UNIVERSITÄT
BAYREUTH

Fakultät für Mathematik, Physik und Informatik
Mathematisches Institut

Implementierung numerischer Algorithmen auf CUDA–Systemen

Diplomarbeit

von

Thomas Jahn

14. Mai 2010
Überarbeitete Version vom 22. Juli 2010

Aufgabenstellung / Betreuung:
Prof. Dr. L. Grüne

Danksagungen

Für die hervorragende Betreuung meiner Diplomarbeit möchte ich mich herzlichst bei Herrn Prof. Grüne bedanken, der mir bei der Anfertigung der vorliegenden Arbeit jederzeit mit Rat und Tat hilfreich zur Seite stand. Der Dank gilt jedoch auch dem gesamten Lehrstuhl, der mir unter der Leitung von Herrn Prof. Grüne leistungsfähige Hardware zur Verfügung gestellt hat, ohne die die Bearbeitung des Themas dieser Arbeit nicht möglich gewesen wäre.

Insbesondere bedanke ich mich bei Dr. Jürgen Pannek, Karl Worthmann und Herrn Prof. Lempio, die mich wie auch Herr Prof. Grüne mit ihrem Engagement immer wieder aufs neue begeistern konnten und mich auf diese Weise während des Großteils meines Studiums begleitet und geprägt haben.

Dafür, dass sie es schafft, dass sich "harte Zeiten", wie etwa die Zeit während der Ausarbeitung dieser Diplomarbeit, nicht hart anfühlen, möchte ich mich sehr bei meiner lieben Partnerin Chrissie bedanken. Es ist schwer vorstellbar, dass man mehr Unterstützung durch seinen Partner bekommen könnte, als ich es habe.

Mein ganz besonderer Dank gilt meinen Eltern. Nicht nur, weil sie mir in meinem Leben sehr viel ermöglicht haben und mir immer ein Vorbild waren; sondern auch, weil sie mir *stets* den richtigen Weg in meinem Leben gezeigt haben und niemals den falschen.

Inhaltsverzeichnis

Einleitung

Die Prozessoren moderner Grafikkarten (GPUs) unterscheiden sich wesentlich von "normalen" CPUs. Die wachsenden grafischen Anforderungen der Computerspieleindustrie haben diese GPUs förmlich zu Rechengiganten wachsen lassen, deren Leistung in Anspruch zu nehmen ausschließlich dem mitgelieferten Grafiktreiber vorbehalten war. Bisher.

CUDA (Compute Unified Device Architecture) bezeichnet eine spezielle Architektur der GPUs bestimmter *n*Vidia Grafikkarten, die die starren Funktionsbegrenzungen auf grafische Berechnungen aufhebt. Statt lediglich beeindruckende dreidimensionale Bilder zu berechnen, können CUDA–fähige Prozessoren nun für eigene Anwendungen verwendet werden.

Die starke Optimierung der Prozessoren auf Berechnungen laden dazu ein, GPUs als mathematische Koprozessoren für eigene komplexe Algorithmen zu verwenden. Dabei drängten sich regelrecht eine Reihe von Fragen auf, wobei "Kann mein Computer das auch?" und "Geht das automatisch, oder muss man dann den Algorithmus anders programmieren?" an erster Stelle stehen dürften.

Die Anwort auf die zweite Frage werde ich bereits vorab geben: *Man muss*. Ein CUDA–System stellt eine Reihe von Anforderungen sowohl an den Programmierer als auch an den Algorithmus selbst. Dies geht so weit, dass bereits im Vorfeld von der Implementierung vieler Algorithmen auf CUDA–Systemen abgesehen werden kann, weil sie auf einem handelsüblichen Prozessor mit seiner Funktionsvielfalt und "Genügsamkeit" wesentlich effektiver umgesetzt werden können. Andererseits wird sich herausstellen, dass – sofern die Anforderungen erfüllt werden können – die spezielle Hardwarearchitektur Möglichkeiten bietet, Algorithmen um ein Vielfaches schneller auszuführen.

Um zu erkennen, was nun die "geeigneten" Algorithmen von den "ungeeigneten" unterscheidet, ist ein relativ detailierter Exkurs in die Hard– und Softwarewelt von CUDA–Systemen erforderlich. Im ersten Teil dieser Arbeit soll genau dieser Aspekt behandelt werden.

Im weiteren Verlauf werden zwei Algorithmen betrachtet, bei denen sich Implementierungen auf einem CUDA–System als vorteilhaft erwiesen haben. Hierbei sollen jedoch nicht die Algorithmen selbst im Vordergrund stehen, sondern vielmehr sollen wesentliche Fragen zu deren Umsetzung und Anwendung geklärt werden:

- Was unterscheidet die Programmierung der Grafikkarte von der "normalen" Programmierung einer herkömmlichen CPU?
- Wie kann die Hardware so effektiv wie möglich angesprochen werden?
- Wie wird ein Algorithmus hinsichtlich seiner Eignung für die Hardwarestrukturen eines CUDA–Systems analysiert?

- Wie könnte die praktische Umsetzung eines Algorithmus auf einem CUDA–System aussehen?
- Wie unterscheidet sich die Laufzeit dieser Umsetzungen von der Laufzeit einer äquivalenten Umsetzung auf der CPU?

Das Ziel dieser Arbeit ist, dem Leser einen schnellen Einstieg in die Entwicklung auf CUDA–Systemen zu ermöglichen und von Beginn an einen möglichst vielseitigen "Werkzeugkasten" für die Implementierung, bestehend aus Strategien zur Algorithmenanalyse, Konzepten und exemplarischen Umsetzungen, mit auf den Weg zu geben. Wie jedoch bereits oben erwähnt, ist dafür zuerst eine gründliche Betrachtung der relevanten Hard– und Software erforderlich.

Die sehr schnelle Weiterentwicklung dieser Hard– und Software lässt die Suche nach gedruckter Literatur über CUDA–Programierung als weitgehend sinnlos erscheinen. Ein Programmierer ist auf die regelmäßig überholten, englischsprachigen Handbücher [8–10] im PDF–Format angewiesen, die *n*Vidia auf der Homepage zum Download anbietet. In dieser Arbeit werden ausschließlich die englischen Fachbegriffe aus dieser Literatur verwendet, um eine problemlose Identifikation dieser Begriffe in der Literatur zu ermöglichen. Eine Zusammenfassung dieser Fachbegriffe findet sich im Glossar im Anhang dieser Arbeit.

Teil I

Programmieren mit CUDA

Kapitel 1

Technische Vorarbeit

Bevor die Innereien des CUDA-Systems genauer unter die Lupe genommen werden, wird natürlich erst ein solches benötigt. Das muss nicht (immer) bedeuten, dass erst viel Geld für neue Hardware ausgegeben werden muss. Woran erkannt werden kann, ob bereits ein CUDA–fähiges System vorhanden ist, wie das Betriebssystem vorbereitet wird und wie Hard- und Software zusammenspielt, soll in diesem Kapitel erläutert werden.

1.1 Anforderungen an die Hardware

Bei einem CUDA–System spielt natürlich die Grafikkarte die entscheidende Rolle. Neben der Rechengeschwindigkeit kann sie sogar über die Tatsache entscheiden, ob ein Algorithmus überhaupt auf dieser Karte ausgeführt werden kann oder nicht. Der Grund dafür ist, dass sich die Fähigkeiten der GPUs (sog. *Compute Capabilities*) stark unterscheiden. Auf diese Fähigkeiten kann erst später eingegangen werden, nachdem die Architektur näher betrachtet wurde (siehe Kapitel 2.6 ab Seite 18).

1.1.1 Unterstützte Grafikkarten

Bei einer *n*Vidia Grafikkarte mit einem Fertigungsdatum ab dem Jahr 2007, ist die Wahrscheinlichkeit groß, dass sie bereits CUDA–fähig ist. Dies betrifft insbesondere *GeForce*–Karten ab der 8er–Serie und professionellere *Quadro*–Karten ab Quadro FX 370M. Zudem werden seit einiger Zeit "Grafikkarten" mit dem Namen *Tesla* produziert, die lediglich für den Einsatz als CUDA–Koprozessor konzipiert sind und keinen Monitoranschluss mehr besitzen.

Eine kurze Liste (Tabelle 1.1) soll einen kleinen Überblick über die Grafikkarten geben, die mit dem CUDA–Treiber programmiert werden können. Da sie aus Platzgründen nicht vollständig ist und sicherlich ein paar Wochen nach Veröffentlichung dieser Arbeit wieder veraltet sein dürfte, empfiehlt es sich, einen Blick auf die offizielle Liste des Herstellers zu werfen[1]. Dort lässt sich schnell feststellen, ob die eigene *n*Vidia Karte CUDA–fähig ist.

[1] http://www.nvidia.de/object/cuda_learn_products_de.html (24.04.2010).

Karte	Kerne	*Capabilities*
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 9800 GT, 8800 GT	14	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	8	1.1
GeForce GT 120, 9500 GT, 8600M GS	4	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GS	2	1.1
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
Quadro Plex 2100 D4	4x14	1.1
Quadro Plex 2100 Model S4	4x16	1.0
Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5800	30	1.3
Quadro FX 4800	24	1.3
Quadro FX 4700 X2	2x14	1.1
Quadro FX 5600	16	1.0
Quadro FX 3700	14	1.1
Quadro FX 4600	12	1.0
Quadro FX 1700, FX 570, NVS 320M	4	1.1
Quadro FX 370, NVS 140M	2	1.1
Quadro FX 370M, NVS 130M, NVS 295	1	1.1

Tabelle 1.1: Liste CUDA–fähiger Grafikkarten (unvollständig).

1.1.2 In dieser Arbeit verwendete PC–Systeme

Es standen drei PCs mit CUDA–fähigen Grafikkarten zur Verfügung: ein Laptop (Dell XPS M1330), eine Workstation mit zusätzlicher GeForce Karte (HP Z600) und ein weiter PC, der für Spiele optimiert wurde. Alle PCs unterscheiden sich in der System–Konfiguration und – was noch wichtiger ist – in den *Compute Capabilities* der Grafikkarten. Diese "Vielfalt" ermöglicht später einen noch tieferen Einblick in die Funktionalität der Chips. Die relevanten Daten der drei Systeme werden in Tabelle 1.2 aufgeführt.

1.2 Vorbereiten der Entwicklungsumgebung

Wenn sichergestellt wurde, dass eine CUDA–fähige Grafikkarte installiert ist, kann das Betriebssystem mit der notwendigen Software ausgestattet werden. Da alle verwendeten Test-

	Dell XPS M1330	HP Z600	Gamer–PC
PC–Daten			
Prozessor	Intel Core2Duo T7250	Intel Xeon E5504	Intel Core2Duo E6850
Kerne	2	2x4	2
CPU–Takt	2.0 GHz	2.0 GHz	3.0 GHz
Speichertakt	667 MHz	1333 MHz	800 MHz
Betriebssystem			
Name	openSUSE 11.0	openSUSE 11.1	openSUSE 11.2
Architektur	64 Bit	64 Bit	64 Bit
Kernelversion	2.6.25	2.6.27	2.6.31
Grafikkarte 1			
Name	GeForce 8400M GS	Quadro NVS 295	GeForce 8800 GTS
Multiprozessoren	2	1	12
GPU–Takt	800 MHz	1300 MHz	1188 MHz
Compute Capabilities	1.1	1.1	1.0
Speicher	128 MB	256 MB	640 MB
Speichertakt	600 MHz	695 MHz	800 MHz
Speicherbandbreite	9.6 GB/s	11.2 GB/s	64 GB/s
Grafikkarte 2			
Name	-	GeForce GTX 285	GeForce 8800 GTS
Multiprozessoren	-	30	12
GPU–Takt	-	1476 MHz	1188 MHz
Compute Capabilities	-	1.3	1.0
Speicher	-	1024 MB	640 MB
Speichertakt	-	1242 MHz	800 MHz
Speicherbandbreite	-	159 GB/s	64 GB/s

Tabelle 1.2: In dieser Arbeit verwendete PC–Systeme.

rechner unter openSUSE als Betriebssystem laufen, wird hier die Installation der Software ausschließlich für diese Linuxdistrubution beschreiben. Der Hersteller bietet allerdings auch Software für andere Distributionen, Windows und Mac OS an.

Die wichtigste Software wurde bereits genannt: der Treiber. Er stellt die niedrigste einheitliche Ebene dar, mit der jede CUDA–fähige Grafikkarte angesprochen werden kann. Die direkte Programmierung über den Treiber – so vielseitig und effizient sie auch sein mag – ist jedoch sehr mühsam. Auch aus diesem Grund stellt *n*Vidia ein zusätzliches Toolkit zum Download bereit. Dieses Softwarepaket kapselt die Treibermethoden in einer sog. *CUDA Runtime Library*, nimmt dem Programmierer sehr viel Arbeit ab und sollte schon allein deswegen nicht auf der Festplatte fehlen. Das Toolkit beinhaltet allerdings auch den Compiler *nvcc* für die Übersetzung der GPU–Programme[2], weshalb es für die Entwicklung unentbehrlich ist.

[2]Sog. *Kernels*, Näheres in den Kapiteln 2.2 und 3.1.

Der Vollständigkeit halber soll an dieser Stelle auch erwähnt sein, dass mitterweile weitere Projekte veröffentlicht wurden, die in kapselnder Funktion direkt auf dem Treiber aufsetzen, wie z.B. DirectX–basiertes *DirectCompute*[3], Apples offener, neuer Standard *OpenCL*[4] oder das auf Fortran basierende Projekt *PGI Accelerator*[5]. Diese werden allerdings nicht in die Betrachtung dieser Arbeit mit einbezogen. Abbildung 1.1 illustriert den Zusammenhang zwischen der Hardware und den oben genannten Softwarepaketen.

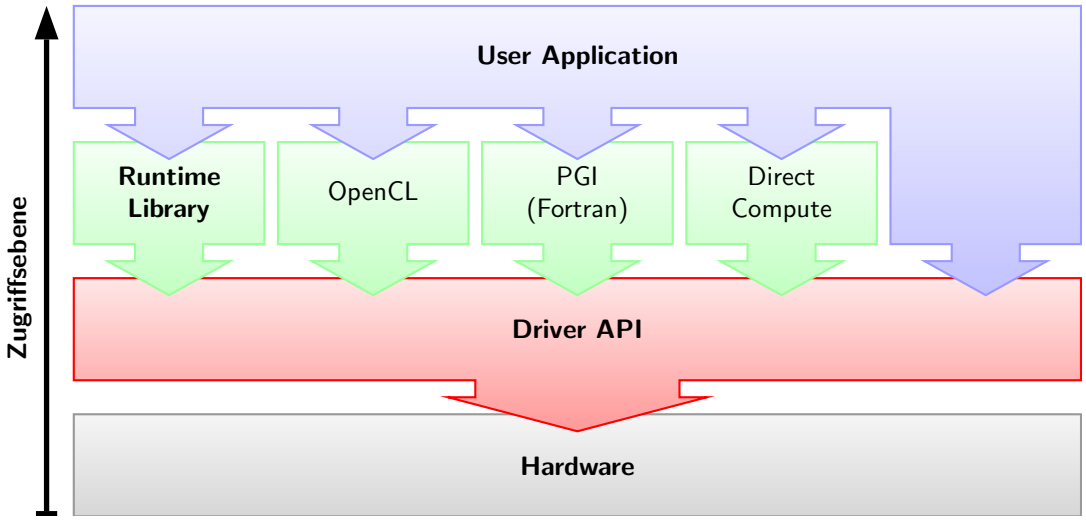


Abbildung 1.1: Zugriffsebenen auf CUDA–Hardware.

