Parallel Computing 2 - Projektarbeit Erweiterung der Matrix-Vektor Implementierung

Anne Schulte-Kroll
Matrikelnummer 192904

Juni 2021

Inhaltsverzeichnis

1 Einleitung

Aufwändige Berechnungen können durch den Einsatz von Parallel Computing in kleinere Probleme zerlegt werden, welche dadurch gleichzeitig berechnet werden können. Grund dafür ist, dass die Parallelisierung es ermöglicht, mehrere Prozesse zeitgleich auszuführen. Laufzeiten von Berechnungen können dadurch erheblich reduziert werden [1].

In der vorliegenden Projektarbeit wird als "Problemstellung" eine Matrix-Vektor Multiplikation betrachtet. Zur parallelen Implementierung dieser existieren verschiedene Möglichkeiten, wie beispielsweise die Grafikkartenprogrammierung mit CUDA [2], welche in diesem Projekt angewendet wird. CUDA bezeichnet dabei eine spezielle Architektur der GPUs, die es nicht nur ermöglicht, 3D-Grafiken zu berechnen, sondern auch beispielsweise für die Matrix-Vektor Multiplikation verwendet werden kann [3].

Im Folgenden wird zunächst das Problem näher erläutert. Anschließend werden die Grundprinzipien der Matrix-Vektor-Multiplikation mit Shared Memory Reduction, atomare Addition der Shared Memory Reduction, alleiniger Verwendung atomarer Operationen und Nutzung der Intra-Grid-Kommunikation zur Synchronisation erklärt. Abschließend folgt eine Evaluation der Ergebnisse.

2 Problemstellung

Ziel des Projekts ist eine Implementierung einer Matrix-Vektor Multiplikation zwischen einer Matrix mit Dimension $R^{i \times j}$ und einem Vektors mit Dimension R^j.

$$\mathbf{A} \cdot \vec{x} = \vec{y}$$

Zur Multiplikation einer Matrix mit einem Vektor werden alle Werte jeder Matrixzeile $\mathbf{A}[i,:]$ mit allen Werten des Vektors $\vec{x}[:]$ komponentenweise multipliziert. Anschließend werden die Produkte aufsummiert. Das Ergebnis ergibt die i-te Komponente $\vec{y}[i]$ des Ergebnisvektors $\vec{y}$. Die Berechnung kann formal wie folgt formuliert werden.

$$\sum_{j=1}^{J} a_{ij} \cdot x_j = y_i$$

Dieses Vorgehen soll unter Verwendung von CUDA durch ein paralleles Programm implementiert werden. Dabei sollen verschiedene Vorgehensweisen in Form von unterschiedlichen Kernels verwendet werden. Die Vorgehensweisen der einzelnen Kernels werden im folgenden Kapitel erläutert.

3 Prinzipien der verschiedenen Kernel

3.1 Kernel 1: Wiederholter Aufruf eines Shared-Memory-Reduktions Kernels

Das Grundprinzip dieses Kernels ist parallele Reduzierung. Jeder Thread berechnet dabei die Summe aus seinem eigenen und einem anderen Element. Das resultierende Element wird anschließend an die nächste Runde weitergegeben. Dieser Vorgang wird so lange wiederholt, bis nur noch ein einziges Element übrig ist. Das resultierende Element stellt das Ergebnis der Operation dar. Das Prinzip wird in Abbildung 1 veranschaulicht.

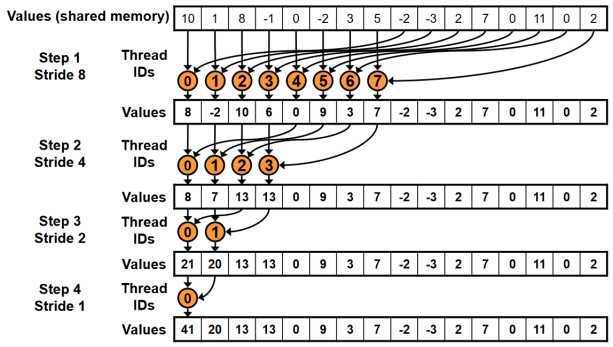


Abbildung 1: Parallele Reduktion [4]

Da bei der Matrix Vektor Multiplikation nicht nur die Summe eines einzelnen Vektors berechnet wird, ist es nötig, den Kernel ln(size) mal aufzurufen. Hierbei wird in der ersten Iteration die Multiplikation der aij · xj berechnet und in einer Buffermatrix gespeichert. In jeder weiteren Iteration wird diese Matrix nun mit dem oben beschrieben Algorithmus reduziert und die Teilelemente werden an den Anfang der Buffermatrix geschrieben. In der letzten Iteration stehen nun für alle Multiplikationen die Endsummen an Stelle 0 der Zwischenmatrix und werden in die entsprechende Stelle des Ergebnisvektors geschrieben.

