Institut für Informatik

Softwareprojekt: Anwendungen von Algorithmen

Sommersemester 2010

Prof. Helmut Alt
Ludmilla Scharf, Sven Scholz

Lineare Algebra

Sebastian Kürten
sebastian.kuerten@fu-berlin.de

Zusammenfassung

Es wurden effiziente parallele Algorithmen für die Matrizenmultiplikation sowie für die Bestimmung der Determinante und das Lösen linearer Gleichungssysteme in CUDA implementiert. Die Algorithmen basieren zum Teil auf fremden Veröffentlichungen, wurden aber auch selbst weiterentwickelt. Die Auswertungen zeigen, dass die vorliegenden Implementierungen die parallele Architektur von CUDA effizient ausnutzen und zu einer Beschleunigung der Berechnungen gegenüber einfachen CPU-Implementierung führen können. Es wird aber auch aufgezeigt, wo die Implementierungen noch verbessert werden können.

Inhaltsverzeichnis

1 Einleitung **4**

2 Matrizenmultiplikation **4**
2.1 Algorithmus . . . 4
2.2 Implementierung . . . 5
2.3 Verfeinerung . . . 6
2.4 Evalutation . . . 6
2.5 Bewertung . . . 8

3 Gleichungssysteme & Determinante **8**
3.1 Gauß-Elimination . . . 9
3.1.1 Algorithmus . . . 9
3.1.2 Probleme . . . 9
3.1.3 Implementierung . . . 10
3.2 Jordan-Elimination und Bestimmung des Lösungraums . . . 11
3.3 Evaluation . . . 11
3.4 Determinante . . . 12
3.5 Bewertung . . . 12

4 Projekt- und Testumgebung **13**
4.1 Verzeichnisbaum . . . 13
4.2 Build-Prozess . . . 14
4.3 Tests . . . 14

5 Erfahrungen mit Cuda **15**

A Testaufrufe **16**

B Liste der Dateien **20**

C Ausgewählte Quellen **21**
C.1 Matrixmultiplikation . . . 21
C.2 Gauß-Elimination . . . 25

1 Einleitung

Zunächst wird ein Algorithmus zur parallelen Matrizenmultiplikation und dessen Umsetzung auf CUDA vorgestellt. Die Laufzeiten der Implementierung werden anschließend mit denen einer CPU-Implementierung verglichen. Da der Algorithmus nicht für alle Problemstellungen geeignet ist, wird entsprechend auf Alternativen verwiesen.

Im Weiteren wird eine parallele Gauß-Elimination präsentiert. Diese wird im Projekt dazu verwendet um die Determinante einer Matrix zu bestimmen und um lineare Gleichungssysteme zu lösen. Zum Lösen der Gleichungssystem sind weitere Algorithmen notwendig, die hier nicht im Detail ausgeführt werden. Die Laufzeiten einzelner Teilalgorithmen werden im Bezug zur Gesamtlaufzeit untersucht und es wird auch der Vergleich mit einer CPU-Implementierung gezogen.

2 Matrizenmultiplikation

2.1 Algorithmus

Generell lässt sich die Matrizenmultiplikation sehr gut parallelisieren, da jeder Eintrag in der Zielmatrix von allen anderen Einträgen unabhängig berechnet werden kann. Seien A, B Matrizen mit $A \in \mathbb{R}^{m \times l}$ und $B \in \mathbb{R}^{l \times n}$. Dann kann das Produkt $A \cdot B = C \in \mathbb{R}^{m \times n}$ in $\mathcal{O}(m \cdot l \cdot n)$ Schritten berechnet werden.

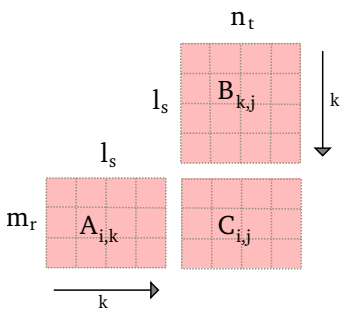
Die damit verbundenen $\mathcal{O}(m \cdot l \cdot n)$ Speicherzugriffe stellen den Flaschenhals bei der Implementierung mit CUDA dar. Ziel ist es, möglichst viele dieser Zugriffe mit Hilfe von Caching beschleunigen zu können. Dazu wird die Multiplikation blockweise durchgeführt.

Die Matrizen A und B werden komplett in Blöcke der Größen $r \times s$ bzw. $s \times t$ zerlegt. Dabei sei $l_s = l/s$ die Anzahl der Blöcke in die A horizontal bzw. B vertikal zerlegt wird. Dann lässt sich der (i, j)-te Block der Zielmatrix berechnen mit $C_{i,j} = \sum_{k=1}^{l_s}(A_{i,k} \cdot B_{k,j})$. Die Zielmatrix ist somit unterteilt in $m_r \times n_t$ Blöcke der Größe $r \times t$.

Alle $C_{i,j}$ können unabhängig berechnet werden. Daher lässt sich folgender parallele Algorithmus formulieren:

for $i \leftarrow 1$ *to* m_r **do in parallel**
  for $j \leftarrow 1$ *to* n_t **do in parallel**
    $C_{i,j} \leftarrow (0)$
    for $k \leftarrow 1$ *to* l_s **do**
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \cdot B_{k,j}$
    end
  end
end

Algorithmus 1: parallele Matrizenmult.



Dieser Algorithmus lässt sich hervorragend mit CUDA implementieren, indem die innerste Schleife als Kernel umgesetzt wird. Die beiden äußeren Schleifen werden durch die entsprechende Konfiguration beim Aufruf des Kernels ersetzt.

2.2 Implementierung

Es wurde ein Kernel implementiert, der eine Blockmatrix von C berechnet. Dazu werden drei Matrizen im Shared-Memory verwendet. Zwei, um Blockmatrizen aus A und B und eine weitere um die kumulierten Produkte dieser Teilmatrizen zu halten. Seien sA, sB und sC diese Matrizen. Dann sieht der Kernel folgendermaßen aus:

shared sA
shared sB
shared sC
$sC \leftarrow (0)$
for $k \leftarrow 1$ *to* l_s **do**
  load sA_k
  load sB_k
  $sC \leftarrow sC + sA_k \cdot sB_k$
end
store sC

Algorithmus 2: Ein Kernel für die Matrizenmultiplikation

Dabei werden die Operationen **load** und **store** sowie die Berechnung von sC parallel von den Threads innerhalb eines Threadblocks ausgeführt.

Größe der Blockmatrizen Je größer die Blockmatrizen im Shared-Memory gewählt werden, desto größer wird der Durchsatz des Algorithmus, da die Anzahl der insgesamt nötigen Zugriffe auf den globalen Speicher sich reziprok zu dieser Größe verhält. Der Shared-Memory hat jedoch eine feste Größe von 16 kB, wodurch sich bei der Verwendung quadratischer Blockmatrizen eine maximale Dimension von 36×36 ergibt.

Ein anderes Argument spricht jedoch gegen diese Dimensionierung. Die Operationen auf den Feldern müssen von den Threads des Threadsblocks ausgeführt werden. Die hierfür benötigten Adressierungsindizes müssen aus den Threadindizes hergeleitet werden. Werden hier viele Fallunterscheidungen getroffen oder der Modulo-Operator verwendet, so schlägt sich dies immens in der Laufzeit nieder. Daher sollten die Adressierungsindizes simpel herleitbar sein. Dies wird möglich, indem der Threadblock entrsprechend mit einem zur Feldgröße passend dimensionierten Threadgitter gestartet wird.

Die Anzahl der Threads im Threadgitter beeinträchtigt die Performance jedoch ebenfalls. So ist zu beachten, dass möglichst ein Vielfaches oder ein Teiler der Warpgröße als Gesamtzahl der Threads gewählt wird um die Auslastung der Multiprozessoren zu maximieren. Zudem ist es wesentlich einfacher, die Verteilung der Zugriffe auf die Speicherbänke des Shared-Memory effizient zu verteilen, wenn zumindest eine von zwei Dimensionen ein Vielfaches oder ein Teiler von 16 ist. Daher ist es sinnvoll Blockmatrizen der Größe 32×32 zu verwenden.

Größe des Threadblocks Die Größe des Threadblocks sollte also zur Größe der Blockmatrizen passen, um eine leichte Adressierbarkeit der Felder zu gewährleisten. Zusätzlich sollte die Anzahl der Threads möglichst hoch sein, damit der Scheduler geschicktes Pipelining betreiben kann, falls Operationen auftreten, die mehrere Taktzyklen lang dauern (wie bspw. Zugriff auf den globalen Speicher). Andererseits sollte sie auch nicht zu hoch sein, um den Overhead zu vermeiden, der entsteht wenn der Scheduler unnötig tätig ist. Am besten ließen sich hier experimentell Werte bestimmen, für die sich optimale Laufzeiten ergeben.