1.2.1 Installation des CUDA–Treibers

Bei der Installation auf einem Linux–Betriebssystem wird ein Kernelmodul kompiliert und dem Systemkernel hinzugfügt.

Der standard *n*Vidia –Grafiktreiber, der beispielsweise über Yast aus dem *n*Vidia Repository installiert werden kann, beinhaltet *keine* Unterstützung der CUDA Funktionen. Ein Grund hierfür könnte sein, dass das CUDA–Projekt von *n*Vidia hochaktiv weiterentwickelt wird und sich noch im Beta–Stadium befindet. Sollte also auf dem System noch ein Standardtreiber für die vorhandene Grafikkarte installiert sein, so muss dieser *vor* der Installation des CUDA–Treibers entfernt werden. Aus der Softwareverwaltung zusätzlich das *n*Vidia Repository zu entfernen hat sich als sinnvoll erwiesen, da es nach der Deinstallation des Standardtreibers nicht mehr benötigt wird.

Anschließend muss das grafische System heruntergefahren werden. Dies erreicht man, indem der Runlevel des Betriebssystems in einer Konsole auf die Stufe drei zurückgesetzt wird.

```
> su
> init 3
```

[3] http://www.nvidia.de/object/directcompute_de.html (23.04.2010).
[4] http://www.khronos.org/opencl/ (23.04.2010).
[5] http://www.pgroup.com/resources/accel.htm (23.04.2010).

Die folgende Anmeldung in der Textkonsole des Systems muss als Superuser geschehen. Nachdem die heruntergeladene Installationsdatei des Treibers ausführbar gemacht wurde, kann sie aufgerufen werden.

```
> chmod +x cudadriver_2.3_linux_64_190.18.run
> ./cudadriver_2.3_linux_64_190.18.run
```

Das Installationsskript kompiliert und installiert das Kernelmodul, CUDA– und OpenGL–Bibliotheken und deren C–Headerdateien. Optional wird die X–Server Konfigurationsdatei *xorg.conf* dem neuen Treiber entsprechend modifiziert, was bei einem System mit einer einzigen Grafikkarte auch durchgeführt werden sollte. Das Installationsskript setzt voraus, dass der GCC–Compiler, Kernel–Quellcodes und Kernel–Symbole auf dem System installiert sind.

Wenn eine zweite Grafikkarte installiert ist, die ausschließlich zur CUDA Berechnung benutzt werden soll, ist möglicherweise eine manuelle Änderung der *xorg.conf* notwendig. Die Anzeigegrafikkarte muss in diesem Fall dort explizit per PCI–Bus identifiziert werden, während die zweite Karte in keiner Konfigurationssektion genannt werden darf.

Der relevante Konfigurationsabschnitt auf der HP Z600 Workstation sieht beispielsweise folgendermaßen aus:

```
Section "Device"
    Identifier     "Device[0]"
    Driver         "nvidia"
    VendorName     "NVIDIA"
    BoardName      "Quadro NVS 295"
    BusID          "40:0:0"
EndSection
```

Die Zahl `40` im Attribut `BusID` entspricht der hexadezimalen Busnummer `0x28` der gewünschten Anzeigegrafikkarte, die man mit dem folgenden Konsolenbefehl als Superuser ermitteln kann (überflüssige Zeilen wurden aus der Ausgabe entfernt, es werden beide Grafikkarten angezeigt):

```
> hwinfo --gfxcard
57: PCI f00.0: 0300 VGA compatible controller (VGA)
58: PCI 2800.0: 0300 VGA compatible controller (VGA)
```

Nach der Installation kann das Grafiksystem durch einen Systemneustart oder Umschaltung auf Runlevel 5 neu gestartet werden. CUDA–Treiber, deren API–Bibliotheken und C–Headerdateien sind nun installiert und einsatzbereit.

1.2.2 Installation des CUDA–Toolkits

Die Installation des Toolkits ist vergleichsweise einfach. Nach dem Download des Installationsskripts von der *n*Vidia Webseite wird es, wie bereits bei der Treiberinstallation beschrieben, ausführbar gemacht und als Superuser aufgerufen. Nach der Bestätigung des standardmäßig gesetzten Installationsverzeichnisses werden Runtime Library, Header und Compiler unter /usr/local/cuda installiert.

Der Bequemlichkeit halber empfielt es sich nun noch einige Pfade in den Umgebungsvariablen zu setzen, damit auf die installierten Dateien systemweit bequem zugegriffen werden kann. Es hat sich als sehr praktisch erwiesen, die Variablen als Superuser in der Datei /etc/profile.local zu setzen, indem dort die folgenden Zeilen hinzugefügt werden (vier Zeilen, bei 32–Bit Betriebssystemen müssen die entsprechend äquivalenten `lib` Verzeichnisse angegeben werden):

```
export PATH=/usr/local/cuda/bin:$PATH
export LIBRARY_PATH=/usr/local/cuda/lib64:$LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
export CPLUS_INCLUDE_PATH=/usr/local/cuda/include:$CPLUS_INCLUDE_PATH
```

Der Computer verfügt anschließend über eine funktionierende Entwicklungsumgebung.

Kapitel 2

Die kleine Welt des *n*Vidia Grafikchips

Nach Abschluss von Kapitel 1 steht der Programmierung der GPU – zumindest in technischem Sinne – nichts mehr im Weg. Um jedoch effiziente Programme für CUDA–Hardware schreiben zu können, sind detailierte Kenntnisse über die Funktionsweise dieser Hardware erforderlich.

2.1 Was unterscheidet die GPU von der CPU?

GPUs sind von Grund auf für Berechnungen optimiert. Ein sehr beschränkter Befehlssatz und minimalistischer Cache lassen auf dem Chip reichlich Platz für Recheneinheiten, sog. ALUs ("arithmetic logic unit"). Dieser Aufbau ermöglicht der GPU viele Rechenoperationen parallel durchzuführen. Abbildung 2.1 zeigt einen schematischen Vergleich des Layouts von CPU mit GPU.

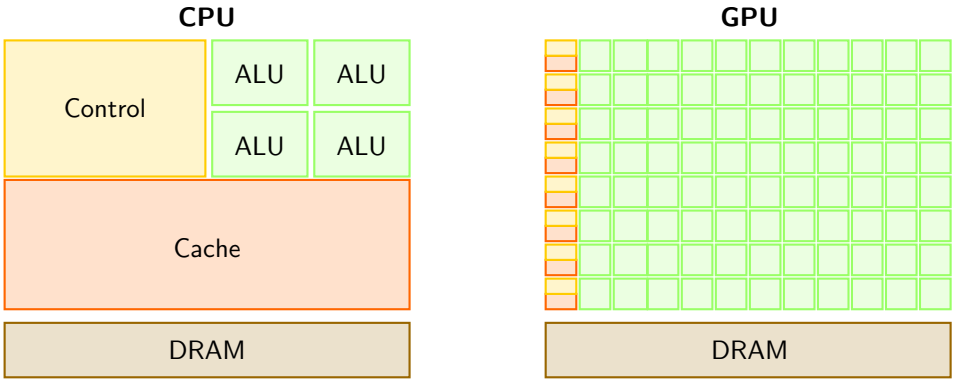


Abbildung 2.1: Schematischer Aufbau von CPU und GPU.

Diese enorme Parallelisierung der Rechenoperationen ist der wesentliche Grund für die hohe Rechenleistung der CUDA-Systeme und muss bei der Konzeption der Algorithmen einbezogen werden.

2.2 Device Kernel: Das GPU-Programm

Der *Device Kernel* ist ein Programm in binärem Code, das direkt auf der GPU ausgeführt wird. Es wird mit Hilfe des Programms *nvcc* aus dem CUDA-Toolkit kompiliert und mit Methoden der CUDA–Treiber API auf der GPU ausgeführt. Zumindest in dieser Hinsicht entspricht ein Device Kernel dem Prinzip eines normalen CPU-Programms, denn das wird letztendlich, zwar nicht durch einen Treiber, jedoch durch das Betriebssystem auf der CPU ausgeführt.

2.2.1 Ablauf einer Kernelausführung

Der Befehl zur Kernelausführung wird von einem Hauptprogramm auf der CPU gegeben. Bezüglich der Kernellaufzeit muss die Grafikkarte als ein abgeriegeltes, unabhängiges Gebiet angesehen werden. Warum das so ist, wird durch die folgende Beschreibung deutlich, die den regulären Ablauf einer Kernelausführung darstellt:

1. Alle Daten, die ein Kernel zum Ausführen benötigt, müssen (mit Hilfe der Treiber API) auf den Grafikspeicher kopiert werden.

2. Durch einen Befehl im Hauptprogramm startet der Treiber anschließend den Kernel auf der Grafikkarte. Das Prinzip einer jeden Kernelausführung ist immer das Folgende (obwohl die einzelnen Punkte in der Praxis oft nicht klar trennbar sind):

   2.1 Der Kernel liest die Daten vom Grafikspeicher

   2.2 Die Daten werden verarbeitet.

   2.3 Nach der Berechnung legt der Kernel die Ausgabedaten im Grafikspeicher ab.

   Bis der Kernel terminiert, hat der Programmierer nun keinerlei Handhabe über das GPU-Programm. Es kann weder abgebrochen werden (etwa durch Strg+C), noch existiert I/O-Peripherie, mit deren Hilfe während der Laufzeit mit dem Kernel kommuniziert werden kann. Der Programmierer hat keine andere Wahl, als auf die fehlerfreie Ausführung des Kernels zu vertrauen.

3. Das Hauptprogramm liest die Daten vom Grafikspeicher und verwendet sie weiter.

Seit der Treiberversion 2.2 hat *n*Vidia ein Prinzip namens *Zero–Copy* in Verbindung mit *Pinned Memory* eingeführt. Das Prinzip sieht vor, CPU–Hauptspeicher von der Treiber API reservieren und markieren zu lassen (Pinned Memory), so dass die GPU direkt auf diesen Speicher zugreifen kann. Motiviert wurde dies durch den Wunsch der Entwickler, Datenströme wie etwa Videosignale schneller auf der GPU verarbeiten zu können, indem Kopieraktionen zwischen CPU– und GPU–Speicher vermieden werden (Zero–Copy). Die Algorithmen, die in dieser Arbeit implementiert wurden, arbeiten nach dem oben dargestellten regulären Prinzip. Der Vollständigkeit halber sollte jedoch diese Möglichkeit zumindest erwähnt werden.

2.2.2 Vorzeitiger Kernelabbruch (Execution-Timeout)

Das völlige Fehlen einer manuellen Kernelabbruchmöglichkeit kann u.a. die sehr unangenehme Folge haben, dass Endlosschleifen ihren Namen alle Ehre machen.

Da die Anzeigegrafikkarte, während ein Kernel darauf ausgeführt wird, blockiert ist, kann das X11–System in dieser Zeit nicht auf Benutzereingaben reagieren bzw. den Bildschirm aktualisieren. Um nun im Falle einer Endlosschleife im Device Kernel keinen Hardware–Reset vornehmen zu müssen, hat der *n*Vidia -Treiber einen Zwangs–Timeout für die Kernelausführung vorgesehen. Jeder Kernel, der auf der Anzeigegrafikkarte ausgeführt wird, wird konsequent nach ca. 5-7 Sekunden (je nach Treiber und Betriebssystem) vorzeitig abgebrochen.

Der (gewaltige) Nachteil ist allerdings, dass dieser Mechanismus eine *beabsichtigte* längere Berechnung ebenfalls verhindert. Sollte man den Kernel–Timeout also doch umgehen wollen, ist der Betrieb einer weiteren Grafikkarte (die nicht zur Anzeige verwendet wird) die einzige Möglichkeit. Auf einer zweiten Grafikkarte werden Kernels immer mit unbegrenzter Laufzeit ausgeführt. In diesem Fall muss eine Endlosschleife jedoch mit dem Herunterfahren des grafischen Systems (Beenden des *n*Vidia – Treibers) abgebrochen werden. In Kapitel 1.2.1 wird die Systemkonfiguration für zwei Grafikkarten unter Linux beschrieben.

2.3 SIMD als essentielles Prinzip

Es wurde bereits in Kapitel 2.1 erwähnt, dass der entscheidende Vorteil der GPUs deren sehr hohe Anzahl von ALUs ist, mit denen viele Rechenoperationen gleichzeitig ausgeführt werden können. Die schematische Abbildung 2.1 auf Seite 11 zeigt allerdings auch deutlich, dass weit weniger Transistoren für Ansteuerung und Flusskontrolle der ALUs verbaut werden. Wie können nun diese vielen ALUs, deren parallele Nutzung eine umfangreiche Kontrolle verschiedener Programmthreads erfordert, mit einer derart reduzierten Kontrolleinheit effizient genutzt werden?

Die Lösung dieses Problems ist eine Prozessorarchitektur namens *Single Instruction Multiple Data*, oder kurz SIMD. Das Prinzip dieser Architektur sieht vor, dass alle Recheneinheiten des Prozessors synchron laufen und exakt die gleichen Befehle ausführen. Dank dieser Synchronisierung ist nur noch eine Steuereinheit für mehrere Threads notwendig. Vergleichen lässt sich dieses Prinzip mit einer Tischfußballstange, bei der fünf Spielfiguren mit einer einzigen Hand kontrolliert werden können.

Die Komplexität des SIMD-Programms entsteht durch Variation der dem jeweiligen Thread zu Grunde liegenden Daten. Es wendet also jeder Thread die gleichen Instruktionen auf Daten an, deren Adresse von der Nummer des Threads abhängig ist. Abbildung 2.2 illustriert den typischen Ablauf eines einfachen SIMD-Programms (Zustand nach drei Schritten), bei dem zwei parallele Threads benachbarte Daten (blau) addieren und das Ergebnis (grün) an anderer Stelle im Speicher ablegen. Auf CUDA–Systemen ist das SIMD–Programm gleich dem Device Kernel.