Mit der CUDA Funktion syncthreads() kann eine "Race Condition" zwischen den einzelnen Threads vermieden werden. Eine "Race Condition" bezeichnet dabei eine Situation beim Multithreading, in welcher zwei oder mehr Threads auf einen gemeinsamen Speicherort zugreifen und versuchen gleichzeitig darauf zuzugreifen. Da der Thread-Scheduling-Algorithmus jederzeit zwischen Threads wechseln kann, ist nicht ersichtlich, in welcher Reihenfolge die Threads versuchen werden auf die gemeinsam genutzten Daten zuzugreifen. Daher hängt das Ergebnis der Berechnung vom Thread-Scheduling ab.

Die Funktion syncthreads() ist eine intrinsische Synchronisierungsfunktion auf Blockebene. Da diese allerdings keine Synchronisation zwischen verschiedenen Blocks garantiert, sondern nur eine Synchronisation auf Blockebene erzeugt, sorgt der wiederholte Aufruf des Kernels des weiteren dafür, dass sich die Threads jedes Blocks synchronisieren.

3.2 Kernel 2.1: Atomic Operations am Ende des Shared-Memory-Blocks

Um eine "Race Conditionßu umgehen, und das mehrmalige Aufrufen eines Kernels zu vermeiden, wird der oben beschriebene Kernel 1 so verändert, dass nach jeder Reduktion ein atomicadd() verwendet wird. Der resultierende Kernel muss dadurch nur noch ein einziges Mal aufgerufen werden. Atomare Operationen haben den Vorteil, dass immer nur ein einziger Thread Zugriff auf einen Speicherplatz hat, während eine Operation, in diesem Fall die Addition, abgeschlossen wird. .

3.3 Kernel 2.2: Atomic Operations für alle notwendigen Operationen

In diesem Kernel sollen alle Berechnungen anhand von atomaren Operationen durchgeführt werden. Dies sind unterbrechungsfreie Lese-, Änderungs , Schreib- und Speicheroperationen, welche von Threads angefordert werden und einen Wert an einer bestimmten Adresse aktualisieren. Ein großer Vorteil dieser Operationen ist in Hinblick auf die "RaceCondition" bei Lese- ,Änderungs-, Schreib- und Speicheroperationen auf geteiltem Speicher ersichtlich.

3.4 Kernel 1.3: Verwendung von Intra-Grid Groups

CUDA bietet seit Version 9.0 kooperativen Gruppen an, die mit dem Grid arbeiten. Unter Verwendung dieser Gruppen kann die Nutzung des Grids auf einer einzelnen GPU beschrieben werden.

In diesem Kernel wird die Funktionalitat grid group verwendet, welche das Grid bei Reduktionsproblemen synchronisieren kann. Die Grid-Level-Synchronisation ermöglicht einen weiteren Kernel-Entwurf, der die blockweisen Reduktionsergebnisse intern synchronisiert, sodass der Host nur einen einzigen Aufruf ausführen muss, um das Reduktionsergebnis zu erhalten.

In kooperativen Gruppen bietet grid group.sync() eine solche Funktionalität, mit Hilfe welcher der Reduktionskernel ohne Iteration auf Kernel-Ebene schreiben kann. Damit alle Thread-Blöcke synchronisiert werden, sollte die Gesamtzahl der aktiven Thread-Blöcke im Grid die Anzahl der maximal aktiven

Blöcke für die Kernel- Funktion und dem Device nicht überschreiten. Die maximale aktive Blockgröße auf einer GPU ist eine Multiplikation der maximalen Anzahl aktiver Blöcke pro Shared Memory und der Anzahl der Streaming-Multiprozessoren.