2.3 Verfeinerung

Der beschriebene Algorithmus geht davon aus, dass die Dimensionen der Eingabematrizen Vielfache der Blockgröße sind. Um den Algorithmus zu verallgemeinern, werden die Ein- und Ausgabematrizen mit Nullzeilen und Nullspalten erweitert, so dass sich eine äquivalente Matrizenmultiplikation ergibt, die vom Algorithmus berechnet werden kann.

Würden die Matrizen tatsächlich mit Nullen aufgefüllt, so würde dies das Umkopieren der Matrizen mit sich bringen. Um dies zu vermeiden werden die Matrizen lediglich virtuell aufgefüllt. Dazu wird der Kernel so abgeändert, dass die Operationen **load** and **store** in jenen Bereichen, wo eigentlich keine Einträge in den Originalmatrizen vorhanden sind, die Blockmatrizen mit Nullen initialisiert bzw. keine Werte zurückgeschrieben werden.

Hierfür ist jedoch bedingter Code im Kernel nötig, welcher die Ausführungszeit in etwa auf das Doppelte ausbremst. Allerdings ist es gar nicht nötig, diesen Kernel im gesamten Algorithmus zu verwenden, sondern lediglich an den Stellen, wo Blockmatrizen aus den Randbereichen auftreten. Wo dies der Fall ist, ist zum Zeitpunkt des Aufrufs des Kernels bekannt und kann entsprechend berücksichtigt werden.

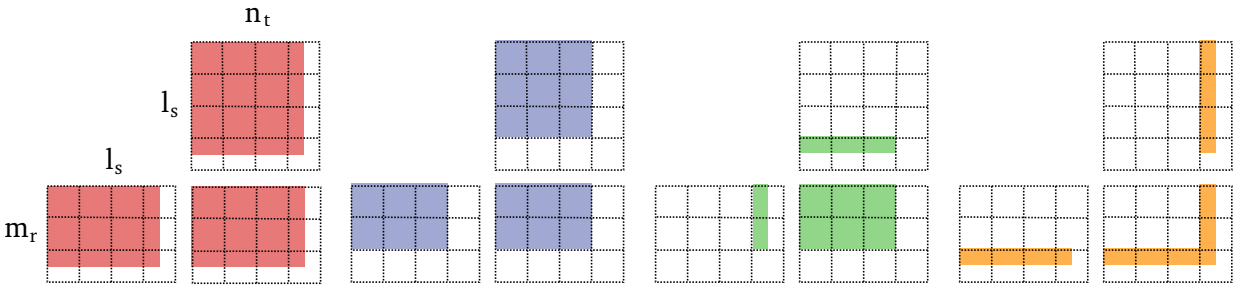


Abbildung 1: Kernelaufrufe bei Matrizen mit Dimensionen die kein Vielfaches der Blockgröße sind

Abbildung 1 zeigt, wie eine solche Matrizenmultiplikation (rot) auf verschiedene Kernel verteilt werden kann. Zunächst können die ersten $m_r - 1 \times n_t - 1$ Blöcke der Zielmatrix berechnet werden. In den ersten $l_s - 1$ (blau) Schritten kann hier ein Kernel verwendet werden, der völlig ohne Fallunterscheidungen auskommt. Lediglich der l_s-te Schritt kann Matrizen enthalten, die aufgefüllt werden müssen. Für diesen Fall wird ein zweiter Kernel (grün) verwendet. Der verbleibende Bereich (gelb) wird komplett über einen dritten Kernel abgedeckt. Der zweite und dritte Kernel benötigen Fallunterscheidungen.

Diese Aufteilung bewirkt, werden die Matrizen nur groß genug, dass asymptotisch der schnelle erste Kernel für die Ausführungszeit ausschlaggebend ist.

2.4 Evalutation

Die Abbildungen 2 und 3 stellen die Ausführungszeiten dreier Algorithmen gegenüber. Die zugrunde liegenden Daten entsammen der Multiplikation quadratischer Matrizen, wobei die Datenpunkte je 8 Einheiten entfernt liegen.

Bei der CPU-Variante handelt es sich um ein blockweises Verfahren, welches auf einem einzelnen Rechenkern ausgeführt wird. Dieses wird als Vergleichswert herangezogen, da dieses gegenüber einer einfachen seriellen CPU-Variante wesentlich schneller ist.

Die treppenförmige Erscheinung der CPU-Kurve ist durch das blockweise Verfahren erklärbar. Die Ausführungszeit ist für Matrizengrößen gleich, bei denen gleichviele Blöcke benötigt werden. Bei einer Blockgröße von 32 gibt es einen solchen Sprung immer nach 32 Elementen.

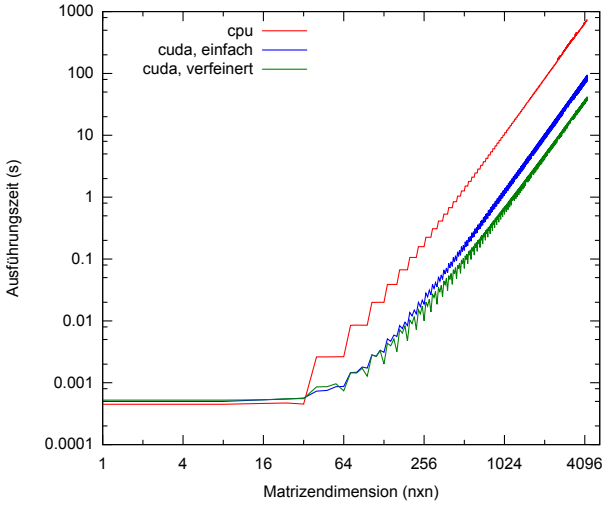


Abbildung 2: Ausführungszeiten

Die beiden CUDA-Kurven weisen periodisch eine Zick-Zack-Struktur auf. Da die Blockgröße hier ebenfalls je 32 beträgt, wäre eigentlich ein Bild wie bei der CPU-Kurve zu erwarten. Bei der einfachen Variante, lässt sich dies nur durch die Warp-Serialisierung ausgelöst durch Fallunterscheidungen erklären. An der Anzahl der Speicherzugriffe zumindest kann es nicht liegen, da diese streng monoton steigend sind und somit keine Sprünge nach unten rechtfertigen.

Bei der verfeinerten Variante tritt immer dort, wo ein Vielfaches der Blockgröße erreicht wird, ein plötzlicher Abfall der Kurve statt — dies deckt sich mit der Tatsache, dass in diesem Fall ausschliesslich der effizienteste Kernel verwendet wird. Die weiteren Ausschläge der Kurve erklären sich wie bei der einfachen Variante.

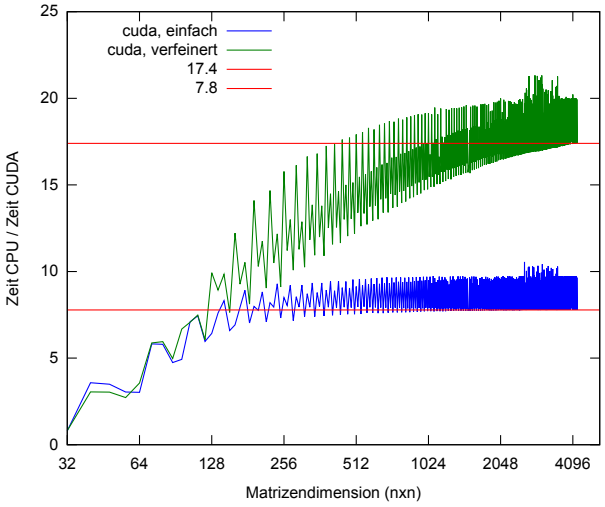


Abbildung 3: Vergleich mit der CPU

Wie schon in Abbildung 2 zu erkennen ist, sind beide Implementierungen in CUDA für ausreichend große Matrizen schneller als die CPU-Implementierung. Abbildung 3 zeigt, *wie viel* schneller die jeweilige CUDA-Implementierung gegenüber der CPU-Implementierung sind. Die untere Schranke für die Beschleunigung konvergiert dabei für die einfache Variante in etwa gegen 8 und für die verfeinerte Variante in etwa gegen 17.

2.5 Bewertung

Die vorliegende Implementierung erzielt überraschend gute Ergebnisse in der Laufzeit. Darüberhinaus sollte sie auch mit einer steigenden Anzahl an verfügbaren Prozessoren skalieren. Die Aufteilung auf Threadblöcke und damit auch die Aufteilung auf physische Multiprozessoren ist jedoch davon abhängig, dass sich die Zielmatrix in hinreichend viele Blöcke unterteilen lässt.

Ist diese Voraussetzung nicht gegeben, können nur begrenzt viele Multiprozessoren eingesetzt werden. Im Extremfall, einer Zielmatrix mit einer Dimension $\leqq 32 \times 32$, ist es gar nur möglich einen einzigen Multiprozessor zu verwenden. In einem solchen Fall könnte abgeholfen werden, indem die Blockgröße verringert würde, wodurch die Anzahl der Blöcke steigen würde.