Der Preis, der dafür zu zahlen ist, ist Flexibilität. Es liegt auf der Hand, dass ein solch hohes Maß an Synchronisation keine komplizierten Abläufe erlaubt, sondern gewisse strukturelle Anforderungen an die Programme stellt, die auf der GPU ausgeführt werden sollen. Die Tragweite der Einschränkungen wird besonders beim Versuch deutlich, mit der oben

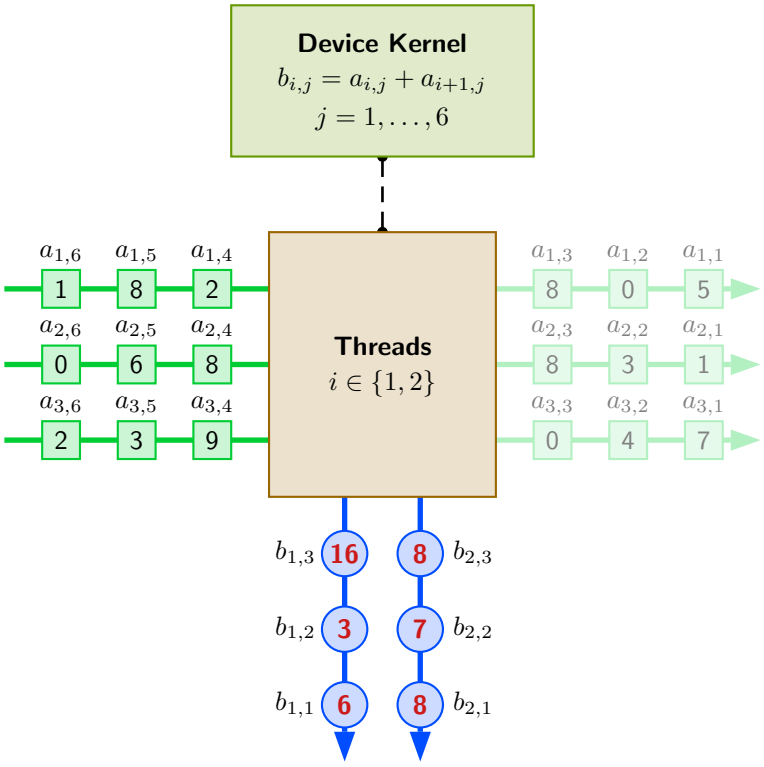


Abbildung 2.2: Ablauf eines SIMD-Programms mit zwei Threads.

erwähnten Tischfußballstange beispielsweise fünf Bälle gleichzeitig zu spielen.

2.4 Threads, Blöcke und Warps – So arbeitet die GPU

Eine GPU hat, ähnlich wie ein Dual Core–Prozessser von Intel, mehrere unabhängige Kerne. In jedem Kern arbeiten (und das ist ein wesentlicher Unterschied zur CPU) acht absolut synchron laufende Prozessoren, weshalb *n*Vidia den Kern einer GPU als *Multiprozessor* bezeichnet.

2.4.1 Nummerierung der Threads

Das Beispiel von Abbildung 2.2 macht deutlich, dass die GPU als SIMD–Prozessor immer eine gewisse Indizierung zur Datenaddressierung benötigt. *n*Vidias CUDA-Prozessor unterstützt hierfür ein dreidimensionales[1] Indexgitter mit der maximalen Ausbreitung $512 \times 512 \times 64$, wobei das Gitter höchstens 512 Elemente – also Threads – besitzen darf. Diese 512 Threads werden *Threadblock* oder *Block* genannt.

Um größer skalierte Probleme verarbeiten zu können, werden auf der GPU mehrere dieser Threadblöcke in einem weiteren, hierarchisch höheren, Indexgitter angeordnet. Dieses sog. *Blockgitter* (oder *Grid*) ist zweidimensional und höchstens 65535×65535 Elemente groß.

[1] Die Dimension kann auch geringer sein. In diesem Fall ignoriert man einfach Komponenten (Projektion).

Im Gegensatz zum Threadblock besitzt das Blockgitter keine zusätzliche Beschränkung der Elementanzahl. Abbildung 2.3 zeigt ein Beispiel dieser hierarchischen Gitteranordnung, bei der ein Blockgitter der Größe 3×2 und als Größe der Threadblöcke 3×4 gewählt wurde.

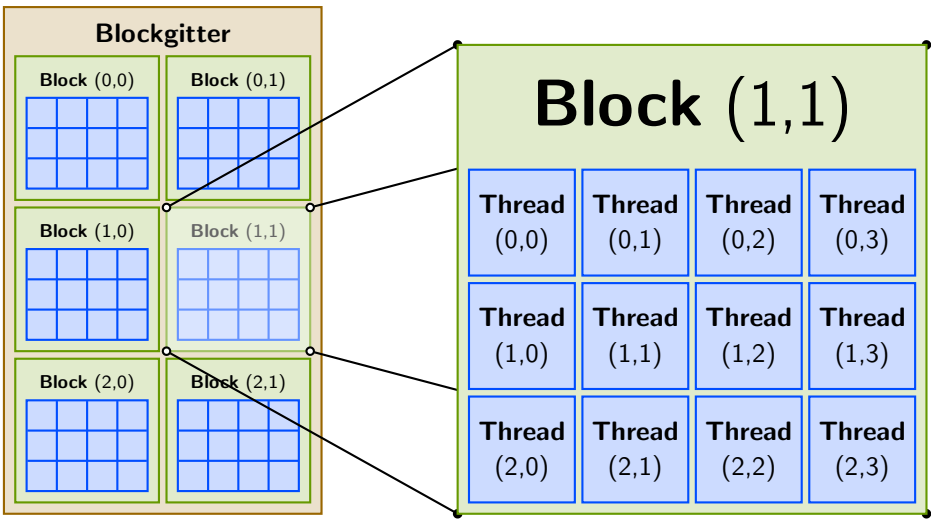


Abbildung 2.3: Beispiel eines Blockgitters mit Threadblöcken auf der GPU.

Bei der Ausführung des Device Kernels werden alle Blöcke in einer im Allgemeinen nicht vorhersehbaren Reihenfolge auf die ggf. verschiedenen Multiprozessoren der GPU aufgeteilt und dort abgearbeitet. Es existiert also eine Art von Pool, aus dem sich alle beteiligten Multiprozessoren zweidimensional numerierte Threadblöcke blind "ziehen".

Dieses flexible Prinzip ermöglicht offensichtlich, dass sich eine nahezu beliebige Anzahl von Multiprozessoren die Kernelausführung teilen. Ob der Kernel auf einer GPU mit einem einzelnen Multiprozessor oder auf einer GPU mit 30 Multiprozessoren (wie etwa in der HP Workstation) ausgeführt wird, macht somit aus der Sicht des Programmierers keinen Unterschied und ist für den Kernel und dessen Aufruf per Treiber API irrelevant.

In der Produktlinie von *n*Vidia tauchen mittlerweile Karten auf, die ihre Multiprozessoranzahl z.B. mit 2×30 benennen (siehe Tabelle 1.1 auf Seite 6). Eine solche Karte wird von Betriebssystem und Treiber wie mehrere Karten mit jeweils eigenem Grafikspeicher behandelt, in diesem Fall wie zwei Karten mit jeweils 30 Multiprozessoren. Der Programmierer müsste hier selbst dafür sorgen, dass die Threadblöcke auf die unterschiedlichen Karten aufgeteilt werden und dementsprechend auch den Grafikspeicher jeder Karte befüllen. Da jede Karte ihr eigenes Blockgitter nach dem oben beschriebenen Prinzip verarbeitet, muss die Berechnung also auf mehrere völlig unabhängige Teile aufgespalten werden.

2.4.2 Ausführung der Threadblöcke

Je nach Größe des Kernels[2] nimmt sich ein Multiprozessor bis zu acht Blöcke aus diesem oben genannten Threadblock–Pool. Für jeden Threadblock startet der Multiprozessor gleichzeitig

[2]Gemeint ist hier die Menge der Prozessorressourcen, die ein Kernel benötigt. In Kapitel 4 wird dieser Aspekt genauer betrachtet.

den Device Kernel. Diese acht ausgewählten Threadblöcke werden als *aktiv* bezeichnet.

Die Threads aller aktiven Blöcke werden zu *Warps* gruppiert. Jeder Warp umfasst dabei höchstens 32 Threads und wird von den acht Prozessoren eines Multiprozessors absolut synchron nach dem SIMD–Prinzip bearbeitet. Die Warps bzw. Threads, die zu einem aktiven Block gehören, heißen ebenfalls *aktiv*. Der Fokus dieser acht Prozessoren wechselt während der Laufzeit immer wieder zwischen sämtlichen aktiven Warps des Multiprozessors, wenn es möglich oder erforderlich ist. Auf diese Art und Weise werden beispielsweise Wartezeiten (u.a. auf Grund von Speicherzugriffen) effizient überbrückt oder alle Threads innerhalb eines Blocks synchronisiert (siehe *Threadsynchronisation* in Kapitel 3.2.4 auf Seite 27).

Der genaue Ablauf soll exemplarisch in Abbildung 2.4 verdeutlicht werden. Der Multiprozessor verarbeitet in diesem Beispiel einen Kernel mit sieben Instruktionen, wobei hier ein Threadblock 64 Threads umfasst und zwei Blöcke auf dem Multiprozessor parallel ausgeführt werden. Nach der vierten Instruktion müssen die Threads auf Grund eines Speicherzugriffs einen Moment warten. Die Größe des Blockgitters ist hierbei nicht relevant. Man beachte jedoch, dass die Nummerierung der gleichzeitig bearbeiteten Blöcke im Allgemeinen nicht fortlaufend ist, die Nummerierung der Threads jedoch schon.

2.5 Der Speicher auf der Grafikkarte

Nun ist bisher häufig vom Grafikspeicher die Rede gewesen. Das Ablaufschema in Kapitel 2.2.1 zeigt deutlich, dass der Grafikspeicher der Dreh– und Angelpunkt einer jeden Kernelausführung ist, denn offenbar findet über ihn sämtliche Kommunikation des aufrufenden CPU–Programms mit den GPU–Berechnungen statt. Hinzu kommt die Tatsache, dass viele Algorithmen einen Zwischenspeicher benötigen, der ebenfalls auf der Karte reserviert werden muss. Dies sind Gründe genug, die Speicherorganisation auf der CUDA–Grafikkarte genauer zu betrachten[3].

2.5.1 Device Memory

Device Memory ist die populäre Speicherart auf der Grafikkarte. Dieser Speicher variiert in seiner Größe von Karte zu Karte und wird vom Hersteller (wie auch im bisherigen Verlauf dieser Arbeit) als "Grafikspeicher" angegeben. Als Beispiel für die Größe des Device Memory sei 1 GB genannt, wie es bei der GTX 285 der HP Workstation der Fall ist.

Mit Hilfe der Treiber API können Daten vom CPU-Hauptspeicher in das Device Memory kopiert werden, wo sie verweilen, bis das aufrufende CPU–Programm beendet wird. Die Daten stehen allen Kernelausführungen zur Verfügung, bis sie explizit wieder freigegeben werden.

Technisch betrachtet erfordert diese globale Verfügbarkeit, dass die Speicherbänke außerhalb des Prozessors auf der Grafikkarte untergebracht sind. In der Regel handelt es sich hierbei um schnellen DDR3-Speicher, wie er auch beim Hauptspeicher der CPU verwendet wird. Diese Auslagerung hat ihren Preis: ein Zugriff auf diesen Speicher benötigt je Geschwindigkeit des Datenbusses und der GPU ca. 400-600 GPU–Taktzyklen.

[3]Vorgestellt werden nur die Speichertypen, die für diese Arbeit relevant sind. Eine CUDA–Grafikkarte verfügt z.B. auch über ein spezielles Texturspeicher–Management, auf das hier aus Platzgründen nicht eingegangen wird.

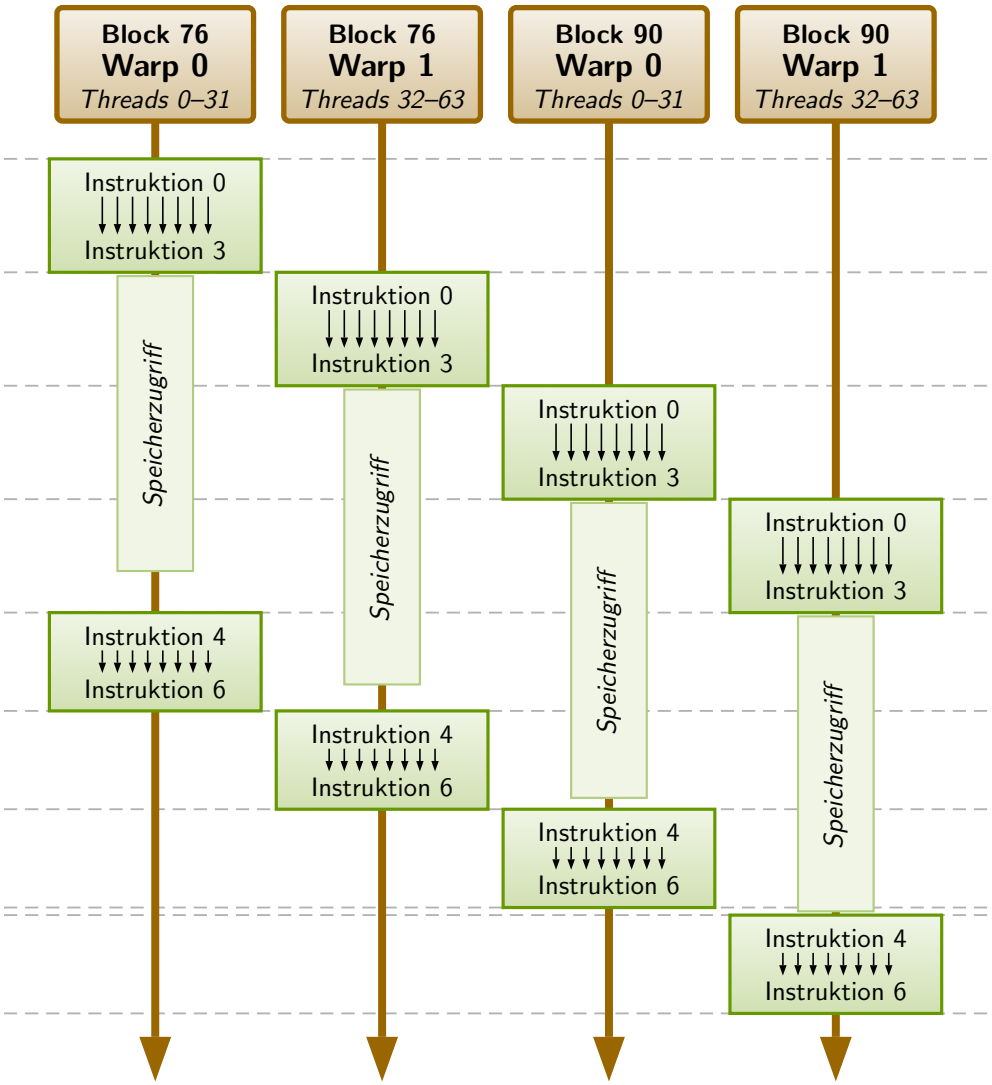


Abbildung 2.4: Exemplarische Ausführung von zwei aktiven Threadblöcken auf einem Multiprozessor.