Da nur eine begrenzte Anzahl an Blöcken möglich ist und die Matrix in den meisten Fällen größer als die maximale Anzahl ist, ist es also nötig die parallele Reduktion in einer For-Schleife im Kernel grid.size() mal auszuführen. Im inneren dieser Schleife werden nun zunächst alle Multiplikationen akkumuliert, um alle Daten mit einer begrenzten Anzahl von Thread-Blöcken abzudecken, woraufhin die parallele Reduktion auf Blockebene durchgeführt wird und deren Ergebnisse anschließend zusammengerechnt und in den Ergebnisvektor geschrieben werden.

4 Evaluation

Zur Evaluation wurde in den Experimenten die Matrixgröße (1024*16) x (1024*16) und Eine Blockgröße von (32,32) gewählt. Im ersten Experiment wurde die Laufzeit der verschiedenen Kernels auf der GPU01 (Nvidia Geforce RTX2070 Super) und der GPU03 (Nvidia Geforce GTX780) der Universität Jena in Abhängigkeit von der Anzahl der Threads verglichen verglichen.

Im Hinblick auf die Laufzeiten der verschiedenen Algorithmen, ist zu erkennen, dass der Shared Memory Kernel (Kernel 1) von allen Kernels am schlechtesten abschneidet. Dies ist vermutlich auf den wiederholten Aufruf dieses Kernels zurückzuführen, welcher notwendig ist, da unterschiedliche Thread-Blöcke auf einem Device nicht innerhalb des Kernels synchronisiert werden können. Durch den wiederholten Aufruf des Kernels wird garantiert, dass alle Threads synchronisiert sind. Das Erzwingen dieser globalen Barriere, hat einen starken negativen Effekt auf die Laufzeit. Dies kann die schlechte Laufzeit des ersten Kernels begründen.

Der Shared Memory Kernel mit AtomicAdd (Kernel 2.1)hingegen schneidet besser ab, als der Kernel, welcher nur Atomic Operations (Kernel 2.2) verwendet und auch als Kernel 1. Der Intra-Grid Kernel schneidet ebenfalls minimal schlechter ab, als Kernel 2.1. Dies ist vermutlich auf die for-Schleife, die sich innerhalb des Kernels befindet zurückzuführen, da diese teilweise Parallelität verhindert. Allgemein kann festgehalten werden, dass der Shared-Memory Kernel mit Atomic-Add am besten abschneidet, da die Serialität in diesem verringert wird.