Auch dieses Vorgehen ist jedoch nur begrenzt anwendbar. Beim Skalarprodukt beispielsweise lässt sich überhaupt nicht nach Blöcken der Zielmatrix parallelisieren. Hier müssten vollkommen andere Parallelisierungsstrategien verwendet werden. Vorstellbar wäre hier eine Parallelisierung nach Blöcken der Ausgangsmatrizen. Da dies aber eine völlig andere Implementierung mit sich bringen würde, wurden diese Fälle nicht eingehend behandelt.

3 Gleichungssysteme & Determinante

Für das Lösen von Gleichungssystemen wurde die Gauß-Jordan-Elimination implementiert. Die Implementierung der beiden Teilschritte dieses Verfahrens erfolgte separat, sodass die Gauß-Elimination allein für das Berechnen der Determinante verwendet werden kann.

Die Algorithmen arbeiten zunächst nur auf einer Matrix. Die Gauß-Elimination bringt eine Matrix in strikte obere Dreiecksform. Die Jordan-Elimination kann anschließend angewendet werden und eliminiert die Elemente oberhalb der Hauptdiagonalen.

Beide Algorithmen erhalten die Koeffizienten, die während der Verfahren auftreten und speichern diese innerhalb der Matrix ab. Dazu werden diejenigen Einträge der Matrix verwendet, die bekanntermaßen nur Nullen enthalten. Mit Hilfe dieser Koeffizienten können im nachhinein Gleichungssysteme gelöst werden, die aus der ursprünglichen Matrix und einem assoziierten Zielvektor bestehen. Es müssen lediglich die Operation aus den Eliminationen auf den Vektor angewandt werden und der Lösungraum kann mit vergleichsweise geringem Aufwand abgelesen werden.

Durch diese Trennung von Matrixmanipulation und Lösungsbestimmung können mehrere Gleichungssysteme, die sich nur im Zielvektor unterscheiden, durch einmaliges Bestimmen der Koeffizientenmatrix gelöst werden.

Um den Einfluss der auftretenden Rundungsfehler während der Gauß-Elimination möglichst gering zu halten, wird eine Pivotisierung durchgeführt. Da eine vollständige Pivotisierung die hier vorgestellte Parallelisierungsmethode ausschließen würde, wird eine Teilpivotisierung über die Spalten angewendet. Eine solche Teilpivotisierung führt in der Praxis in der Regel zu annehmbaren Ergebnissen [2].

Ziel war eine Implementierung für Fließkommazahlen. Um die Korrektheit der Algorithmen besser überprüfen zu können, wurden diese zunächst für Restklassenkörper implementiert, da hier immer exakte Ergebnisse berechnet werden können.

3.1 Gauß-Elimination

3.1.1 Algorithmus

Die Gauß-Elimination wird ebenfalls Blockweise durchgeführt und ist an [1] angelehnt. Sei $A \in \mathbb{K}^{m \times n}$ eine Matrix und b die Blockgröße. Dann wird, beginnend mit $A_0 = A$, zunächst auf den ersten b Zeilen von A_i eine gewöhnliche Gauß-Elimination durchgeführt. Danach werden mit Hilfe dieser b Zeilen die ersten b Spalten der verbleibenden Matrix eliminiert. Dieses Verfahren wird auf die um b Zeilen und Spalten kleinere Matrix A_{i+1} erneut angewendet, solange bis die Gauß-Elimination auf der gesamten Matrix durchgeführt wurde.

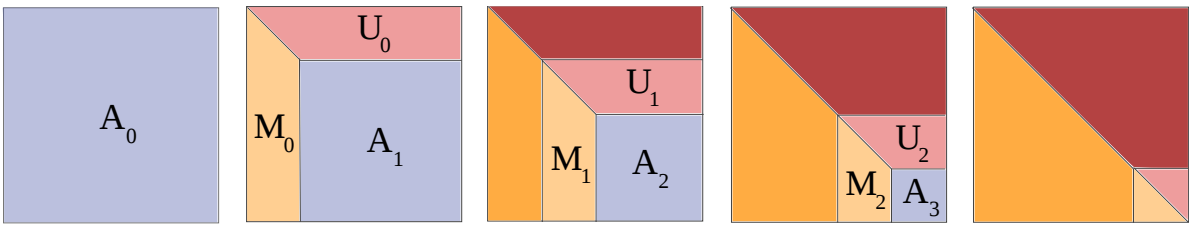


Abbildung 4: Iterationen der blockweisen Gauß-Elimination

Der entscheidende Effizienzgewinn liegt hier darin, dass sich das Anwenden von je b Zeilen auf die verbleibende Matrix als Matrizenmultiplikation darstellen lässt.

Beim Schritt $A_i \to A_{i+1}$ werden zunächst die ersten b Zeilen (U_i) berechnet. Daraufhin werden die Multiplikatoren für diese b Zeilen (M_i) berechnet. Gemeint sind hiermit diejenigen Faktoren, mit denen eine Zeile multipliziert wird, um einen Eintrag in einer darunterliegenden Zeile zu eliminieren. Die Multiplikatoren stehen in den ersten b Spalten von A_i. Daraufhin müssen noch die b Zeilen des Blocks auf A_{i+1} angewendet werden. Hier wird die Matrizenmultiplikation angewandt, denn $A_{i+1} = A^i_{i+1} + M_i \cdot U_i$. Dabei meint A^i_{i+1} die Matrix A_{i+1} welche noch mit den Werten aus Iteration i befüllt ist.

Während sich die Berechnungen von U_i und M_i ohne weiteres nur mittelmäßig effizient mit CUDA parallelisieren lassen, lässt sich die Matrizenmultiplikation hervorragend parallelisieren. Asymptotisch überwiegt der Arbeitsaufwand für die Matrizenmultiplikation deutlich, wodurch der Algorithmus insgesamt gut parallel durchführbar ist.

3.1.2 Probleme

Eliminierung Der obige Algorithmus funktioniert, solange im Schritt k immer ein von Null verschiedener Eintrag (k, k) auf der Diagonalen vorliegt. Da im Allgemeinen hiervon aber nicht ausgegangen werden kann, führt der Algorithmus unter Umständen einen Spaltentausch aus, um diese Eigenschaft herzustellen.

Ist die k-te Zeile jedoch eine Nullzeile, so schlägt auch dieses Vorgehen fehl. In diesem Fall wird ein Zeilentausch vorgenommen bei dem die k-te Zeile gegen die letzte Zeile getauscht wird, die noch linear unabhängig sein könnte. Dies wird so lange wiederholt, bis entweder eine Zeile gefunden wurde, mit der fortgefahren werden kann oder bis keine linear unabhängigen Zeilen mehr vorhanden sind. Auf diese Weise sammeln sich alle Nullzeilen am unteren Ende der Matrix. Zwar könnte eine Nullzeile auch stehengelassen werden, jedoch erhielte man so keine strikte obere Dreiecksform, welche für die Jordan-Elimination besonders günstig ist.

Pivotisierung Nachdem der Eintrag (k, k) von Null verschieden ist, kann eine Spaltenpivotisierung durchgeführt werden. Im Prinzip müsste die gefundene Pivotzeile p lediglich mit Zeile k vertauscht werden.

Jedoch würde sich dies im Weiteren negativ auf die Parallelisierbarkeit des Verfahrens auswirken. Hierzu wird kurz ausgeholt:

Der Algorithmus teil sich grob in zwei Phasen:

- Phase 1: Berechnen der Blöcke U_i und M_i
- Phase 2: Berechnen der Restmatrix A_{i+1}

Um den Algorithmus nicht zu komplex werden zu lassen, wird während Phase 1 so vorgegangen, dass eine Zeile k des Blocks U_i immer zunächst in einen Zustand versetzt wird in dem gilt, dass genau $k-1$ Zeilen auf diese Zeile aufaddiert wurden. Für alle weiter unten liegenden Zeilen gilt, dass bisher $(i-1)*b$ Zeilen aufaddiert wurden.

Deshalb unterscheiden sich die beiden an der Pivotisierung beteiligten Zeilen k und p in der Anzahl der bisher aufaddierten Zeilen. Ein einfacher Tausch der beiden Zeilen würde also dazu führen, dass der Bereich A_{i+1} sich nicht mehr für die Weiterverarbeitung mittels Matrizenmultiplikation in Phase 2 eignen würde.

Um dies zu vermeiden wird ein Dreiertausch vorgenommen in den die $k+1$-te Zeile einbezogen wird und mit dem die Permutaion $(p, k, k+1)$ dieser drei Zeilen hergestellt wird.

Da die $k+1$-te Zeile in der i-ten Iteration noch nicht verändert wurde, ist der Bereich A_{i+1} weiterhin in einem sinnvollen Zustand. Einzige Ausnahme bildet die ursprünglich k-te Zeile, welche sich nun an Position $k+1$ befindet. Im weiteren Verlauf von Phase 1 kann diese Zeile jedoch recht einfach behandelt werden.

Es kann maximal eine Zeile geben, auf die — durch die Pivotisierung bedingt — bereits mehr Zeilen aufaddiert wurden als zunächst geplant. Ist Phase 1 abgeschlossen und es existiert eine solche Zeile, so wird der Bereich A_{i+1} um diese Zeile verkleinert wodurch Phase 2 ohne Probleme durchgeführt werden kann.