2.5.2 Shared Memory

Als *Shared Memory* wird ein kleiner Speicherbereich auf einem Multiprozessor bezeichnet. Die Größe ist i.d.R. auf 16 KB beschränkt.

Der Zugriff auf diesen Speicher erfolgt ausschließlich durch den Device Kernel. Daten, die in diesem Speicher abgelegt werden, bleiben nur für die Laufzeit eines einzigen Threadblocks gültig, können in dieser Zeit jedoch von allen Threads eines Blocks bearbeitet werden. Der vorgesehene Verwendungszweck dieses Speichers ist die Zwischenspeicherung von Rechenergebnissen während der Laufzeit eines Threadblocks.

Da dieser Speicher direkt in einem Multiprozessor integriert ist, erfolgt der Zugriff darauf in lediglich vier Taktzyklen, was zeigt, dass dieser Speicher dem Device Memory wenn möglich immer vorzuziehen ist.

2.5.3 Register

In den *Registern* eines Multiprozessors werden alle lokalen Variablen abgelegt, die ein einzelner Thread reserviert. Der Programmierer hat keinen direkten Zugriff auf die eigentlichen Register, da der Compiler die Variablen automatisch verteilt. Daten auf den Registern haben nur innerhalb eines einzigen Threads Gültigkeit.

Der Zugriff auf Register erfolgt in einem Taktzyklus und stellt somit die schnellste Speicherart dar. Die Anzahl der Register ist stark begrenzt, worauf in den Kapiteln 2.6 und 4 weiter eingegangen wird.

2.5.4 Constant Memory

Das *Constant Memory* ist ein 64 KB kleiner Bereich im Device Memory, auf den ein Kernel nur lesend zugreifen kann. Da dieser Speicher in den Speicherbänken des Device Memory liegt, sind die Daten während der gesamten Laufzeit des CPU–Programms gültig und ein Zugriff darauf erfolgt ebenfalls relativ langsam in 400-600 Taktzyklen.

Im Gegensatz zum normalen Zugriff auf Device Memory können die Daten pro Multiprozessor zwischengespeichert werden. Der Zugriff auf konstante Daten, die sich bereits im Cache befinden, ist so schnell wie ein Registerzugriff. Allerdings ist dieser Cache auf 8 KB pro Multiprozessor begrenzt. Die Daten im Constant Memory werden mit Methoden der Treiber–API durch das Hauptprogramm zur Verfügung gestellt.

2.5.5 Local Memory

Sollten die Register für einen Thread nicht ausreichen, wird ein Teil der Variablen im *Local Memory* abgelegt. Der Compiler reserviert hierfür soviel Platz im Device Memory, wie er benötigt und speichert dort die Variablen. Wie bei den Registern hat der Programmierer auch hier keine direkte Handhabe über die eigentliche Speicherreservierung – der Compiler erledigt dies automatisch. Lokale Arrays eines Threads, deren Zugriffe nicht zum Zeitpunkt der Kompilierung bekannt sind, werden ebenfalls im Local Memory abgelegt.

Bezüglich der Geschwindigkeit ist der Begriff “local” irreführend. Der Zugriff auf eine Variable im Local Memory ist ebenso langsam wie auf Device Memory. Der Name beschreibt lediglich die Gültigkeit der Daten, die auf die Ausführung eines einzelnen Threads beschränkt ist.

2.6 Fähigkeiten der Grafikprozessoren

Die sogenannten *Compute Capabilities* beschreiben Fähigkeiten und Kenngrößen einer CUDA–Grafikkarte. Bei der Konzeption eines Device Kernels spielen sie eine große Rolle. Die Compute Capabilities einiger Grafikkarten sind in Tabelle 1.1 auf Seite 6 angegeben. In den folgenden Abschnitten werden die unterschiedlichen Compute Capabilities vorgestellt. Es werden lediglich Eigenschaften aufgeführt, die für den weiteren Verlauf dieser Arbeit relevant sind. Weitere Angaben können der CUDA–Dokumentation von *n*Vidia entnommen werden.

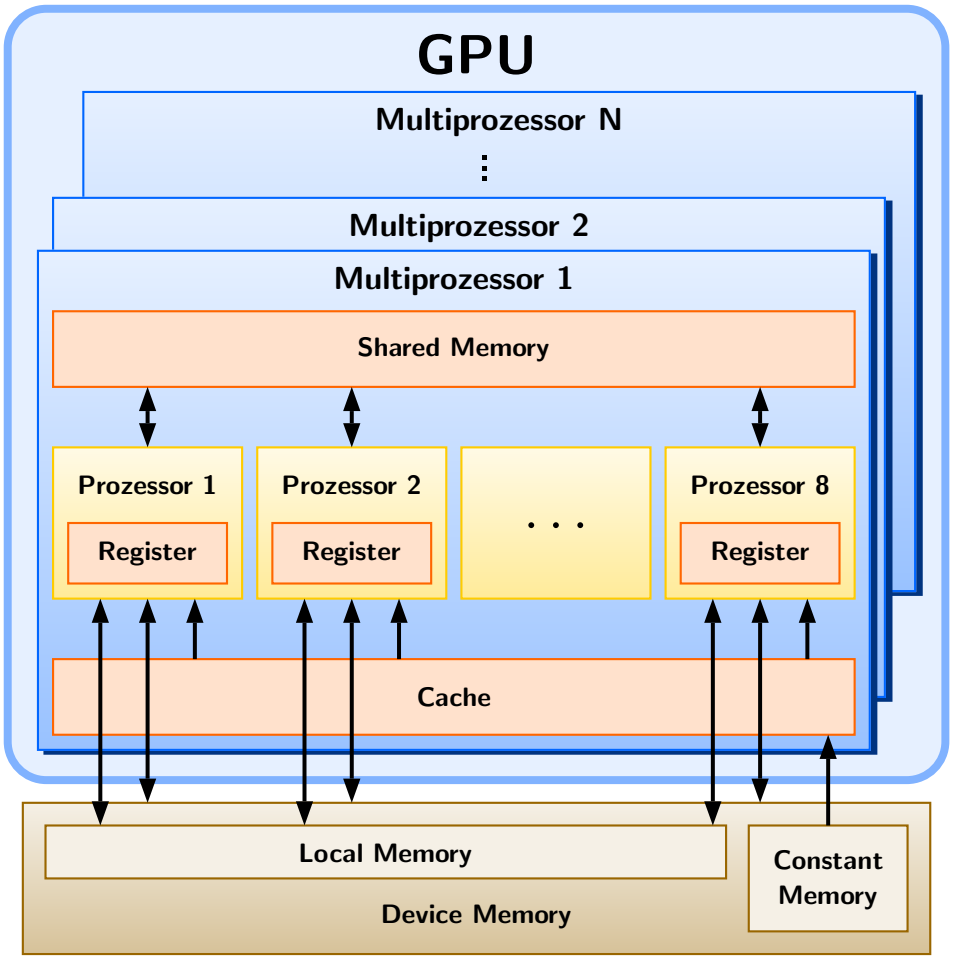


Abbildung 2.5: Speicherbereiche auf der Grafikkarte.

Höhere Compute Capabilities (höhere Versionsnummer) erben die Eigenschaften der niedrigeren und verändern sie gegebenenfalls.

2.6.1 Compute Capabilities 1.0

- Die maximale Größe eines Threadblocks ist $512 \times 512 \times 64$.
- Die maximale Anzahl an Threads pro Block ist zusätzlich auf 512 begrenzt.
- Die maximale Größe des Blockgitters ist 65535×65535.
- Ein Multiprozessor hat 8192 Register.
- Ein Warp hat 32 Threads.
- Die Größe des Shared Memory ist 16 KB pro Multiprozessor.
- Die Größe des Constant Memory ist 64 KB.

- Der Cache für das Constant Memory ist 8 KB pro Multiprozessor.
- Die maximale Anzahl aktiver Blöcke pro Multiprozessor ist 8.
- Die maximale Anzahl aktiver Warps pro Multiprozessor ist 24.
- Die maximale Anzahl aktiver Threads pro Multiprozessor ist 768.
- Die Größe eines Device Kernels ist auf 2 Millionen Instruktionen begrenzt.
- Unterstützt single–precision Gleitkommarechnung.

2.6.2 Compute Capabilities 1.1

- Unterstützt *Atomic Functions*[4], die im Device Memory auf 32–Bit–Worten operieren.

2.6.3 Compute Capabilities 1.2

- Unterstützt Atomic Functions, die im Device Memory und im Shared Memory auf 64–Bit–Worten operieren.
- Ein Multiprozessor hat 16384 Register.
- Die maximale Anzahl aktiver Warps pro Multiprozessor ist 32.
- Die maximale Anzahl aktiver Threads pro Multiprozessor ist 1024.

2.6.4 Compute Capabilities 1.3

- Unterstützt double–precision Gleitkommarechnung.

[4]Siehe Kapitel 3.2.6 ab Seite 28.

Kapitel 3

Die C–Schnittstelle zur GPU

In diesem Kapitel soll gezeigt werden, wie die theoretischen Prinzipien aus Kapitel 2 beim Programmieren umgesetzt werden können. *n*Vidia hat hierfür die etablierte Programmiersprache *C* vorgesehen und deren Syntax und Sprachumfang um Elemente erweitert, die die Abbildung dieser theoretischen Prinzipien in ein reales Programm ermöglichen.

3.1 Das erste Programm: "Hello, World" mal anders

Was würde sich besser für die Einführung in eine fremde Programmiersprache[1] eigen, als ein traditionelles "Hello, World"–Beispiel, das seit 1974 bei unzähligen Programmieranfängern für dankbare Erleichterung sorgt[2]? Auch für die GPU lassen sich an einem derartigen Minimalbeispiel die einfachsten Grundlagen der Kernelprogrammierung demonstrieren: Wie ist der Programmcode aufgebaut? Wie kompiliert man ein Programm? Wie wird es ausgeführt?

Bei einem typischen "Hello, World"–Programm wird lediglich ein kurzer Text auf dem Bildschirm ausgegeben. Zwar ist dies bei einem Device Kernel mangels Peripherie nicht möglich, es kann allerding auch der Grafikspeicher als eine Art von "Standardausgabe" interpretiert werden. Da sich die Fähigkeiten eines Device Kernels auf paralleles Rechnen beschränken, könnte der Ablauf eines "Hello, World"–Programms auf einem CUDA–System also folgendermaßen aussehen.

1. Lege den Text "`Ifmmp-!Xpsme`" auf den Grafikspeicher. Dies entspricht dem Text "`Hello, World`" wobei der ASCII–Wert jedes Zeichens um eins erhöht ist.

2. Der Kernel wird ausgeführt und reduziert den ASCII–Wert jedes Zeichens dieses Texts in zwei Threadblöcken mit jeweils sechs Threads (ein Thread pro Zeichen) um eins.

3. Das Hauptprogramm liest den Speicher aus und gibt das Resultat auf dem Bildschirm aus.

[1] Genau genommen handelt es sich nicht wirklich um eine fremde Sprache. Zusätzliche Schlüsselwörter, ein neues Programmierkonzept und eine andere Art der Programmausführung lassen die Kernelprogrammierung jedoch durchaus fremdartig wirken.

[2] Das "Hello, World"–Beispiel stammt ursprünglich aus dem Buch *The C Programming Language* von Brian Kernighan und Dennis Ritchie aus dem Jahr 1974 und demonstriert in minimalistischer Art und Weise die Verwendung einer Programmiersprache.

Die Ausgabe von "`Hello, World`" auf dem Bildschirm gilt demnach als Hinweis, dass der Kernel wie erwartet ausgeführt wurde.

3.1.1 Aufbau des Quellcodes

In Kapitel 2 wurde gezeigt, dass ein Programm, das Befehle auf der GPU ausführt, immer zwei Teile haben muss: Den Device Kernel und ein CPU-Programm, das mit Treiber–Befehlen den Kernel auf der GPU startet. Für den Programmierer bedeutet dies, dass dem Compiler mitgeteilt werden muss, welche Methode für die CPU (als sog. *Host Code*) und welche für die GPU (als sog. *Device Code*) kompiliert werden muss. Dafür werden Methoden, die als Device Code kompiliert werden sollen, mit einem speziellen Kennzeichner versehen.

Der folgende Programmcode[3] stellt das gesamte "Hello, World"–Programm dar:

```
1  #include <stdio.h>
2
3  // Initialize global memory space
4  __device__ char cutext[13] = "Ifmmp-!Xpsme";
5
6  // define a method called "kernel" as device code
7  __global__ void kernel()
8  {
9      // compute index of corresponding character
10     int index=blockDim.x*blockIdx.x + threadIdx.x;
11
12     // decrease ASCII-code of the character
13     cutext[index] -= 1;
14 }
15
16 int main()
17 {
18     // allocate host memory
19     char text[13];
20
21     // call kernel with two blocks each with six threads
22     kernel<<<2, 6>>>();
23
24     // copy modified text from device memory to host
25     cudaMemcpyFromSymbol(text, cutext, 13, 0, cudaMemcpyDeviceToHost);
26
27     // print modified text
28     printf("%s\n", text);
29     return 0;
30 }
```

Quellcode 3.1: "Hello, World" vom Grafikprozessor.

Nach der `#include`–Direktive für die Bildschirmausgabe wird in Zeile 4 Device Memory reserviert. Das Schlüsselwort `__device__` kennzeichnet diese Variable als dem Device Code angehörend.

Der Device Kernel wird in den Zeilen 7 bis 14 definiert. Auch hier kennzeichnet ein Schlüsselwort (`__global__`) die Methode als Device Code. In Zeile 10 wird für den aktuellen Thread ausgerechnet, welches Zeichen zu dekrementieren ist. Ein Thread greift hier also auf das Zeichen mit dem Index *Threadblockgröße* × *Blockindex* + *Threadindex* zu (eine genauere Beschreibung der zusätzlichen Befehle im Device Code findet im Kapitel 3.2 ab

[3]Die beigelegte CD dieser Arbeit enthält die Implementation dieses "Hello, World"–Beispiels, siehe Seite 141ff.

Seite 24 statt). Bei zwei Blöcken, jeweils mit Blockgröße 6, entspricht dies exakt den Zeichen 0 bis 11 des Texts “`Ifmmp-!Xpsme`”.