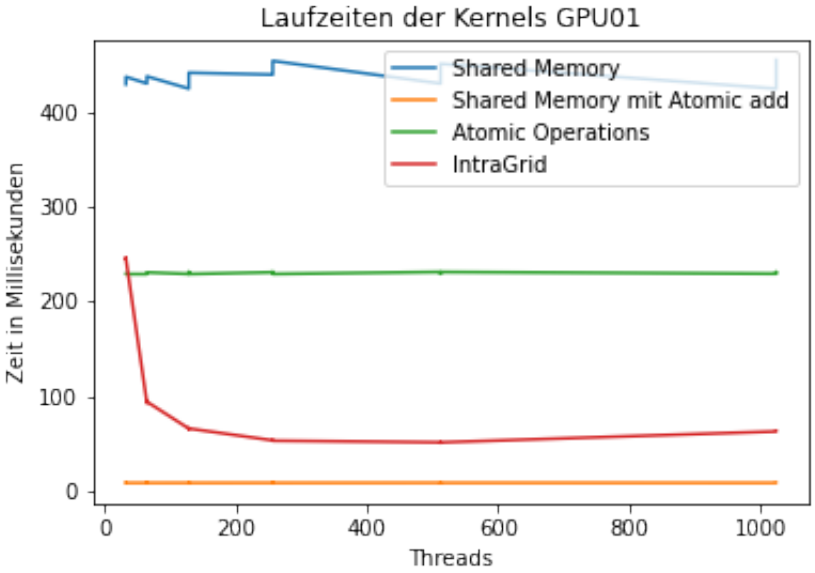


Abbildung 2: Laufzeiten der einzelnen Kernels auf GPU01

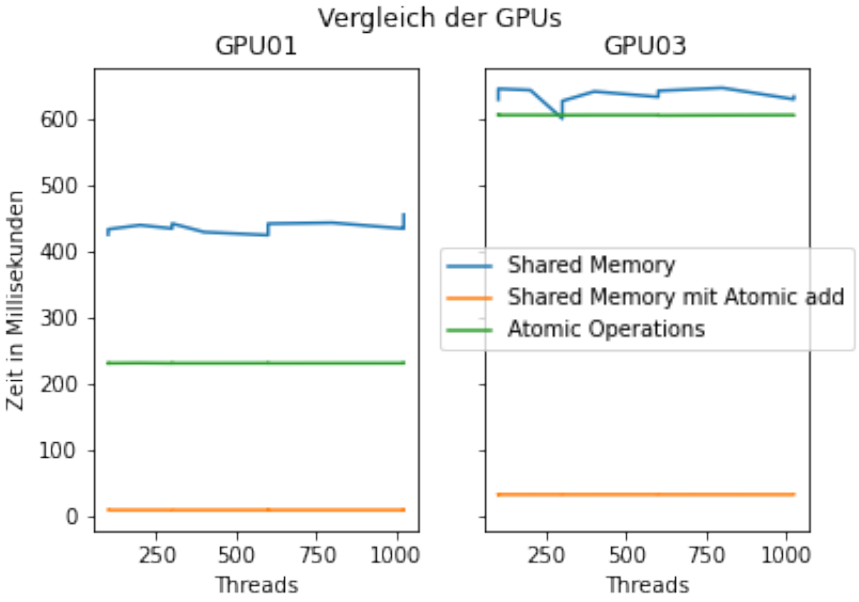


Abbildung 3: Vergleich der Laufzeiten auf GPU01 und GPU03

Darüber hinaus wurde ebenfalls der Einfluss verschiedener Cache Konfigurationen auf den verschiedenen GPUs evaluiert. In CUDA ist es möglich einzustellen wie viel L1-Cache und wie viel Shared Memory von den Kerneln verwendet werden soll. Dabei gibt es folgende Konfigurationsmöglichkeiten:

- cudaFuncCachePreferShared: Bevorzuge 48KB Shared Memory und 16KB L1-Cache
- cudaFuncCachePreferL1: Bevorzuge 48KB L1-Cache und 16KB Shared-Memory
- cudaFuncCachePreferEqual:Bevorzuge 32KB L1-Cache und 32KB Shared-Memory
- cudaFuncCachePreferNone: Keine Präferenz (Standard)

Welche Konfiguration gewählt werden sollte hängt davon ab, wie viel Shared Memory vom Kernel verwendet wird. Daraus folgt, dass mehr Shared-Memory gewählt werden sollte wenn der Kernel viel verteilten Speicher verwendet. Ebenfalls folgt daraus, dass mehr L1-Cache gewählt werden sollte, falls der Kernel mehr Register verwendet.

Die zur Verfügung stehenden Cache-Konfigurationen werden in Abbildungen 4-7 evaluiert. Zu erkennen ist, dass die unterschiedlichen Cache-Konfigurationen keinen deutlich erkennbaren Einfluss auf die Performance haben. Dies kann darauf zurückgeführt werden, dass bei einer Matrix der gewählten Größe nicht viel gemeinsamer Speicher oder L1-Cache verwendet wird und sich somit kein Effekt auf die Laufzeit abzeichnet.

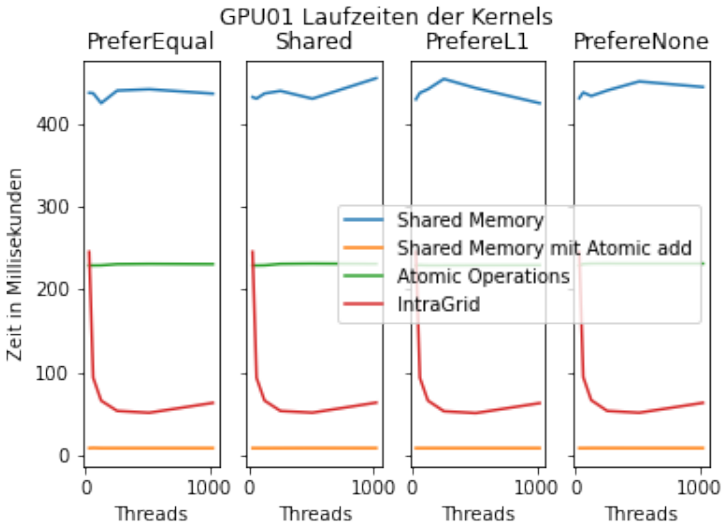


Abbildung 4: Vergleich der Laufzeiten unter verschiedenen Cache Konfigurationen auf GPU01

