IDR(4) bei einer Toleranz von 10^{-4}

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/reduction.pdf