Ab Zeile 16 wird die gewöhnliche Hauptmethode für das CPU–Programm als Host Code definiert. Der CPU–Speicher wird dort in altbekannter Manier unter dem Zeiger `char text[13]` reserviert.

Die Ausführung des Device Kernels erfolgt in Zeile 22, indem die als `__global__` markierte Methode `kernel()` aufgerufen wird. Die Gitterkonfiguration für die Kernelausführung wird direkt nach dem Methodennamen durch zwei Parameter umschlossen von dreifachen spitzen Klammern festgelegt[4]. Der erste Parameter steht für die Größe des Blockgitters und der Zweite für die Größe der Threadblöcke. In diesem Fall werden also ein eindimensionales Blockgitter der Größe 2 und eindimensionale Threadblöcke der Größe 6 vorgeschrieben. Nach den spitzen Klammern folgt die Liste der Methodenparameter, wie sie bei der Methodendefinition angegeben wurde (in diesem Fall leer). Intern werden durch den Aufruf der `kernel()`–Methode diverse Methoden der Treiber–API ausgeführt, die den entsprechenden Device Code auf die GPU laden und diesen dort ausführen.

Mehrdimensionale Gitter können angegeben werden, indem man statt skalaren Gitterparametern Variablen des Typs `dim3` innerhalb der spitzen Klammern übergibt. Variablen dieses Typs sind Strukturen mit drei `int`–Komponenten (siehe Kapitel 3.2.2 ab Seite 26). Alternativ könnte der Kernelaufruf in Zeile 26 also auch folgendermaßen aussehen[5]:

```
dim3 gridsize(2, 1, 1);         // define blockgridsize 2x1x1
dim3 blocksize(6, 1, 1);        // define threadblocksize 6x1x1

kernel<<<gridsize, blocksize>>>();
```

Nach der Kernelausführung wird der modifizierte Text mit der Runtime Library Methode `cudaMemcpyFromSymbol()` vom Grafikspeicher in die CPU–Variable `char text[]` kopiert und anschließend ausgegeben.

3.1.2 Kompilieren und ausführen

Das Programm aus Quellcode 3.1 sei unter dem Dateinamen `helloworld.cu` abgespeichert. Für das Übersetzen stellt *n*Vidia einen Compiler namens *nvcc* zur Verfügung, dessen Nutzung dem unter Linux bekannten C++–Compiler *g++* sehr ähnlich ist. Das Kompilieren der Datei erfolgt in einer Linux–Konsole mit

```
> nvcc helloworld.cu
```

Der Compiler erzeugt eine Datei namens `a.out`, deren Aufruf das erwartete Ergebnis liefert:

```
> ./a.out
Hello, World
```

[4]Es gibt noch zwei zusätzliche (optionale) Parameter, die innerhalb dieser spitzen Klammern angegeben werden können. Diese sind jedoch in dieser Arbeit nicht relevant.

[5]Man beachte, dass das Blockgitter trotzdem nur zweidimensionale Ausmaße besitzen darf. Die dritte Komponente einer `dim3`–Variable muss bei der Blockgitterdefinition also immer genau eins sein.

nvcc ist vollständig zu *g++* kompatibel, so dass Objektdateien, die mit einem Compiler erstellt wurden, beim jeweils anderen Compiler gelinkt werden können. Beim folgenden Beispiel wird *nvcc* zum Kompilieren von `helloworld.cu` verwendet. Die erstellte Objektdatei wird mit *g++* zu einer ausführbaren Datei namens `helloworld` gelinkt. Bei der Verwendung von *g++* als Linker muss die CUDA Runtime Bibliothek explizit eingebunden werden (bei *nvcc* würde das automatisch geschehen).

```
> nvcc -c helloworld.cu
> g++ -o helloworld helloworld.o -lcudart
> ./helloworld
Hello, World
```

3.2 Erweiterter C–Befehlssatz im Device Code

Da Device Code und Host Code auf unterschiedlichen Prozessorarchitekturen ausgeführt werden, ist es nicht verwunderlich, dass sich die Befehlssätze stark unterscheiden. Insbesondere sind Methoden, die mit C–Headerdateien wie etwa `#include <stdlib.h>` eingebunden werden, im Device Code nicht verwendbar, da diese natürlich nur als Host Code kompiliert werden. Es bleibt somit nach dieser starken Reduzierung des Befehlsumfangs auf den ersten Blick nur noch nativer C–Code übrig.

Die Einschränkung für den Programmierer ist jedoch nicht so groß, wie man im ersten Moment vermuten könnte. *n*Vidias *nvcc* verfügt im Gegensatz zum "normalen" ANSI–C Compiler über einen wesentlich größeren nativen Befehlssatz, so dass beispielsweise Methoden zur Threadsynchronisation oder aufwendige mathematische Berechnungen direkt vom Compiler umgesetzt werden können.

Der Übersichtlichkeit halber beschränkt sich dieser Abschnitt hauptsächlich auf Elemente, die für die spätere Implementation der vorgestellten Algorithmen relevant sind. Die Dokumentation [9] von *n*Vidia ist diesbezüglich wesentlich umfangreicher.

3.2.1 Methoden– und Variablenkennzeichner

Methodenkennzeichner werden vor der eigentlichen Methodendeklaration angegeben und legen fest, ob die folgende Methode als Device Code oder als Host Code kompiliert werden soll. Zusätzlich wird so die Methodenverfügbarkeit definiert.

`__device__` Methoden, die mit diesem Kennzeichner versehen sind, werden als Device Code kompiliert. Der Aufruf dieser Methoden ist nur vom Device Code aus möglich.

`__global__` Diese Methoden werden ebenfalls als Device Code kompiliert, können aber ausschließlich vom Host Code aus aufgerufen werden. Sie dienen als Einstiegsmethoden für Device Kernel.

Aufrufe von `__global__` Methoden sind immer *asynchron*. Das bedeutet, dass der Device Code auf der GPU gestartet wird, während der Host Code in der Zwischenzeit weiter abgearbeitet wird. Findet im Host Code ein weiterer Zugriff auf die GPU statt (weiterer Kernelaufruf, Grafikspeicher kopieren, usw...) wird an dieser Stelle gewartet, bis die `__global__` –Methode beendet wurde.

`__host__` Derart gekennzeichnete Methoden werden als Host Code kompiliert und können auch nur vom Host Code aus aufgerufen werden. Es handelt sich also um eine gewöhnliche Methode, daher kann der Kennzeichner auch weggelassen werden.

`__host__` kann zusammen mit dem `__device__` Kennzeichner verwendet werden. Dies bedeutet, dass die Methode als Host Code *und* als Device Code kompiliert wird.

Im "Hallo, Welt"–Beispiel in Kapitel 3.1 wird demnach die Methode `kernel()` als Device Code definiert, der nur vom Host aufgerufen werden kann:

```
7 __global__ void kernel()
```

Gleichermaßen können auch Variablen mit entsprechenden Kennzeichnern versehen werden. In diesem Fall soll damit die Art des Speichers festgelegt werden, in dem die Variable abgelegt wird (siehe Kapitel 2.5). Gekennzeichnete Variablen können nur im Device Code direkt verwendet werden.

`__device__` Sofern kein weiterer Kennzeichner in Kombination angegeben wurde, werden die so gekennzeichneten Variablen im Device Memory abgelegt. Sie haben in diesem Fall während der gesamten Anwendung Gültigkeit und können von allen Threads in allen Blöcken bearbeitet werden.

Der `__device__` Variablenkennzeichner darf nicht innerhalb von Funktionen verwendet werden.

`__constant__` Legt fest, dass eine Variable im Constant Memory liegt. Die Variable hat wie beim `__device__` Kennzeichner während der gesamten Laufzeit des Programms Gültigkeit und ist von allen Threads gleichermaßen zu erreichen.

Der `__constant__` Variablenkennzeichner darf ebenfalls nicht innerhalb von Funktionen verwendet werden.

`__shared__` Diese Variablen werden im Shared Memory abgelegt. Dort gespeicherte Daten sind lediglich für die Laufzeit eines Threadblocks gültig und können auch nur von den Threads im jeweiligen Block bearbeitet werden.

Dieser Kennzeichner kann innerhalb und außerhalb von Funktionen verwendet werden, um Shared Memory statisch zu reservieren:

```
__shared__ int foo[16];
__device__ void function()
{
    __shared__ int value;
    __shared__ int array[100];
}
```

3.2.2 Vektor Variablentypen

Im Device Code, wie auch im Host Code sind die folgenden Variablentypen definiert: `char1`, `uchar1`, `char2`, `uchar2`, `char3`, `uchar3`, `char4`, `uchar4`, `short1`, `ushort1`, `short2`, `ushort2`, `short3`, `ushort3`, `short4`, `ushort4`, `int1`, `uint1`, `int2`, `uint2`, `int3`, `uint3`, `int4`, `uint4`, `long1`, `ulong1`, `long2`, `ulong2`, `long3`, `ulong3`, `long4`, `ulong4`, `longlong1`, `longlong2`, `float1`, `float2`, `float3`, `float4`, `double1`, `double2`

Diese Typen sind Strukturen mit 1 bis 4 Komponenten. Die Komponenten selbst heißen `x`, `y`, `z` und `w`. Die Anzahl der Komponenten eines Vektortyps entspricht immer der Ziffer im Namen (beginnend bei `x`). Die Variablen werden mittels eines Konstruktors mit dem Namen `make_<typ>()` erzeugt. Die Verwendung dieser Datentypen zeigt Quellcode 3.2 exemplarisch am Typ `uint3`:

```
uint3 vectorvar = make_uint3(10,5,20);
// vectorvar is (10,5,20)
vectorvar.x = 2;
vectorvar.y = 3;
vectorvar.z = 4;
// vectorvar has been set to (2,3,4)
```

Quellcode 3.2: Verwendung von Vektordatentypen.

Eine Besonderheit bietet der zusätzliche Datentyp `dim3`. Er wird auch für die Definition von Gittergrößen verwendet und basiert auf `uint3`. Der Unterschied ist, dass bei der Initialisierung alle Komponenten optional sind. Komponenten, die nicht angegeben sind, werden standardmäßig auf 1 gesetzt. Ebenso unterscheidet sich die Initialisierung etwas vom `uint3`–Datentyp. Der folgende Befehl erzeugt den Vektor (4, 3, 1):

```
dim3 dimvar(4,3);
```

3.2.3 Konstanten zur Kernellaufzeit

Während der Laufzeit eines Kernels bzw. Threads sind im Device Code diverse Konstanten definiert, die die Konfiguration der Block– und Threadgitter beschreiben.

`threadIdx` ist vom Typ `uint3` und beinhaltet den Index des aktuellen Threads innerhalb des zugehörigen Threadblocks.

`blockDim` ist vom Typ `dim3` und beschreibt die Größe des Threadblocks in jeder Dimension.

`blockIdx` ist vom Typ `uint3` und beinhaltet den Index des aktuellen Threadblocks innerhalb des Blockgitters.

`gridDim` ist vom Typ `dim3` und beschreibt die Größe des Blockgitters in jeder Dimension

`warpSize` ist vom Typ `int` und beschreibt die Anzahl der Threads pro Warp.

3.2.4 Threadsynchronisation

Da Threads lediglich innerhalb eines Warps synchron laufen, ein Threadblock aber im Allgemeinen aus mehreren Warps besteht, müssen die Threads innerhalb eines Blocks ggf. synchronisiert werden. Dies ist insbesondere dann der Fall, wenn Daten, die von einem Thread im Device Memory oder Shared Memory abgelegt wurden, von Threads in anderen Warps genutzt werden sollen. Für diesen Zweck existiert im Device Code die Methode `__syncthreads()`.

```
void __syncthreads();
```

Sie wirkt wie eine synchronisierende "Barriere" innerhalb eines Threadsblocks, weil Warps an dieser Stelle so lange pausiert werden, bis alle anderen Warps des Threadblocks ebenfalls bei dieser Instruktion angekommen sind. Erst dann wird die Abarbeitung der pausierten Warps fortgesetzt. Abbildung 3.1 verdeutlicht dieses Prinzip.

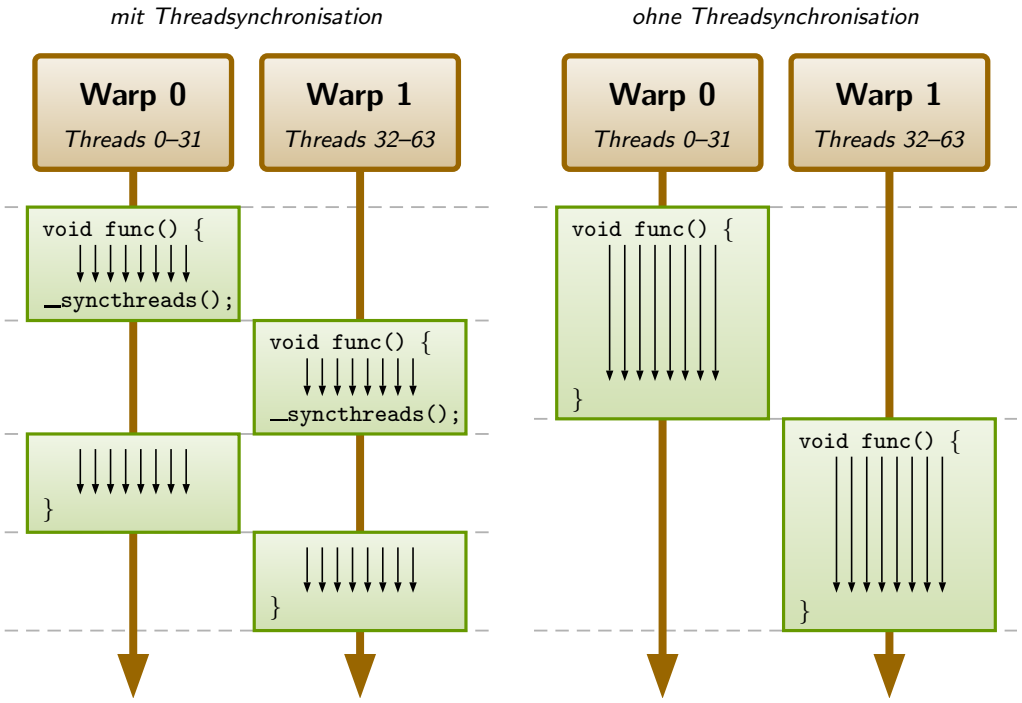


Abbildung 3.1: Ablauf einer Kernelausführung mit und ohne Threadsynchronisation.

Man beachte, dass lediglich Threads innerhalb eines Blocks synchronisiert werden können. Da die Threadblöcke im Allgemeinen nicht alle gleichzeitig bearbeitet werden, ist eine Synchronisation über mehrere Blöcke hinweg nicht möglich.