3.1.3 Implementierung

Die Implementierung der Gauß-Elimination lässt sich vereinfacht folgendermaßen darstellen:

```
foreach Block-Iteration do
    foreach Zeile k im Block do
        sichereZeile k
        if k ist Nullzeile then
            return
        else
            aktualisiereSpalte k
            pivotisiere über Spalte k
            berechne Multiplikatoren für Zeile k in Spalte k
        end
    end
    berechne Restmatrix über Matrixmultiplikation
end
```

Algorithmus 3: Blockweise Gauß-Elimination

Die Methode **sichereZeile** angewendet auf eine Zeile k bewirkt, dass nach dem Aufruf gilt, dass alle vorhergenden Zeilen auf Zeile k aufaddiert wurden.

Nach einem Aufruf von **aktualisiereSpalte** auf eine Spalte k ist sichergestellt, dass auf alle Elemente dieser Spalte das jeweils k-te Element aller zuvor bearbeiteten Zeilen bereits addiert wurde. Dies ist notwendig um eine Pivotisierung über die Spalte ausführen zu können.

CUDA Das Hauptaugenmerk effizienter Implementierung liegt hier auf dem Kernel zur Matrizenmultiplikation. Dieser wurde im Prinzip so implementiert wie die einfache Variante, die im Abschnitt zur Matrizenmultiplikation beschrieben wurde.

Jedoch müssen auch die anderen Operationen für die Grafikkarte implementiert werden. Diese wurden so ausgelegt, dass sie auf beliebig vielen Multiprozessoren laufen können und jeweils eine feste Anzahl Threads innerhalb dieser verwenden. Dadurch soll gewährleistet werden, dass auch diese Algorithmen in ihrer Ausführungszeit beschleunigt werden, falls mehr Multiprozessoren zur Verfügung stehen.

Alle dieser Operationen arbeiten auf einem eindimensionalen Feld. Die Aufteilung auf Threadblöcke passiert indem jedem Threadblock ein Bereich dieses Felds zugeordnet wird. Innerhalb des Threadblocks geschieht die Aufteilung auf Threads meist durch eine Schleife, die über das Feld iteriert und dabei einen eindimensionalen Threadindex bei der Feldadressierung mit einbezieht. Bei manchen Kerneln konnte außerdem zur leichten Verbesserung der Shared-Memory verwendet werden.

Da diese Algorithmen vergleichsweise naiv implementiert wurden, wird auf eine genauere Darstellung verzichtet.

3.2 Jordan-Elimination und Bestimmung des Lösungraums

Als zweiter Schritt zur Lösung eines Gleichungssystems wird die Jordan-Elimination ausgeführt. Daraufhin kann der Lösungsraum bestimmt werden, indem eine Lösung für das spezielle Gleichungssystem als Stützvektor gefunden wird und über das assoziierte homogene Gleichungssystem Basisvektoren für den Lösungsraum ermittelt werden.

Aus Zeitgmangel wurde darauf verzichtet, für diese Probleme hocheffiziente Algorithmen umzusetzen. Es wurden lediglich einfache Algorithmen implmentiert, die nur einen Threadblock verwenden, daher könnte hier also noch mehr herausgeholt werden.

Problematisch wird diese ineffiziente Implementierung bei vieldimensionalen Lösungsräumen, da hier viel Rechenaufwand bei diesen Algorithmen anfällt. Eine Alternative zur Implementierung besserer Algorithmen stellt insbesondere bei der Bestimmung des Lösungsraums die Berechnung auf der CPU dar.

Die Jordan-Elimination ließe sich ähnlich der Gauß-Elimination blockweise durchführen. Auf diejenigen Spalten, die nicht eliminiert werden konnten, müssen wiederum alle darunterliegenden Zeilen angewendet werden. Die Umsetzung wäre vergleichsweise einfach, da hier schon alle Diagonalelemente von Null verschieden sind und keine Pivotisierung durchgeführt werden kann.

3.3 Evaluation

Abbildung 5 stellt die Ausführungszeit der CPU-Implementierung der CUDA-Implementierung gegenüber. Grundlage für die Messungen sind Gleichungssyteme der Größe $n \times (n+1)$ für die es mindestens eine Lösung gibt. Es ist zu erkennen, dass die CPU bei kleineren Gleichungssystemen zunächst schneller ist als CUDA. Ab einer Matrixdimension von etwa 500 holt CUDA die CPU jedoch ein und baut für größere Matrizen einen Vorsprung auf. Genauer lässt sich das Verhältnis der Laufzeiten in Abbildung 6 ablesen. Für große Matrizen pendelt sich die Beschleunigung gegenüber der CPU auf etwa 2,5 ein.

Desweiteren lässt sich erkennen, auf welche Teilschritte des Algorithmus sich die Laufzeit aufteilt. Ange-

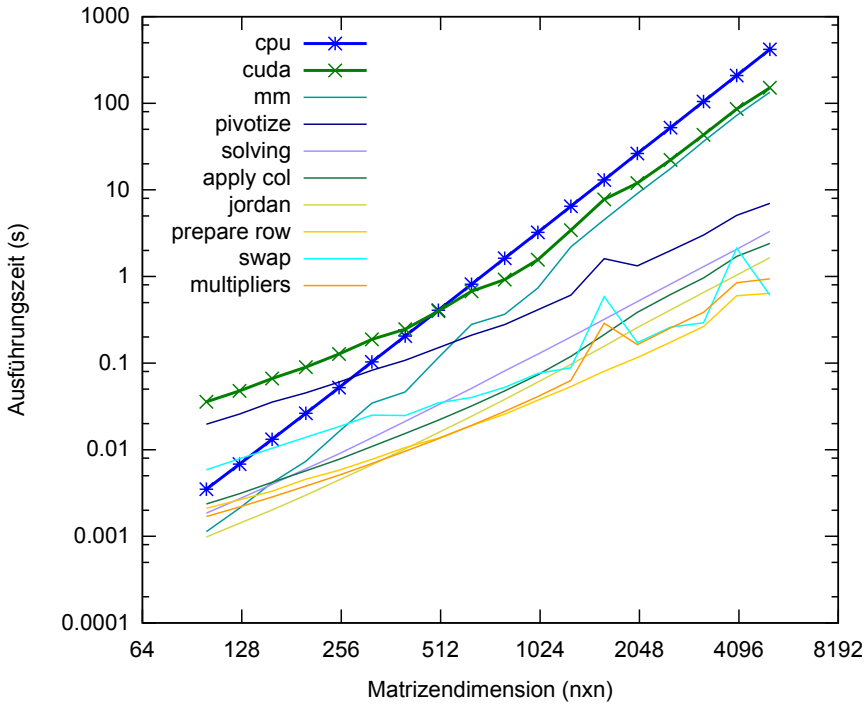


Abbildung 5: Ausführungszeiten

geben sind hier sämtliche Kernel, die im Algorithmus verwendet werden. Während zunächst der Kernel für die Pivotisierung überwiegt, verliert dieser für größere Probleme an Bedeutung und die Matrizenmultiplikation entwickelt sich zum ausschlaggebenden Faktor für die Laufzeit des Gesamtalgorithmus.

3.4 Determinante

Die Determinante kann ermittelt werden, nachdem die Gauß-Elimination angewendet wurde. Hierfür wurde kein besonderer Algorithmus implementiert. Das Produkt der Diagonalen wird von der CPU nach der Überführung der modifizierten Matrix in den Hauptspeicher berechnet. Das Vorzeichen kann bestimmt werden, da die Eliminationsalgorithmen die Anzahl der Tauschvorgänge zurückgeben. Da die Laufzeit für die Determinantenbestimmung sich primär aus der Laufzeit der Gauß-Elimination ergibt, wird auf eine genauere Darstellung verzichtet.

3.5 Bewertung

Die Implementierung der Gauß-Elimination ist meiner Ansicht nach gelungen, da auch diese Lösung — da sie auf der Matrizenmultiplikation aufbaut — mit einer steigenden Anzahl an Multiprozessoren skalieren sollte.

Da allerdings nur ein einfacher Kernel (vgl. Abschnitt 2.3) für die Matrizenmultiplikation verwendet wird, wäre eine offensichtliche Verbesserung, auch hier einen verfeinerten Kernel zu verwenden, welcher die Ausführungszeit der Multiplikation asymptotisch halbieren sollte.