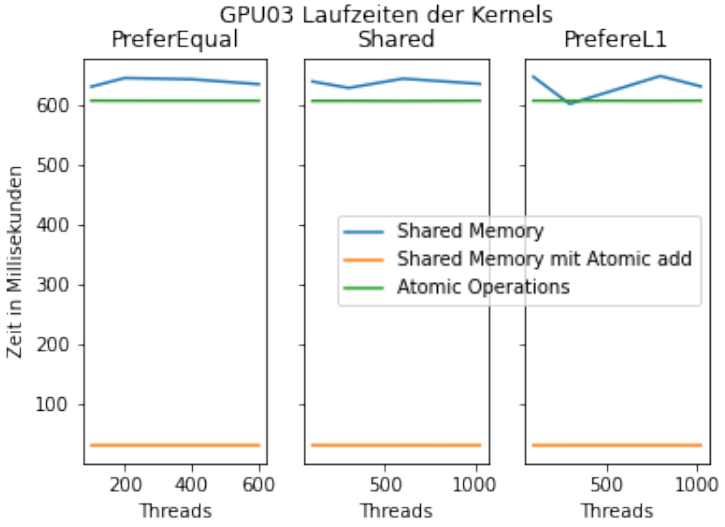
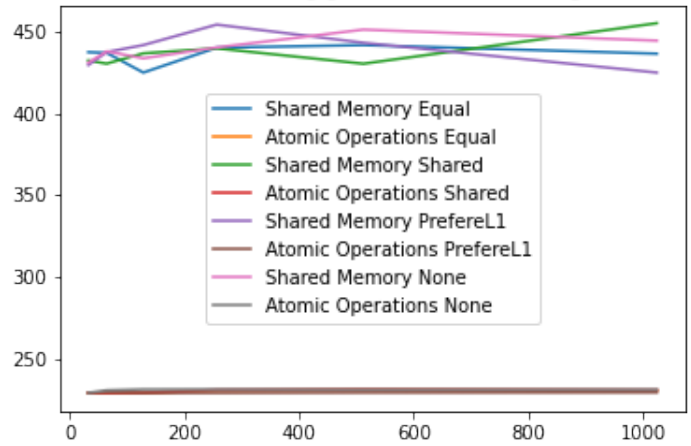
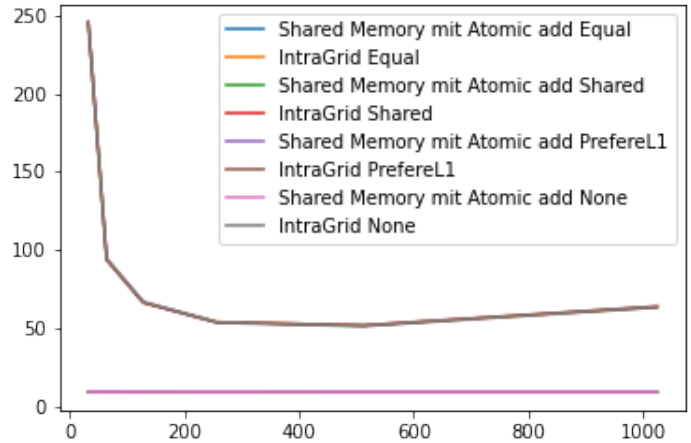


Abbildung 5: Vergleich der Laufzeiten unter verschiedenen Cache Konfigurationen auf GPU03

GPU01 Laufzeiten in Abhängigkeit von Cache Konfiguration (Teil 1)



GPU01 Laufzeiten in Abhängigkeit von Cache Konfiguration (Teil 2)



Literatur

[1] Marc Snir. Universal parallel computing research center at illinois. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–36. IEEE, 2009.

[2] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[3] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[4] Shared memory reduction. https://github.com/mateuszbuda/GPUExample . Accessed: 2021-07-04.

5 Eigenständigkeitserklärung

Ich versichere, dass ich die beiliegende Hausarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Jena den 04.07.2021