Eine etwas andere Art der Synchronisation ermöglichen `__threadfence()` und `__threadfence_block()`.

```
void __threadfence();
void __threadfence_block();
```

Diese Methoden blockieren alle Threads an dieser Stelle so lange, bis alle Schreibzugriffe auf den Speicher abgeschlossen wurden. Dabei stellt `__threadfence_block()` lediglich für den einzelnen Threadblock eine Barriere dar, während die Methode `__threadfence()` alle momentan ausgeführten Threadblöcke gleichzeitig blockieren kann.

`__threadfence()` wird in dieser Arbeit nur bei der Summierung bzw. Maximierung vieler Werte über alle Threadblöcke in Kapitel 6.3.2 benötigt.

3.2.5 Intrinsic Functions

Die Anzahl der nativ im Device Code unterstützten mathematischen Funktionen ist so groß, dass an dieser Stelle auf das Kapitel C.1 im Programmierhandbuch [9] verweisen wird.

Die meisten Methoden (wie z.B. `sin()`) sind per Software implementiert, so dass der Compiler ggf. für deren Auswertung einen Algorithmus kompiliert. Im Gegensatz zur CPU kann die GPU jedoch auch einige dieser Funktionen direkt mit Prozessorbefehlen berechnen. Diese sog. *Intrinsic Functions* benötigen nicht nur wesentlich weniger Register, die Auswertung ist in diesem Fall auch viel schneller. Tests haben ergeben, dass sie ca. 5–6 mal schneller berechnet werden als dies bei den Softwarevarianten der Fall ist. Intrinsic Functions werden im Programmierhandbuch [9] in Kapitel C.2 vorgestellt. Der Compiler kann mit dem Parameter `-use_fast_math` angewiesen werden, automatisch alle mathematischen Funktionen wenn möglich als Intrinsic Functions zu kompilieren.

Der Nachteil ist, dass die meisten dieser Prozessorbefehle nur mit `float`–Genauigkeit (*single precision*) rechnen können.

3.2.6 Atomic Functions

Ein Problem der extremen Threadparallelisierung ist, dass es manchmal schwierig ist, eine Reihe von Anweisungen auszuführen, ohne dass andere Threads die Datenintegrität beeinträchtigen. Für diese Fälle gibt es *Atomic Functions*. Sie wenden einige Befehle nacheinander auf einen bestimmten Speicherbereich an, der in dieser Zeit garantiert vor Zugriffen anderer Threads geschützt ist. Der Rückgabewert einer Atomic Function ist immer der Wert, der vor dem Aufruf im Speicher abgelegt war. Es hängt von den Compute Capabilities ab, ob eine Atomic Function im Device Memory und/oder im Shared Memory arbeiten kann (siehe Kapitel 2.6).

Die Funktionen sind für mehrere Datentypen definiert. Eine Liste der Atomic Functions zeigt Tabelle 3.1. Die Variable `old` steht hier für den Wert, der vor dem Methodenaufruf im Speicher stand.

3.2.7 Abfrage der GPU–Uhr

```
clock_t clock();
```

Eine Möglichkeit zur Laufzeitanalyse von Algorithmen bietet die Methode `clock()`. Sie liefert die Anzahl der GPU–Zyklen seit Start des Device Kernels in einer 64–Bit Zahl zurück. Es bietet sich an, diese Methode für die Codeoptimierung zu verwenden, da viele Faktoren,

Atomic Function	Wert in <T> * addr nach Aufruf	erlaubte Typen für <T>
<T> atomicAdd(<T> * addr, <T> val)	old + val	int, unsigned int, unsigned long long int
<T> atomicSub(<T> * addr, <T> val)	old - val	int, unsigned int
<T> atomicExch(<T> * addr, <T> val)	val	int, unsigned int, float unsigned long long int
<T> atomicMin(<T> * addr, <T> val)	min(old,val)	int, unsigned int
<T> atomicMax(<T> * addr, <T> val)	max(old,val)	int, unsigned int
<T> atomicInc(<T> * addr, <T> val)	(old >= val) ? 0 : (old + 1)	unsigned int
<T> atomicDec(<T> * addr, <T> val)	((old == val)|(old > val)) ? val : (old - 1)	int, unsigned int
<T> atomicCAS(<T> * addr, <T> cmp, <T> val)	(old == cmp) ? val : old	int, unsigned int, unsigned long long int
<T> atomicAnd(<T> * addr, <T> val)	old & val	int, unsigned int
<T> atomicOr(<T> * addr, <T> val)	old | val	int, unsigned int
<T> atomicXor(<T> * addr, <T> val)	old ^ val	int, unsigned int

Tabelle 3.1: Liste der Atomic Functions.

die Laufzeitmessungen stören würden (Treiberaufruf, Hintergrundprozesse, usw...) eine Messung mit `clock()` nicht beeinflussen können.

3.3 Einführung in die CUDA Runtime Library

Die Runtime Library beinhaltet Funktionen im Host Code, die u.a. der Kernelverwaltung und der Kommunikation zwischen Device Code und Host Code dienen.

Die vielen Seiten im Referenzhandbuch [10] über Funktionen der Runtime Library lassen bereits erahnen, dass hier bei Weitem keine vollständige Liste beschrieben werden kann. Vielmehr werden im Folgenden nur die Methoden vorgestellt, die für die Implementierung der Algorithmen im weiteren Verlauf dieser Arbeit wichtig sind.

3.3.1 Fehlercodes der Runtime Library

Die meisten Methoden der Runtime Library liefern einen Wert des Typs `cudaError_t` zurück. War der Aufruf fehlerfrei, dann hat die jeweilige Rückgabe den Wert `cudaSuccess`. In Ausnahmefällen kann ein Fehlercode auch die frühere Ausführung einer anderen Methode aus der Runtime Library betreffen. Dies kann insbesondere dann der Fall sein, wenn dieser Aufruf asynchron war (z.B. Kernelaufruf).

Der aktuelle Fehlercode kann immer zusätzlich mit `cudaGetLastError()` ermittelt werden, was bei einem Kernelaufruf sogar die einzige Möglichkeit zur Fehlerermittlung ist.

```
cudaError_t cudaGetLastError ();
```

Einen benutzerverständlichen Hinweis zur Bedeutung des Fehlercodes gibt die Methode `cudaGetErrorString()`.

```
const char * cudaGetErrorString (cudaError_t error);
```

Für eine genaue Angabe der möglichen Fehlercodes der jeweiligen Funktionen wird auch hier auf das Referenzhandbuch [10] verwiesen.

3.3.2 Auswahl des GPU-Chips

Besonders bei Systemen mit mehreren CUDA–fähigen Grafikkarten muss der Programmierer auswählen können, auf welcher Grafikkarte die Kernels ausgeführt werden. Zudem ist es aber generell wichtig, Informationen über verfügbare Grafikkarten, wie etwa die Compute Capabilities, ermitteln zu können. Die Runtime Library stellt hierfür die folgenden Methoden bereit:

```
cudaError_t cudaGetDeviceCount (int * count);
```

`cudaGetDeviceCount()` ermittelt die Anzahl der CUDA–fähigen Grafikkarten und gibt diese über `count` aus. Der Treiber nummeriert intern alle Grafikkarten beginnend bei 0.

```
cudaError_t cudaSetDevice (int device);
```

`cudaSetDevice()` markiert die Grafikkarte mit der Nummer `device` als aktiv. Von nun an betreffen alle Methoden der Runtime Library (Speicher kopieren, Kernel ausführen, usw...) genau diese Grafikkarte.

```
cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp * prop,
                                     int device);
```

`cudaGetDeviceProperties()` legt Informationen über die Grafikkarte mit der Nummer `device` in der Struktur `prop` ab. Quellcode 3.3 demonstriert die Interpretation der `cudaDeviceProp`–Struktur anhand einiger wichtiger Eigenschaften. Eine vollständige Dokumentation findet sich im Referenzhandbuch [10] auf Seite 11.

```
cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, devicenr);

printf("NVIDIA-Card Properties\n");
printf("  name: %s\n",prop.name);
printf("  global memory: %d MByte(s)\n", prop.totalGlobalMem/1024/1024);
printf("  shared memory: %d kByte(s)\n", prop.sharedMemPerBlock/1024);
printf("  register per multiprocessor: %d\n", prop.regsPerBlock);
printf("  warp size: %d threads\n", prop.warpSize);
printf("  max. threads per block: %d\n", prop.maxThreadsPerBlock);
printf("  max. threadblock size: %dx%dx%d\n",
          prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
printf("  max. blockgrid size: %dx%dx%d\n",
          prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2]);
printf("  constant memory: %d Byte(s)\n", prop.totalConstMem);
printf("  compute capability: %d.%d\n", prop.major, prop.minor);
printf("  clock rate: %1.0f MHz\n", (float)prop.clockRate/1000.0);
printf("  multiprocessor count: %d\n\n", prop.multiProcessorCount);

/*************************
 output on XPS M1330 Laptop:
 *************************
NVIDIA-Card Properties
  name: GeForce 8400M GS
  global memory: 127 MByte(s)
  shared memory: 16 kByte(s)
  register per multiprocessor: 8192
  warp size: 32 threads
  max. threads per block: 512
  max. threadblock size: 512x512x64
  max. blockgrid size: 65535x65535x1
  constant memory: 65536 Byte(s)
  compute capability: 1.1
  clock rate: 800 MHz
  multiprocessor count: 2
*/
```

Quellcode 3.3: Ermitteln der Grafikkarteneigenschaften.

3.3.3 Dynamische Verwaltung von Device Memory

Mit Hilfe der Runtime Library können normale Zeigervariablen aus dem Host Code auf Speicherbereiche im Device Memory zeigen. Dies bewirkt die Methode `cudaMalloc()`.

```
cudaError_t cudaMalloc (void ** devPtr, size_t size);
```

Diese Methode reserviert `size` Bytes im Device Memory. Der generierte Zeiger wird unter `*devPtr` abgelegt. Man beachte, dass dieser neue Zeiger zwar vom "normalen" Typ `void*` ist, jedoch nach der Reservierung auf für die CPU ungültige Bereiche zeigt. Eine Dereferenzierung dieses Zeigers im Host Code würde also einen Speicherzugriffsfehler verursachen. Der Zeiger kann als Parameter an einen Kernel übergeben werden.

```
cudaError_t cudaMemcpy (void *dst, const void * src,
                        size_t count, enum cudaMemcpyKind kind);
```

`cudaMemcpy()` wird wie die bekannte C–Methode `memcpy()` angewendet. Sie unterscheidet sich jedoch durch den zusätzlichen Parameter `kind`, der angibt, ob die Zeiger `dst` und `src` auf Device Memory und/oder CPU–Speicher zeigen. Die gültigen Werte für `kind` sind `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, und `cudaMemcpyDeviceToDevice`.

```
cudaError_t cudaFree (void * devPtr);
```

`cudaFree()` gibt Device Memory, das mit Hilfe der Methode `cudaMalloc()` reserviert wurde, wieder frei.

```
int main()      // Host Code
{
    // allocate some host memory
    char hostptr[100];

    // ... store some data to hostptr ...

    // define device pointer
    char * devptr;

    // allocate 100 bytes of device memory
    cudaMalloc((void**)&devptr, 100);

    // copy 100 bytes from hostptr to devptr
    cudaMemcpy(devptr, hostptr, 100, cudaMemcpyHostToDevice)

    // call a kernel and pass devptr to it
    mykernel<<<3, 5>>>(devptr);

    // free device memory
    cudaFree(devptr);
}
```

Quellcode 3.4: Dynamische Grafikspeicherverwaltung.

Speicherbereiche, die direkt im Device Code mit Hilfe von `__device__` definiert wurden, können nicht mit `cudaMemcpy()` bearbeitet werden, weil das Zeigerformat unterschiedlich definiert ist. Hier helfen die folgenden Methoden weiter:

```
cudaError_t cudaMemcpyFromSymbol (void * dst, const char * symbol,
                                  size_t count, size_t offset,
                                  enum cudaMemcpyKind kind);
```

`cudaMemcpyFromSymbol()` kopiert Daten aus einem Speicherbereich, der durch einen `__device__` oder `__constant__` Zeiger über den Parameter `symbol` angegeben wird, in den Speicherbereich `dst`, der durch `cudaMalloc()` oder einen gewöhnlichen Host Zeiger definiert wird. Bei einem Aufruf werden `count` Bytes beginnend bei `offset` kopiert. Die Zielarchitektur von `dst` wird wieder wie bei `cudaMemcpy()` durch `kind` definiert.

```
cudaError_t cudaMemcpyToSymbol (const char * symbol, const void * src,
                                size_t count, size_t offset,
                                enum cudaMemcpyKind kind);
```

`cudaMemcpyToSymbol()` funktioniert analog, mit dem einzigen Unterschied, dass `symbol` keine Quelle, sondern Ziel ist.

Die Anwendung wurde bereits beim "Hello, World"–Programm demonstriert (Quellcode 3.1 auf Seite 22).

3.4 Einschränkungen und Fehlerquellen

Obwohl der Befehlsumfang im Device Code aus mathematischer Sicht im Vergleich zum Host Code sicherlich enorm umfangreich erscheint, gibt es trotzdem noch gravierende Einschränkungen, die den Programmierer sehr deutlich spüren lassen, dass er für eine völlig andere Prozessorarchitektur programmiert.

3.4.1 Functionpointer und Rekursion

Alle Funktionen, die im Device Code aufgerufen werden, haben eine folgenreiche Eigenschaft: Sie sind "inline". Dies bedeutet, dass zwar die als `__global__` deklarierten Methoden als Einstiegspunkt im herkömmlichen Sinne zu sehen sind, während jedoch alle weiteren Funktionen, die innerhalb einer mit `__global__` gekennzeichneten Methode aufgerufen werden, vom Compiler direkt in diese Methoden hineingeschrieben werden. Dadurch ergeben sich zwei wesentliche Nachteile:

1. Es existieren keine Functionpointer, wie sie z.B. aus dem ANSI–C bekannt sind. Dies ist in gewisser Weise die direkte Konsequenz aus der Tatsache, dass aus der Sicht des Compilers keine "echten" Funktionen mit Ansprungadresse existieren. Die Definition von `__device__` –Funktionen im Device Code dient demnach ausschließlich der Übersichtlichkeit des Quellcodes.

2. Ein rekursiver Aufruf von Funktionen ist nicht möglich. Wenn der Compiler sämtlichen Device Code in eine zugrundeliegende `__global__` Methode schreibt, wäre dies bei einer rekursiven Funktion ein schier endloses Unterfangen. Schließlich steht bei rekursiven Funktionen während des Kompiliervorgangs noch nicht fest, wie oft diese sich selbst aufrufen.