Die auftretenden Zeilen- und Spaltentauschoperationen werden tatsächlich ausgeführt. Auch wenn hierfür vergleichsweise wenig Zeit gebraucht wird, ist doch fraglich ob es nicht besser wäre, diese Operationen ausschließlich über eine Indextabelle zu realisieren. Ob dies signifikant negative Auswirkungen auf andere

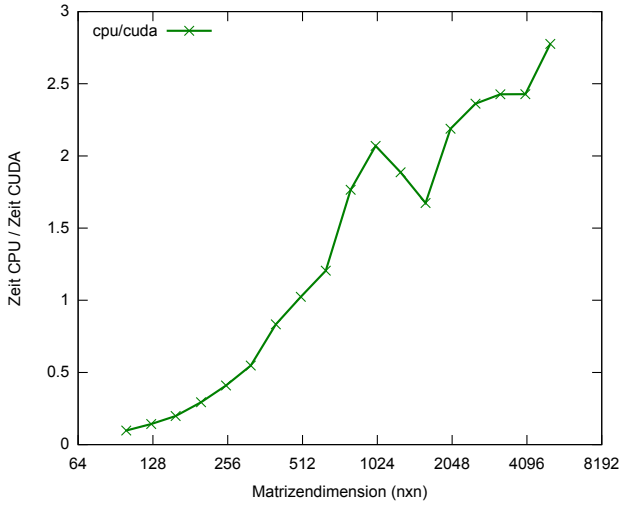


Abbildung 6: Vergleich mit der CPU

Kernel hätte ist nicht erforscht worden, ist aber vorstellbar.

Offen sind desweiteren effizientere Implementierung für Jordan-Eliminierung und Bestimmung des Lösungsraums.

4 Projekt- und Testumgebung

Das Projekt wurde auf den Rechnern im Linux-Rechnerpool entwickelt und kann dort fehlerfrei kompiliert werden. Alle Tests wurden auf den dort verbauten *nVidia Quadro NVS 290* durchgeführt. Diese Grafikkarten unterstützen *Compute Capability* 1.1 und haben 16 *Cuda Cores* verteilt auf 2 Multiprozessoren. Die CPU-Vergleichstests wurden auf einem der vier Kerne eines *Intel Quadcore Q9550* mit 2.83 GHz durchgeführt.

4.1 Verzeichnisbaum

Im Hauptprojektverzeichnis finden sich die Ordner *bin, doc, src* und *tools*

src Hier finden sich die gesamten Quelltexte.

- **matrix** — stellt die Basistypen *Matrix, CudaMatrix, MatrixInt* und *CudaMatrixInt* und deren Speicherverwaltungsfunktionen zur Verfügung. Außerdem finden sich hier Hilfsfunktionen um Matrizen mit Nullen oder zufälligen Werten zu initialisieren und um Matrizen zwischen Host- und Devicespeicher zu transferieren. Daneben gibt es Makros, welche den Zugriff auf Matrixeinträge vereinfachen, Methoden zur Ausgabe von Matrizen auf der Konsole und Methoden, die zwei Matrizen (unter Berücksichtigung eines Fehlers) abgleichen.
- **modulo** — stellt Datenstrukturen und Operationen für das Rechnen mit Restklassenkörpern zur Verfügung. Wichtig sind hier die Typen *InvTable* und *CudaInvTable*, welche für das Arbeiten mit linearen Gleichungssystem verwendet werden, um effizient das multiplikative Inverse nachschlagen zu können.

- **matrix_multiplication** — hier finden sich die Implementierungen der Matrizenmultiplikation sowohl für die CPU als auch für CUDA. Zentral sind hier die Dateien *mm_advanced.cu*, in der die GPU-Implementierung steht sowie *testing.cu*, welche verschieden Testroutinen zur Verfügung stellt.
- **linear_systems** — teilt sich in drei Unterverzeichnisse: *generic, float* und *modulo*. Die Inhalte der Verzeichnisse *float* und *modulo* sind aus dem generischen Quelltext generiert worden. Zentral ist hier die Datei *linsys.cu*, welche die Implementierung der Gauß-Elimination bereitstellt. Einen guten Einstieg sollte auch hier die Datei *testing.cu* bieten.

bin enthält alle ausführbaren Dateien und zusätzlich Skripte, welche die Bedienung der Artefakte erleichtern.

doc Hier liegen nebem dem Bericht noch per *Doxygen* erzeugte Dokumentationen zu den Quelltexten. Diese können dazu dienen, sich einen Überblick über die vorhandenen Dateien und Methoden zu verschaffen.

4.2 Build-Prozess

Die Erstellung der ausführbaren Artefakte ist über ein *Makefile* geregelt, wodurch sich sämtliche Artefakte über den Aufruf des Kommandos **make** erzeugen lassen. Da der Compiler von *nVidia* allerdings nicht die Möglichkeit bietet, bestimmte Warnungen mit zweifelhaftem Nutzen zu deaktivieren, empfiehlt sich das Kompilieren über das Skript *compile*, welches diese Warnmeldungen ausblendet, aber alle sinnvollen Warnmeldungen und Hinweise erhält.

Die Quelltexte für die linearen Gleichungssysteme werden über eine Art Präprozessor generiert. Eigentlich würde sich für diese Aufgabe ein genereller Präprozessor wie *gpp* eignen. Da dieser jedoch auf den Poolrechnern nicht intalliert ist, wird hier ein kleines Python-Skript verwendet — zu finden unter *tools/convert.py* — dessen Verwendungsweise aus dem *Makefile* klar wird. Diese Skript wird mittels der Datei *linear_systems/generic/conversion.conf* darüber informiert, welche Modifikationen es an einer Quelldatei durchführen soll.

4.3 Tests

im Verzeichnis *bin* finden sich sämtliche Tests.

- **matrix** lässt das Testen der Matrixbasisoperationen zu.
- **modulo** gibt wahlweise eine Multiplikationstabelle oder eine Tabelle mit den multiplikativen Inversen zu einem gegebenen Restklassenkörper aus.
- **mm_debug** führt eine einfache Matrixmultiplikation mit einem der Algorithmen aus. Dieses Programm dient primär zum Debuggen anhand kleiner Matrizen, da hier eine Ausgabe der Matrizen selbst möglich ist.
- **mm_performance** führt eine Reihe von Matrizenmultiplikationen aus und speichert die Ergebnisse bei Bedarf in einer Datei zur weiteren Verarbeitung.
- **mm_verification** ist ein Unit-Test für die Matrixbibliothek. Führt für Matrizen, deren Dimensionen zufällig aus einem konfigurierbaren Bereich gewählt werden, die Multiplikation durch und vergleicht die Ergebnisse mehrerer Algorithmen miteinander. Der Test läuft, bis die angegebene Anzahl Testläufe abgeschlossen ist oder ein Fehler auftritt.

- **linsys_debug** Wiederum ein Programm zum Berechnen meist kleiner linearer Systeme, welche zu Debugging-Zwecken auf der Konsole ausgegeben werden können.
- **linsys_performance** misst die Laufzeiten für das Lösen linearer Gleichungssystemen und speichert die Laufzeiten wahlweise ab.
- **linsys_verification** ebenfalls ein Unit-Test. Generiert Gleichungssysteme mit Lösung, findet den Lösungsraum und überprüft die Lösung durch Multiplikation mit der ursprünglichen Matrix. Bei den Restklassenkörpern kann die Gleichheit exakt festgestellt werden. Für Fließkommazahlen werden die Abweichungen zum Ausgangvektor gemessen. Es wird das Maximum bestimmt und der Mittelwert gebildet. Wenn der Mittelwert unter einer bestimmten Schwelle liegt, werden Ergebnisse als korrekt anerkannt.

5 Erfahrungen mit Cuda

Eingebaute Variablen Cuda stellt die Variablen `gridDim`, `blockIdx`, `threadIdx`, etc. zur Verfügung. Oftmals wird im Projekt bspw. eine Variable `tx` deklariert, die als Alias für `threadIdx.x` verwendet wird. An einigen Stellen erscheint deren Verwendung nicht unbedingt sinnvoll (bspw. weil diese Variable nur an einer einzigen Stelle verwendet wird). Leider zeigen die Algorithmen zum Teil schlechtere Performance wenn statt `tx` einfach `threadIdx.x` geschrieben wird. Ein Beispiel hierfür ist die Verwendung von `threadIdx.x` innerhalb eines Schleifenkörpers.

Generezität Um generische Methoden zu implementieren, wurde die Verwendung von Templates getestet. Diese bieten über sogenannte Nicht-Typ-Parameter, die Möglichkeit Methoden so zu gestalten, dass je nach Übergabewert verschiedener Code ausgeführt wird. Leider führt eine solche partielle Spezialisierung nicht zu den erwarteten Optimierungen des Compiliers. Über Templates lassen sich daher nur weniger effiziente Methoden erstellen als wenn für die Spezialisierung separate Methoden formuliert werden.

Literatur

[1] James Demmel. Dense linear algebra 2. Technical report, U.C. Berkeley, 03 2008.

[2] D. Stott Parker and Dinh Le. How to eliminate pivoting from gaussian elimination - by randomizing instead. Technical report, U.C. Los Angeles, 1995.

A Testaufrufe

Genaueres zu den Programmen kann mit dem Parameter '–help' bzw. '–usage' ermittelt werden. Außerdem stehen im Verzeichnis 'bin' einige Shell-Skripte, die exemplarische Aufrufe zeigen.

Matrizenmultiplikation

Debugging

```
$ ./mm_debug --mode=cuda --seed=32342345 -m 6 -l 8 -n4

testing CUDA ADVANCED with 6x8 * 8x4
.....................................................
Printing out Matrix: A
   4.0    3.0    3.0    2.0    1.0    0.0    4.0    3.0
   1.0    3.0    1.0    4.0    4.0    3.0    1.0    4.0
   1.0    2.0    3.0    4.0    3.0    3.0    0.0    4.0
   4.0    1.0    3.0    1.0    1.0    2.0    4.0    3.0
   0.0    2.0    1.0    4.0    3.0    2.0    2.0    4.0
   2.0    4.0    3.0    4.0    2.0    3.0    0.0    1.0
Printing out Matrix: B
   2.0    4.0    2.0    2.0
   2.0    4.0    3.0    2.0
   2.0    1.0    3.0    3.0
   0.0    2.0    0.0    0.0
   1.0    3.0    2.0    1.0
   2.0    0.0    0.0    2.0
   0.0    0.0    3.0    3.0
   0.0    3.0    1.0    0.0
Printing out Matrix: C
  21.0   47.0   43.0   36.0
  20.0   49.0   29.0   24.0
  21.0   44.0   27.0   24.0
  21.0   37.0   37.0   36.0
  13.0   38.0   25.0   20.0
  26.0   44.0   30.0   29.0
sebkur@shanghai:~/dev/cuda$
```

Verifikation

```
$ ./mm_verification --minM=200 --maxM=400 --minL=400 --maxL=800 --minN=300 --maxN=500 --
    ntests=3

============================================================
TEST: 368x558 . 558x614
============================================================
.....................................................
matrix multiplication on GPU
hblocks, vblocks 20 12
.....................................................
.....................................................
Checksum:             505551264.00
Checksum:             505551264.00
verification PASSED
CPU: total, memops:       1.327123, 0.009379
CUDA easy: total, memops: 0.267795, 0.077912
CUDA: total, memops:      0.118336, 0.003451
============================================================
TEST: 272x517 . 517x614
============================================================
.....................................................
matrix multiplication on GPU
hblocks, vblocks 20 9
.....................................................
```

```
............................................................
Checksum:             346476704.00
Checksum:             346476704.00
verification PASSED
CPU: total, memops:       0.941589, 0.007184
CUDA easy: total, memops: 0.136431, 0.002939
CUDA: total, memops:      0.084182, 0.002636
============================================================
TEST: 207x576 . 576x356
============================================================
............................................................
matrix multiplication on GPU
hblocks, vblocks 12 7
............................................................
............................................................
Checksum:             169627760.00
Checksum:             169627760.00
verification PASSED
CPU: total, memops:       0.465454, 0.003081
CUDA easy: total, memops: 0.056916, 0.002044
CUDA: total, memops:      0.031696, 0.001681
```

Performance

```
$ ./mm_performance --do-cpu --do-cuda-easy --do-cuda -o ../results/mm.output 400 800 400
    1000 1400 1000

============================================================
TEST: 400x800 . 800x400
============================================================
............................................................
matrix multiplication on GPU
hblocks, vblocks 13 13
............................................................
............................................................
Checksum:             512978496.00
Checksum:             512978496.00
verification PASSED
CPU: total, memops:       1.298438, 0.008633
CUDA easy: total, memops: 0.209665, 0.076032
CUDA: total, memops:      0.078637, 0.003026
============================================================
TEST: 1000x1400 . 1400x1000
============================================================
............................................................
matrix multiplication on GPU
hblocks, vblocks 32 32
............................................................
............................................................
Checksum:             5598142976.00
Checksum:             5598142976.00
verification PASSED
CPU: total, memops:       13.859840, 0.115332
CUDA easy: total, memops: 1.769727, 0.010544
CUDA: total, memops:      0.886748, 0.011288
sebkur@shanghai:~/dev/cuda/bin$
```

Lineare Gleichungssysteme

Debugging

```
$ ./linsys_debug --height=6 --width=10 --print-original-problem --print-cuda-
    modifications --print-cuda-solution --print-cuda-backmult

==================================
Testing size: 6x10
==================================
both algorithms found a solution
host deviations: total: 0.0003, max: 0.0001, mean: 0.0001
cuda deviations: total: 0.0002, max: 0.0001, mean: 0.0000
CPU: overall, swapping: 0.00, 0.00
CUDA: applyRow 0.00 swap 0.00 applyCol 0.00 pivotize 0.00 computeMults 0.00 MM 0.00
CUDA: jordan: 0.00, solving: 0.00
CPU time, CUDA time: 0.00, 0.00
Printing out Linear System: (A | b), original problem
  8.00   0.00   0.00   8.00   0.00   6.00   6.00   2.00   2.00   2.00  | 168.00
  8.00   4.00   4.00   4.00   2.00   4.00   4.00   8.00   4.00   4.00  | 220.00
  2.00   0.00   0.00   6.00   2.00   4.00   8.00   0.00   6.00   6.00  | 192.00
  4.00   6.00   8.00   2.00   6.00   2.00   8.00   6.00   0.00   0.00  | 168.00
  2.00   8.00   6.00   6.00   0.00   0.00   2.00   8.00   0.00   0.00  | 104.00
  4.00   0.00   6.00   8.00   0.00   6.00   4.00   8.00   8.00   0.00  | 192.00
Printing out Linear System: (A | b), modified by CUDA
  8.00  -0.00  -0.00   1.09  -0.00  -7.65   0.70  -8.84 -27.43 -21.98  | -355.78
 -0.25   8.00  -1.00   0.55  -0.00   1.91  -2.73  19.74  34.29  35.16  | 615.65
 -0.50  -0.00   6.00   0.55  -0.00  -3.82  -1.65   1.58  -7.71 -12.99  | -153.89
 -0.50  -0.75  -0.58  -7.33  -1.14   2.07  -7.58  13.58  24.10  20.39  | 328.55
 -0.25  -0.00  -0.00   0.55   5.27  -2.06   9.50  -9.97 -12.05  -6.95  | -101.17
 -1.00  -0.50  -0.17  -0.91   0.66   0.78  -0.39   3.35   7.28   6.04  | 115.34
Printing out vector-space: Cuda Solution
-44.47  76.96 -25.65 -44.80 -19.19 147.03   0.00   0.00   0.00   0.00

 -0.09   0.34   0.27  -1.03  -1.80   0.49   1.00   0.00   0.00   0.00

  1.10  -2.47  -0.26   1.85   1.89  -4.27   0.00   1.00   0.00   0.00

  3.43  -4.29   1.29   3.29   2.29  -9.29   0.00   0.00   1.00   0.00

  2.75  -4.40   2.16   2.78   1.32  -7.70   0.00   0.00   0.00   1.00
Printing out Matrix: A * (Cuda specific solution)
168.00 220.00 192.00 168.00 104.00 192.00
```

Verifikation

```
$ ./linsys_verification --base=int --ntests=3 --minwidth=800 --maxwidth=1000 --minheight
    =800 --maxheight=1000

Test #0, Size: 968x879
==================================
Testing size: 968x879
==================================
host solution is OK, cuda solution is OK
both solutions are the SAME
CPU: overall, swapping: 2.74, 0.00
CUDA: applyRow 0.05 swap 0.06 applyCol 0.08 pivotize 0.33 computeMults 0.05 MM 1.19
CUDA: jordan: 0.09, solving: 0.15
CPU time, CUDA time: 2.76, 2.00
Test #1, Size: 826x901
==================================
Testing size: 826x901
==================================
host solution is OK, cuda solution is OK
both solutions are the SAME
CPU: overall, swapping: 2.24, 0.00
CUDA: applyRow 0.05 swap 0.06 applyCol 0.07 pivotize 0.32 computeMults 0.04 MM 0.98
CUDA: jordan: 4.11, solving: 0.28
CPU time, CUDA time: 2.57, 5.90
```

```
Test #2, Size: 980x922
================================
Testing size: 980x922
================================
host solution is OK, cuda solution is OK
both solutions are the SAME
CPU: overall, swapping: 3.00, 0.00
CUDA: applyRow 0.05 swap 0.07 applyCol 0.09 pivotize 0.35 computeMults 0.05 MM 1.31
CUDA: jordan: 0.09, solving: 0.16
CPU time, CUDA time: 3.02, 2.18
```