3.4.2 Parallel oder nicht parallel?

Es wurde bereits in Kapitel 2.4.2 beschrieben, dass Threads eines Blocks ausschließlich innerhalb eines Warps absolut synchron laufen. Lässt man diese Tatsache außer acht, können

sich leicht Rechenfehler einschleichen.

Beim folgenden Beispiel werden benachbarte Daten aus einem Array gelesen, addiert und wieder zurückgeschrieben. Am "Rand" eines Warps werden offenbar Daten doppelt addiert, was dazu führt, dass das Element mit dem Index 32 im Array den falschen Wert hat. Der Grund dafür ist, dass hier Thread 31 und 32 nicht parallel, sondern seriell arbeiten, da sie zu unterschiedlichen Warps gehören.

```
__global__ void function(double * data)
{
    // Add adjoining values...
    double result = data[threadIdx.x-1] + data[threadIdx.x]

    // ... and write it back to the array. Now Thread 32 will compute a wrong
    // result, since Thread 31 has already written the data back to data[31].
    data[threadIdx.x] = result;
}
```

Dieser Fehler kann vermieden werden, indem die Threads nach dem Einlesen der Daten synchronisiert werden. Hierbei hilft `__syncthreads()`:

```
__global__ void function(double * data)
{
    // Add adjoining values...
    double result = data[threadIdx.x-1] + data[threadIdx.x]

    // ...wait, until threads of other warps computed the result, too.
    __syncthreads();

    // Finally write it back to the array. No computation error occurs.
    data[threadIdx.x] = result;
}
```

3.4.3 Ausgabe von Zwischenergebnissen und Fehlersuche

Ein bekanntes Problem: Nach einigen Stunden Arbeit ist ein aufwendiger Algorithmus fertig implementiert. Doch bereits beim ersten Testlauf wird deutlich, dass die berechneten Werte falsch sind. Würde man nun ein normales CPU–Programm schreiben, könnten einem an dieser Stelle Debug–Ausgaben via `printf()` weiterhelfen. Bei einem GPU–Programm stehen diese Hilfsmittel nicht zur Verfügung.

Will man nur begrenzt viele Zwischenergebnisse eines Device Kernels überwachen, kann es hilfreich sein, dem Kernel ein zusätzliches Array als Parameter zu übergeben. Der Algorithmus kann die Zwischenergebnisse dort speichern; anschließend werden sie im Host Code ausgegeben.

Eine etwas einfachere Lösung bietet der *nvcc* selbst. Kompiliert man ein Programm mit dem Compiler–Parameter `-deviceemu`, wird kein echter Device Code erzeugt. Statt dessen verpackt der Compiler den Device Kernel in einem Emulator, so dass der programmierte Device Code auf der CPU ausgeführt wird. Nun können alle normalen Methoden des Host Codes auch im Device Code verwendet werden, insbesondere auch `printf()` zur Ausgabe von Zwischenergebnissen auf dem Bildschirm.

Die Möglichkeit zur Emulation hilft in den meisten Fällen, die Fehler aufzuspüren. Es kann vorkommen, dass ein Programm im Emulationsmodus korrekt arbeitet, jedoch auf der

GPU fehlerhafte Daten produziert (oder anders herum). Dies ist ein Hinweis darauf, dass der Fehler möglicherweise durch ähnliche Konstellationen verursacht wird, wie sie im Kapitel 3.4.2 beschrieben wurden, da während der Emulation *alle* Threads seriell arbeiten.

Achtung: Für die Emulation sollte das Problem, das der implementierte Algorithmus lösen soll, minimal skaliert sein, da die Kernellaufzeit im Emulator verständlicherweise exorbitant länger ist.

Kapitel 4

Kerneloptimierung

Im Gegensatz zur Programmierung der CPU, bei der Codeoptimierung oft zu kurz kommt, ist es für die CUDA–Programmierung essentiell, sich mit diesem Thema auseinandergesetzt zu haben. Warum dies so ist, soll durch ein einfaches Beispiel verdeutlicht werden. Alle Quellcodes, die in diesem Kapitel beschrieben werden, können auf der beigelegten CD[1] gefunden werden.

Der folgende Kernel berechnet für gegebene $A, B \in M(16, 16)$

$$\|A \cdot B^T\|_F^2 = \|C\|_F^2 = \sum_{i,j} |c_{i,j}|^2 \tag{4.1}$$

mit $C \in M(16, 16)$. Das Ergebnis wird im Element $c_{1,1}$ von C abgelegt. Dafür wird ein Threadblock der Größe 16×16 verwendet.

```
1  __global__ void kernel(float * A, float * B, float * C)
2  {
3      // Index of matrix in memory
4      int mainidx = blockIdx.x*256;
5
6      // Target index in C for current thread
7      int idx = mainidx + (threadIdx.y<<4) + threadIdx.x;
8
9      // Iterators for matrices A and B
10     int aidx = mainidx + (threadIdx.y<<4);
11     int bidx = mainidx + (threadIdx.x<<4);
12
13     // Compute thread's element
14     float result = 0.0f;
15     for (int i=0; i<16; i++)
16     {
17         result += A[aidx]*B[bidx];
18         aidx += 1;
19         bidx += 1;
20     }
21
22     // square result and store to C
23     C[idx] = result*result;
24
25     // parallel summation of squares in C
26     for (int i=0; i<8; i++)
27     {
```

[1] Siehe Dateiverzeichnis ab Seite 141.

```
28          __syncthreads();
29
30          if ((idx & ((1<<(i+1))-1)) == 0)
31          {
32              C[idx] += C[idx+(1<<i)];
33          }
34      }
35  }
```

Quellcode 4.1: Kernelbeispiel ohne Codeoptimierung.

Die Matrizen sind zeilenweise als eindimensionale Arrays abgelegt. Der Kernel sieht vor, dass später für A und B jeweils mehrere Matrizen übergeben werden können, von denen jeder Threadblock i ein Paar (A_i, B_i) auswertet. Zunächst soll jedoch der Fall betrachtet werden, dass mit den Arrays `A[]` und `B[]` jeweils nur eine Matrix übergeben wurde und der Kernel mit nur einem Block gestartet wird.

Im ersten Teil des Kernels wird das Matrixprodukt berechnet, wobei der Thread (i, j) das Element $c_{i,j}$ von C berechnet.

Im zweiten Teil erfolgt eine parallele Summierung, indem das eindimensionale Array `C[]` als breiteste Ebene eines Binärbaumes interpretiert wird (siehe Beispiel in Abbildung 4.1). Die Tiefe des Baumes ist 8, da $16 \cdot 16 = 256 = 2^8$.

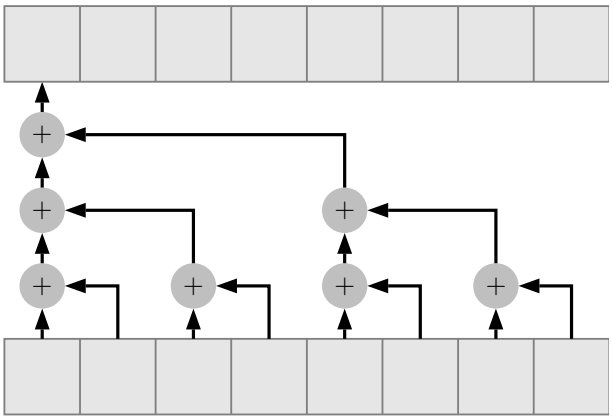


Abbildung 4.1: Intuitive Parallelisierung der Aufsummierung eines Arrays mit acht Elementen.

Obwohl nur ein Multiprozessor der GPU verwendet werden kann (lediglich ein einziger Block), spricht bei einem Laufzeitvergleich mit einem äquivalenten CPU–Programm im Vorfeld viel für die GPU. Die Auswertung erfolgt schließlich auf acht Prozessoren gleichzeitig[2], die auf mathematische Befehle optimiert sind. Die Laufzeitmessung sorgt jedoch für ernüchternde Klarheit: die GPU verliert. Abbildung 4.2 stellt einen Vergleich der Nettolaufzeiten[3] auf den für diese Arbeit verfügbaren Grafikkarten und CPUs dar. Die Bruttolaufzeiten[4] auf den Grafikkarten zeigt Abbildung 4.3.

[2] Acht Prozessoren eines Multiprozessors, siehe Kapitel 2.4.

[3] Laufzeit inkl. Treiberaufrufdauer und Hintergrundprozessen.

[4] Laufzeit eines Blocks ohne zusätzliche Verzögerungen wie z.B. Treiberaufruf, gemessen mit `clock()` (siehe Seite 28).

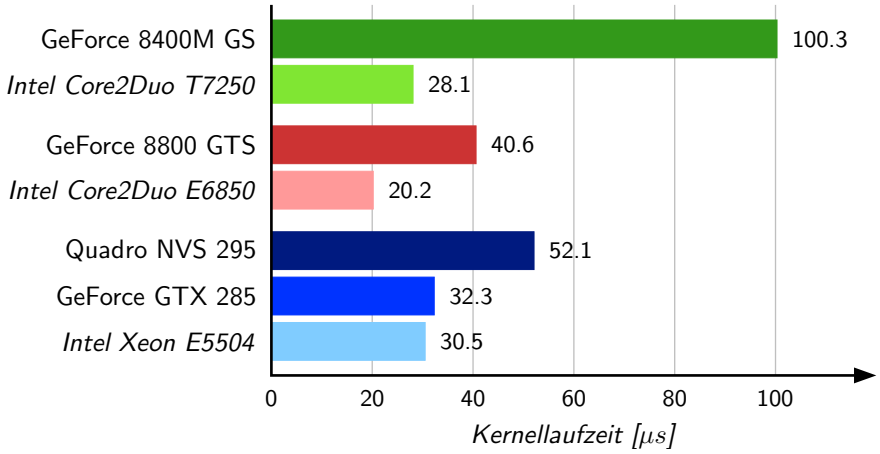


Abbildung 4.2: Vergleich der Nettolaufzeiten ohne Codeoptimierung (Quellcode 4.1) auf den verschiedenen Grafikkarten und CPUs.

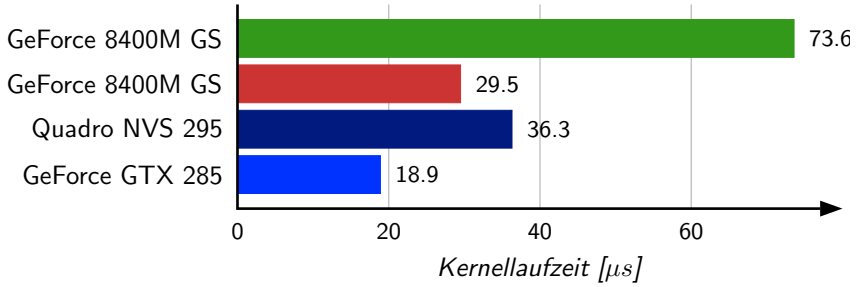


Abbildung 4.3: Vergleich der Bruttolaufzeiten ohne Codeoptimierung (Quellcode 4.1) auf den verschiedenen Grafikkarten.

In diesem Kapitel werden einige Strategien vorgestellt, die die Ausführung von Device Kernels beschleunigen können.

4.1 Die richtige Strategie zur Speichernutzung

Bei der GPU kann bereits allein mit dem richtigen Speicherzugriff die Laufzeit erheblich reduziert werden. Dies hängt nicht nur mit den unterschiedlichen Geschwindigkeiten der verschiedenen Speichertypen auf der Grafikkarte zusammen. Zusätzlich treten durch den breiten parallelen Zugriff von vielen Threads auf den Speicher Effekte auf, die man im Zusammenhang mit der CPU nicht in diesem Ausmaß beobachten könnte.

4.1.1 Paralleler Zugriff auf Device Memory

Für jeweils eine Hälfte eines Warps (16 absolut synchron laufende Threads, sog. *Half–Warp*) versucht die GPU, Speicherzugriffe zusammenzufassen. In der Literatur [8,9] ist hierbei von *Coalesced Memory Access* die Rede. Es werden bis zu 128 Bytes auf einmal vom Device

Memory gelesen oder darauf geschrieben, wobei jeder Thread eines Half–Warps für vier, acht oder 16 Byte (zwei Zugriffe zu je 128 Bytes) zuständig ist. Ob diese "Verschmelzung" funktioniert, hängt von den Device Capabilities ab.

Mit Device Capabilities 1.0 und 1.1 funktioniert diese Verschmelzung genau dann, wenn die folgenden drei Bedingungen erfüllt sind:

1. Alle Threads eines Half–Warps müssen entweder auf 4–Bytes– (64 Bytes parallel), 8–Bytes– (128 Bytes parallel) oder auf 16–Bytes–Wörter (zwei parallele Zugriffe zu je 128 Bytes) zugreifen.

2. Alle 16 Datenwörter aus **1.** müssen als zusammenhängender Block im Speicher liegen, der entsprechend seiner Größe an 64–, 128– oder 256–Bytes–Segmenten ausgerichtet ist (die Startadresse des Blocks ist ein Vielfaches der Segmentgröße).

3. Die Zugriffe dürfen sich innerhalb des zusammenhängenden Blocks nicht kreuzen, es muss also Thread i auf Datenwort i zugreifen.

Es ist jedoch erlaubt, dass sich einzelne Threads nicht am Zugriff beteiligen. Wenn also beispielsweise die obigen Bedingungen erfüllt, jedoch Thread 3 und 6 inaktiv sind, findet trotzdem noch ein verschmolzener Zugriff statt.

Sollte eine der Bedingungen verletzt sein, findet für jeden Thread ein separater Zugriff statt, was die Performance ergehlich reduzieren kann.

Ab Device Capabilities 1.2 wurden die Regeln entschärft. Jeder Thread muss auf die gleiche Wortgröße zugreifen. Die GPU teilt dann automatisch alle Zugriffe auf 32–Bytes–, 64–Bytes– und 128–Bytes–Operationen auf, so dass so wenig Bandbreite wie möglich verschwendet wird. Diese Zugriffe sind ebenfalls an der Segmentgröße ausgerichtet.

Greift also beispielsweise Thread i auf das 8–Bytes–Datenwort $i + 1$ zu, dann werden bei der ersten Operation 128 Bytes aus dem Segment gelesen, auf das die Threads 0 bis 14 zugreifen und bei der zweiten Operation 32 Bytes aus dem angrenzenden Segment für Thread 15. Obwohl auch in diesem Fall eine gewisse Bandbreitenverschwendung stattfindet, ist die Performanceeinbuße nicht so stark wie bei einem äquivalenten Zugriff mit Device Capabilities 1.0 oder 1.1.