Performance

```
$ ./linsys_performance --test=both --base=float -o ../results/linsys.output 500 500 1000
    1000 1500 1500

================================
Testing size: 500x500
================================
both algorithms found a solution
host deviations: total: 2.0835, max: 0.0186, mean: 0.0042
cuda deviations: total: 2.0811, max: 0.0166, mean: 0.0042
CPU: overall, swapping: 0.39, 0.00
CUDA: applyRow 0.01 swap 0.03 applyCol 0.02 pivotize 0.15 computeMults 0.01 MM 0.13
CUDA: jordan: 0.02, solving: 0.03
CPU time, CUDA time: 0.40, 0.40
================================
Testing size: 1000x1000
================================
both algorithms found a solution
host deviations: total: 10.2471, max: 0.0449, mean: 0.0102
cuda deviations: total: 11.2900, max: 0.0537, mean: 0.0113
CPU: overall, swapping: 3.13, 0.00
CUDA: applyRow 0.04 swap 0.07 applyCol 0.07 pivotize 0.41 computeMults 0.04 MM 0.91
CUDA: jordan: 0.06, solving: 0.12
CPU time, CUDA time: 3.15, 1.73
================================
Testing size: 1500x1500
================================
both algorithms found a solution
host deviations: total: 42.1680, max: 0.1289, mean: 0.0281
cuda deviations: total: 38.1602, max: 0.1133, mean: 0.0254
CPU: overall, swapping: 10.61, 0.00
CUDA: applyRow 0.07 swap 0.11 applyCol 0.18 pivotize 0.80 computeMults 0.09 MM 3.43
CUDA: jordan: 0.14, solving: 0.28
CPU time, CUDA time: 10.67, 5.09
```

B Liste der Dateien

Src

```
linear_systems/generic/conversion.conf          // Konfiguration fuer Generierung
linear_systems/generic/field.h                  // Koerper-Operationen
linear_systems/generic/linsys_cpu.cu            // CPU-Implementierung
linear_systems/generic/linsys.cu                // gauss in CUDA
linear_systems/generic/linsys.h                 // header fuer alles
linear_systems/generic/linsys_jordan.cu         // jordan in CUDA
linear_systems/generic/linsys_naive.cu          // naiver gauss in CUDA
linear_systems/generic/linsys_solving.cu        // Loesungsraumbestimmung CUDA
linear_systems/generic/linsys_wrappers.cu       // Convinience-Methoden
linear_systems/generic/testing.cu               // Test-Methoden
linear_systems/generic/testing.h                // Test-Header
linear_systems/generic/utility.cu               // diverses
linear_systems/generic/utility.h
linear_systems/generic/vector_space.cu          // Loesungsraum
linear_systems/generic/vector_space.h

matrix/matrix_float.cu                          // float-Matrizen
matrix/matrix.h                                 // alle matrix-header
matrix/matrix_int.cu                            // int-Matrizen

matrix_multiplication/mm_advanced.cu            // 3-Kernel MM
matrix_multiplication/mm_cpu.cu                 // CPU MM
matrix_multiplication/mm.cu                     // 1-Kernel MM
matrix_multiplication/mm.h                      // alle header
matrix_multiplication/testing.cu                // Test-Methoden
matrix_multiplication/testing.h                 // Test-Header
matrix_multiplication/wrappers.cu               // Convinience-Methoden

modulo/modulo.cu                                // Multiplikations- und
modulo/modulo.h                                 // Inversionstabellen CUDA + CPU

tests/linsys_debug.cu                           // Debuggen kleiner Systeme
tests/linsys_performance.cu                     // Performance-bestimmung
tests/linsys_verification.cu                    // verifikation vieler Systeme
tests/matrix.cu                                 // teste Matrixbasisoperationen
tests/mm_debug.cu                               // debuggen kleiner Matrizen
tests/mm_performance.cu                         // Performance
tests/mm_verification.cu                        // Verifikation
tests/modulo.cu                                 // test Modulobasisoperationen

tools/cuda_debug.h                              // Vereinfachung des Debugging
tools/darray.cu                                 // ein dynamisch wachsendes Array
tools/darray.h                                  // fuer Kommandozeilenparsing
tools/misc.h                                    // diverse Makros
tools/time_tools.cu                             // Zeitmessungsfunktionen
tools/time_tools.h
```

Bin

```
linsys_debug                                    // LGS-Tests
linsys_performance
linsys_verification
matrix                                          // Matrix-Tests
mm_debug                                        // MM-Tests
mm_performance
mm_verification
modulo                                          // Restklassen-Tests
profile_config                                  // Config fuer Cuda-Profiler
test_linsys1.sh                                 // Bsp. fuer linsys_performance
test_linsys2.sh
test_linsys3.sh
test_linsys_debug.sh                            // Bsp. fuer linsys_debug
test_linsys_verficiation.sh                     // Bsp. fuer linsys_debug
test_mm1.sh                                     // Bsp. fuer mm_performance
```

test_mm_verification.sh // Bsp. fuer mm_verification

C Ausgewählte Quellen

C.1 Matrixmultiplikation

Direktiven

```
#define NTHREADS 128
#define BLOCKW 32
#define BLOCKH 4
#define ITER 8
```

Koordination

```
/*****************************************************************************//**
 * fast version that only guards multiplications that overlap the actual matrix
 *****************************************************************************/
void multiply_cuda_advanced(CudaMatrix * A, CudaMatrix * B, CudaMatrix * C,
                int m, int l, int n)
{
        struct timeval t1, t2;
        double time_used = 0;
        TIME(&t1);

        int hblocks = n / BLOCKW;
        int vblocks = m / BLOCKW;

        bool fitL = MULTIPLE(l, BLOCKW);
        bool fitM = MULTIPLE(m, BLOCKW);
        bool fitN = MULTIPLE(n, BLOCKW);
        int marginBlocks = 0;
        if (!fitM)
                marginBlocks += (int) ceil(n / (float)BLOCKW);
        if (!fitN)
                marginBlocks += (m / BLOCKW);

        int x, y;

        cudaError_t err;

        for (y = 0; y < vblocks; y+=2){
                for (x = 0; x < hblocks; x+=2){

                        dim3 numBlocks = dim3(
                                MIN(2, hblocks - x),
                                MIN(2, vblocks - y));
                        dim3 threadsPerBlock(BLOCKW, BLOCKH);

                        CUDA_DEBUG_START(err);
                        mm<<<numBlocks,threadsPerBlock>>>(A, B, C, x, y);
                        CUDA_DEBUG_END(err, "mm stage 1");

                        if (!fitL){
                                CUDA_DEBUG_START(err);
                                mm_last<<<numBlocks,threadsPerBlock>>>(A, B, C, x, y);
                                CUDA_DEBUG_END(err, "mm stage 2");
                        }

                }
        }
        if (marginBlocks > 0){
                dim3 threadsPerBlock(BLOCKW, BLOCKH);
                CUDA_DEBUG_START(err);
                mm_guarded<<<marginBlocks,threadsPerBlock>>>(A, B, C, 0);
                CUDA_DEBUG_END(err, "mm stage 2");
        }

        TIME(&t2);
        time_used += time_diff(&t1, &t2);
}
```

Kernel

```
/*****************************************************************************//**
 * kernel for multiplying unguarded
 ********************************************************************************/
static __global__ void mm(CudaMatrix * A, CudaMatrix * B, CudaMatrix * C,
                int x_offset, int y_offset)
{
        __shared__ float rows[BLOCKW * BLOCKW]; // used to store A_k,y
        __shared__ float cols[BLOCKW * BLOCKW]; // used to store B_x,k
        __shared__ float temp[BLOCKW * BLOCKW]; // used to store C_x,y,
        // or rather the cumulation of A_ky * B_x,k for k = 0...rounds

        int l = A -> n_cols;
        int n = C -> n_cols;

        int k, x, y; // control variables
        int tx = threadIdx.x; // 0...31, short for threadIdx.x
        int ty = threadIdx.y; // 0...7 , short for threadIdx.y
        int tid = ty * BLOCKW + tx; // 32 x 8, threads are enumerated
        int rounds = l / BLOCKW; // number of submatrices to calculate and cumulate
        x = x_offset + blockIdx.x; // this block's submatrix-x-coordinate
        y = y_offset + blockIdx.y; // this block's submatrix-y-coordinate

        init_zero(temp, tid);
        __syncthreads();

        for (k = 0; k < rounds; k++){ // foreach C_x,y
                load(A, l, k, y, rows, tid);
                load(B, n, x, k, cols, tid);
                __syncthreads();

                mult(temp, rows, cols, tid, tx, ty);
                __syncthreads();
        }
        store(C, n, x, y, temp, tid);
}

/*****************************************************************************//**
 * kernel for multiplying the last of k steps guarded
 ********************************************************************************/
static __global__ void mm_last(CudaMatrix * A, CudaMatrix * B, CudaMatrix * C,
                int x_offset, int y_offset)
{
        __shared__ float rows[BLOCKW * BLOCKW]; // used to store A_k,y
        __shared__ float cols[BLOCKW * BLOCKW]; // used to store B_x,k
        __shared__ float temp[BLOCKW * BLOCKW]; // used to store C_x,y,

        int m = C -> n_rows;
        int l = A -> n_cols;
        int n = C -> n_cols;

        int x, y; // control variables
        int tx = threadIdx.x; // 0...31, short for threadIdx.x
        int ty = threadIdx.y; // 0...7 , short for threadIdx.y
        int tid = ty * BLOCKW + tx; // 32 x 8, threads are enumerated
        int rounds = l / BLOCKW; // number of submatrices to calculate and cumulate
        x = x_offset + blockIdx.x; // this block's submatrix-x-coordinate
        y = y_offset + blockIdx.y; // this block's submatrix-y-coordinate

        load(C, n, x, y, temp, tid);
        __syncthreads();

        load_guarded(A, m, l, rounds, y, rows, tid);
        load_guarded(B, l, n, x, rounds, cols, tid);
        __syncthreads();

        mult(temp, rows, cols, tid, tx, ty);
        __syncthreads();

        store(C, n, x, y, temp, tid);
}

/*****************************************************************************//**
 * kernel for multiplying guarded
 ********************************************************************************/
static __global__ void mm_guarded(CudaMatrix * A, CudaMatrix * B, CudaMatrix * C,
                int offset)
{
        __shared__ float rows[BLOCKW * BLOCKW]; // used to store A_k,y
        __shared__ float cols[BLOCKW * BLOCKW]; // used to store B_x,k
        __shared__ float temp[BLOCKW * BLOCKW]; // used to store C_x,y,
        // or rather the cumulation of A_ky * B_x,k for k = 0...rounds

        int m = C -> n_rows;
```

```
84          int l = A -> n_cols;
            int n = C -> n_cols;
86
            int k, x, y; // control variables
88          int tx = threadIdx.x; // 0...31, short for threadIdx.x
            int ty = threadIdx.y; // 0...7 , short for threadIdx.y
90          int tid = ty * BLOCKW + tx; // 32 x 8, threads are enumerated
            int rounds = ceilf(l / (float)BLOCKW); // number of submatrices to calculate and cumulate
92
            int bottomBlocks = MULTIPLE(m, BLOCKW)
94                  ? 0 : (int) ceil(n/ (float)BLOCKW);
            if (blockIdx.x < bottomBlocks){
96                  x = blockIdx.x;
                    y = m / BLOCKW;
98          }else{
                    x = n / BLOCKW;
100                 y = blockIdx.x - bottomBlocks;
            }
102
            init_zero(temp, tid);
104         __syncthreads();

106         for (k = 0; k < rounds; k++){ // foreach C_x,y
                    load_guarded(A, m, l, k, y, rows, tid);
108                 load_guarded(B, l, n, x, k, cols, tid);
                    __syncthreads();
110
                    mult(temp, rows, cols, tid, tx, ty);
112                 __syncthreads();
            }
114         store_guarded(C, m, n, x, y, temp, tid);
}
```

Device-Funktionen

```
/*****************************************************************************//**
 * initialize \p field with all zeros
 *****************************************************************************/
static __device__ void init_zero(float * field, int tid)
{
        int i;
        for (i = 0; i < ITER; i++){ // initialize C_x,y to 0.0
                field[i * NTHREADS + tid] = 0.0f;
        }
}

/*****************************************************************************//**
 * initialize \p field with values of \p A_y,x
 *****************************************************************************/
static __device__ void load(CudaMatrix * A, int n_cols,
                int x, int y, float * field, int tid)
{
        int i;
        for (i = 0; i < ITER; i++){
                int Ax = x * BLOCKW + threadIdx.x;
                int Ay = y * BLOCKW + threadIdx.y + i * BLOCKH;
                int val = maccess(A, Ay, Ax, n_cols);
                field[i * NTHREADS + tid] = val;
        }
}

/*****************************************************************************//**
 * initialize \p field with values of \p A_y,x taking care of access violations
 *****************************************************************************/
static __device__ void load_guarded(CudaMatrix * A, int n_rows, int n_cols,
                int x, int y, float * field, int tid)
{
        int i;
        for (i = 0; i < ITER; i++){
                int Ax = x * BLOCKW + threadIdx.x;
                int Ay = y * BLOCKW + threadIdx.y + i * BLOCKH;
                int val = (Ax < n_cols && Ay < n_rows)
                        ? maccess(A, Ay, Ax, n_cols) : 0;
                field[i * NTHREADS + tid] = val;
        }
}

/*****************************************************************************//**
 * store \p field to \p C_y,x
 *****************************************************************************/
static __device__ void store(CudaMatrix * C, int n_cols,
```

```
                int x, int y, float * field, int tid)
{
        int i;
        for (i = 0; i < ITER; i++){ // write values to global memory
                int Cx = x * BLOCKW + threadIdx.x;
                int Cy = y * BLOCKW + threadIdx.y + i * BLOCKH;
                maccess(C, Cy, Cx, n_cols) = field[i * NTHREADS + tid];
        }
}

/*****************************************************************************//**
 * store \p field to \p C_y,x taking care of access violations
 *****************************************************************************/
static __device__ void store_guarded(CudaMatrix * C, int n_rows, int n_cols,
                int x, int y, float * field, int tid)
{
        int i;
        for (i = 0; i < ITER; i++){ // write values to global memory
                int Cx = x * BLOCKW + threadIdx.x;
                int Cy = y * BLOCKW + threadIdx.y + i * BLOCKH;
                if (Cx < n_cols && Cy < n_rows)
                        maccess(C, Cy, Cx, n_cols) = field[i * NTHREADS + tid];
        }
}

/*****************************************************************************//**
 * multiply and add: temp += rows * cols
 *****************************************************************************/
static __device__ void mult(float * temp, float * rows, float * cols,
                int tid, int tx, int ty)
{
        int i, j;
        for (i = 0; i < ITER; i++){
                float sum = 0.0f;
                int iy = i * BLOCKH + ty;
                int iyB = iy * BLOCKW;
                #pragma unroll // let the compiler unroll this loop
                for (j = 0; j < BLOCKW; j++){
                        // cumulate values
                        sum += rows[iyB + j] * cols[j * BLOCKW  + tx];
                }
                temp[i * NTHREADS + tid] += sum;
        }
}
```

C.2 Gauß-Elimination

Koordination

```
/*****************************************************************************//**
 * Perform gaussian elimination on matrix cA
 *
 * \param cA the matrix to operate on
 * \param height the matrix' height
 * \param width the matrix' width

 * \param[out] swaptable_rows the final permutation of rows
 * \param[out] swaplist_cols the final list of permutations of cols
 * \param[out] rank the determined rank of A
 *****************************************************************************/
int float_gauss_cuda_advanced(
                CUDAMATRIX * cA, int n_rows, int n_cols,

                int ** swaptable_rows, int ** swaplist_cols,
                int * rank, double * ct1, double * ct2, double * ct3,
                double * ct4, double * ct5, double * ct6)
{
        int i, r, rounds, swapsh = 0;
        int done_rows = 0; bool dirty = FALSE;
        int maxrank = MIN(n_rows, n_cols);

        Context ctx;
        ctx.n_rows = n_rows; ctx.n_cols = n_cols; ctx.independent_rows = n_rows;
        ctx.progress = 0; ctx.overallLevel = 0; ctx.swaps = 0;

        cudaMalloc(swaptable_rows, sizeof(int) * n_rows);
        cudaMalloc(swaplist_cols, sizeof(int) * (n_cols * 2 + 1));
        cuda_kernel_init_swaptable <<<1, 1>>> (*swaptable_rows, n_rows);
        cuda_kernel_init_swaplist <<<1, 1>>> (*swaplist_cols, n_cols);
        cudaMalloc(&ctx.swaps, sizeof(int)); // #swaps in cuda-space
        /* init #swaps to 0 */
        cudaMemcpy(ctx.swaps, &swapsh, sizeof(int), cudaMemcpyHostToDevice);

        ctx.swaptable_rows = *swaptable_rows; ctx.swaplist_cols = *swaplist_cols;

        /* the ceilfed number of iterations of size ABS */
        rounds = (int) ceilf(n_rows / ((double) ABS));
        for (r = 0; r < rounds; r++){ // for each iteration
                int start = r * ABS; // start is the first row of the block
                int end = MIN(start + ABS - 1, maxrank - 1);

                /* for each row in this block */
                for (i = start; i <= end; i++){
                        /* make sure this entry is non-zero */
                        if (i <= ctx.independent_rows){
                                bool ok = ensure_row(cA, &ctx, i, done_rows,
                                                ct1, ct2, &dirty);
                                if (!ok) break;
                        }
                        /* update column i to a state as if all preceeding rows
                           had been applied to it already */
                        update_col(cA, &ctx, i, start, i, ct3);
                        /* select the biggest entry of this column */
                        pivotize_cuda_advanced(cA, &ctx,
                                        i, &dirty, ct4);
                        /* compute multipliers for row i */
                        compute_multipliers_for_col(cA, &ctx, i, dirty ?
                                        ctx.progress - 1 : ctx.progress, ct5);
                        /* we're done with this row */
                        done_rows = i;
                }
                ctx.overallLevel += end - start + 1;
                //printf("bounds: %d:%d\n", progress, independent_rows);
                if (r < rounds - 1){
                        /* apply rows _start_:_end_ to _bound+1_:n-1 */
                        TIME(&t1);
                        do_remaining(cA, &ctx, start, end);
                        TIME(&t2);
                        *ct6 += time_diff(&t1, &t2);
                }
        }

        *rank = MIN(ctx.independent_rows, maxrank);
        cudaMemcpy(&swapsh, ctx.swaps, sizeof(int), cudaMemcpyDeviceToHost);

        return swapsh;
}
```