Die Auswirkungen sollen am folgenden kleinen Beispielkernel verdeutlicht werden:

```
1 __global__ void kernel_simple(float * data, int shift)
2 {
3     int idx = threadIdx.x + shift;
4
5     for (int i=0; i<10; i++) data[idx] += 0.1f*data[idx] - 0.4f*data[idx];
6 }
```

Thread i greift hier auf ein im Speicher verschobenes Datenwort $i + \text{shift}$ zu. In Zeile 5 wird lediglich etwas Datenverkehr erzeugt, um die Zugriffszeiten erkennbar zu machen. Die Messergebnisse der Bruttolaufzeit sind in Abbildung 4.4 dargestellt.

Diese Abbildung verdeutlicht bei Compute Capabilities 1.0 und 1.1 die Ausrichtung an 64–Bytes–Segmenten, so dass nur entweder bei keiner Verschiebung, oder einer Verschiebung

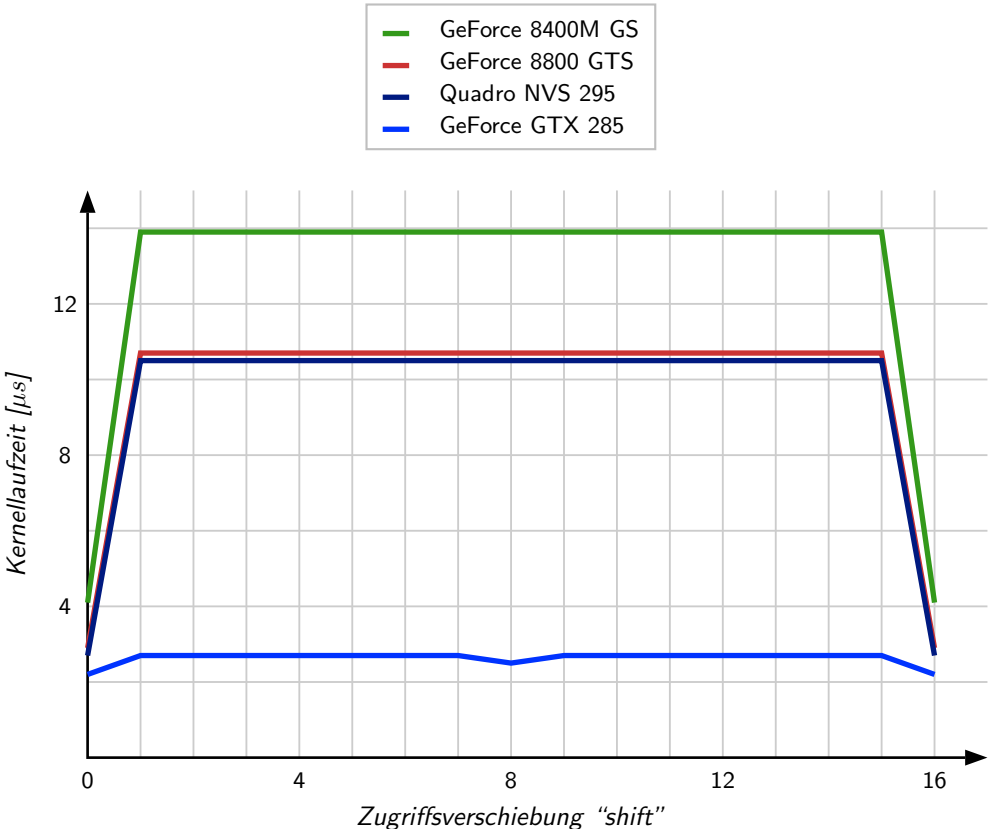


Abbildung 4.4: Bruttolaufzeiten bei verschobenem Zugriff auf Device Memory.

um ein ganzes Segment (16 Elemente zu je 4 Bytes) ein verschmolzener Zugriff stattfinden kann. Anderenfalls finden 16 separate Zugriffe statt, was die in diesem Fall relativ hohe Laufzeiten der 1.0– und 1.1–Karten erklärt.

Außerdem fällt der "Knick" der GTX 285–Karte auf, die mit Device Capabilities 1.3 arbeitet. Im ausgerichteten Fall findet ein einziger 64–Bytes Zugriff statt. Bei einer Verschiebung um genau 8 Elemente greift die GPU in zwei 32–Bytes Operationen auf den Speicher zu. Im allgemein geshifteten Fall müssen ein 64–Bytes– und ein 32–Bytes–Zugriff stattfinden, was die längste Laufzeit verursacht, jedoch immernoch im Vergleich mit der Laufzeit der 1.0– und 1.1–Karten (16 Zugriffe) wesentlich schneller ist.

Betrachtet man unter diesen Gesichtspunkten nun das Beispiel aus Quellcode 4.1, stellt man fest, dass bei der Multiplikation in Zeile 17 kein paralleler Zugriff auf Device Memory stattfindet.

```
10      int aidx = mainidx + (threadIdx.y<<4);
11      int bidx = mainidx + (threadIdx.x<<4);
12
13      // Compute thread's element
14      float result = 0.0f;
15      for (int i=0; i<16; i++)
16      {
17          result += A[aidx]*B[bidx];
```

Bei einem zweidimensionalen Threadblock sind GPU–intern die Threads zeilenweise numeriert. Es befinden sich also die Threads $(i, 0)$ bis $(i, 15)$ in Half–Warp i. Beim Lesen des Arrays `A[]` lesen demnach alle Threads eines Half–Warps die selben 4 Bytes, da der `aidx` nur von `threadidx.y`, also vom Index des Half–Warps abhängt. Mit Device Capabilities 1.0 oder 1.1 hat dies volle 16 Lesezugriffe zur Folge. Der Zugriff auf `B[]` wird ebenfalls auseinandergerissen, indem Thread i eines Half–Warps von `B[i*16]` liest.

Um diese ungünstige Konstellation zu beseitigen, müssten die Arrays zwischengespeichert werden.

4.1.2 Shared Memory vs. Device Memory

Für die Zwischenspeicherung steht dem Programmierer Shared Memory zur Verfügung. Neben der Möglichkeit, damit verschmolzene Zugriffe auf Device Memory zu konstruieren, profitiert man natürlich auch von der sehr viel geringeren Zugriffszeit auf Shared Memory.

Im folgenden Quellcode wird der Kernel aus Quellcode 4.1 entsprechend modifiziert. Die Berechnung wird nun mit Hilfsarrays im Shared Memory durchgeführt. Obwohl die temporären Arrays zweidimensional erzeugt wurden, liegen sie eindimensional im Speicher (zeilenweise). Diese Eigenschaft ist anschließend bei der parallelen Summierung wichtig.

```
1  __global__ void kernel(float * A, float * B, float * C)
2  {
3      //allocate shared memory for all matrices
4      __shared__ float As[16][16];
5      __shared__ float Bs[16][16];
6      __shared__ float Cs[16][16];
7
8      // Pointer to linear allocated shared memory of Cs
9      float * sum = &Cs[0][0];
10
11     // Starting index in device memory of current matrices
12     int mainidx = blockIdx.x*256;
13
14     // Get linear index of the current thread
15     int idx = (threadIdx.y<<4) + threadIdx.x;
16
17     // Read to Matrices to shared memory via coalesced access
18     As[threadIdx.y][threadIdx.x] = A[mainidx + idx];
19     Bs[threadIdx.y][threadIdx.x] = B[mainidx + idx];
20
21     // wait, until all threads finished reading
22     __syncthreads();
23
24     // calculate C[i,j]
25     float result = 0.0f;
26     for (int i=0; i<16; i++)
27     {
28         result += As[threadIdx.y][i]*Bs[threadIdx.x][i];
29     }
30
31     // square values
32     Cs[threadIdx.y][threadIdx.x] = result*result;
33
34     __syncthreads();
35
36     // parallel summation of all elements using the linear
37     // pointer to Cs[0][0]
38     for (int i=0; i<8; i++)
39     {
40         if ((idx & ((1<<(i+1))-1)) == 0)
```

```
41          {
42              sum[idx] += sum[idx+(1<<i)];
43          }
44          __syncthreads();
45      }
46
47      // write normresult to the first element of C
48      if (threadIdx.x == 0 && threadIdx.y == 0)
49      {
50          C[mainidx + idx] = sum[0];
51      }
52  }
```

Quellcode 4.2: Kernelbeispiel mit Zwischenspeicherung im Shared Memory

Die Laufzeitmessung in Abbildung 4.5 demonstriert die Reduzierung der Bruttolaufzeit einer Berechnung von (4.1). Diese Reduzierung folgt sowohl aus der Verschmelzung des Zugriffs auf Device Memory, als auch aus den geringeren Zugriffszeiten auf Shared Memory. In Wirklichkeit ist die Nutzung von Device Memory noch langsamer, die Wartezeiten werden von der GPU jedoch extrem effizient durch Wechseln zwischen den aktiven Warps überbrückt[5].

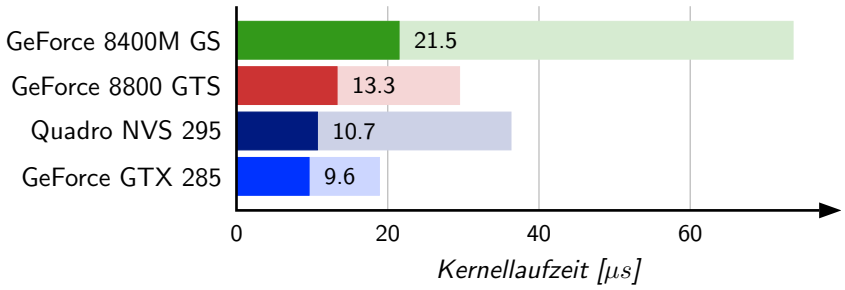


Abbildung 4.5: Vergleich der Bruttolaufzeiten mit Nutzung von Shared Memory (Quellcode 4.2) auf den verschiedenen Grafikkarten.

4.1.3 Speicherbankkonflikte

Shared Memory ist auf der GPU in 16 Speicherbänke mit jeweils 1024 Bytes aufgeteilt. Sollen mehrere Threads eines (absolut synchron arbeitenden!) Half–Warps ein und dieselbe Speicherbank ansprechen, wird der Zugriff im Allgemeinen serialisiert und man spricht von einem Speicherbankkonflikt.

Um diese zeitraubenden Konflikte vermeiden zu können, muss man die Struktur des Shared Memory kennen. Beim sequentiellen Zugriff auf Shared Memory werden die Speicherbänke immer abgewechselt, so dass auf jeder Bank jeweils vier Bytes in Folge abgelegt werden. Abbildung 4.6 demonstiert diese etwas ungewöhnliche Vorgehensweise.

Da immer nur 16 Threads eines Half–Warps gleichzeitig auf Shared Memory zugreifen, genügt es folglich, alle Half–Warps für die Suche nach Konflikten zu betrachten. Bei einem "normalen" Zugriff, bei dem alle Threads sequentiell auf aneinanderhängende Arrayelemente

[5]Siehe Kapitel 2.4.2 ab Seite 15.

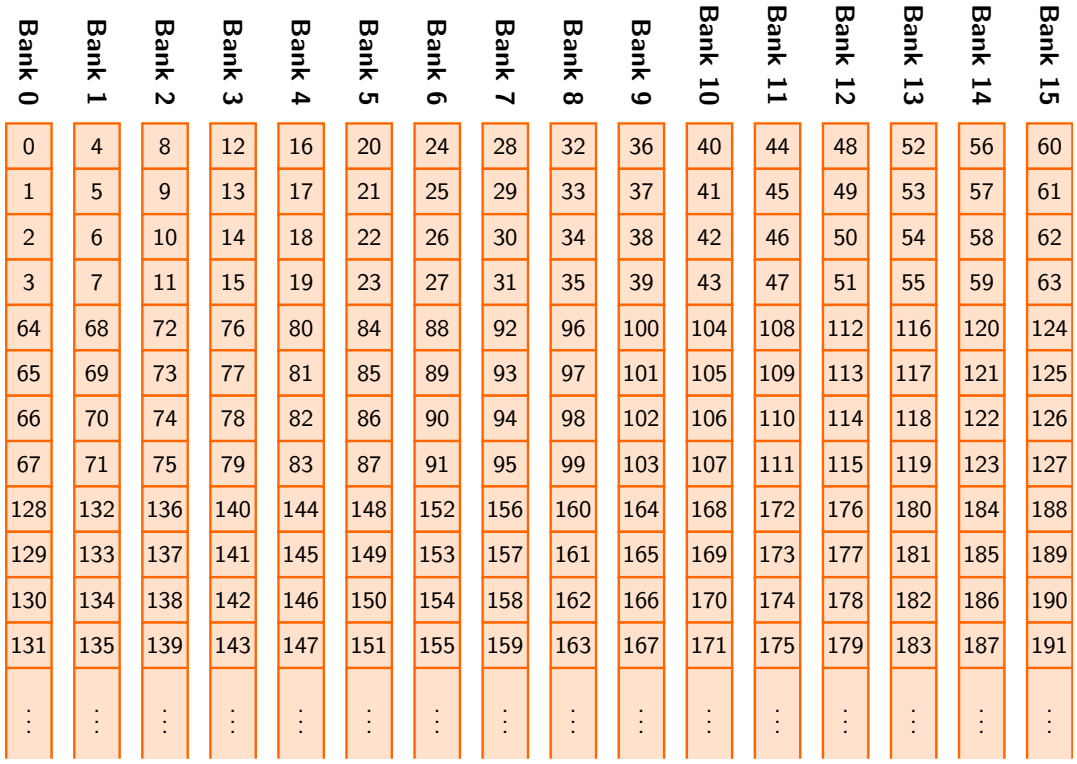


Abbildung 4.6: Nummerierung der Bytes in den Speicherbänken des Shared Memory.

zu je vier Bytes (z.B. `int`, `float`) zugreifen, treten keine Konflikte auf. Sollten aber die Arrayelemente von anderer Größe sein, oder der Zugriff nicht mehr in einer aneinanderhängenden Sequenz stattfinden, ist erhöhte Vorsicht geboten. Die folgenden Quellcodeabschnitte erzeugen beispielsweise jeweils vierfache Konflikte:

```
__shared__ char data[256];
char temp = data[threadIdx.x];
```

```
__shared__ int data[1024];
int temp = data[threadIdx.x*4];
```

Die GPU hat bei Lesezugriffen auf die Speicherbänke zusätzlich die Möglichkeit, einen Zugriff als *Broadcast* durchzuführen, um Konflikte zu reduzieren. Lesen mehrere Threads vom selben Speicherindex, findet auf diesen Index nur ein einziger Lesezugriff statt. Die Daten werden an die anfragenden Threads verteilt. Die GPU kann innerhalb eines Half–Warps diesen Broadcast jedoch nur einmal pro Zugriff durchführen. Bei mehreren verschiedenen Mehrfachzugriffen auf unterschiedliche Speicherbereiche wird der Mehrfachzugriff mit den meisten anfragenden Threads als Broadcast gesendet, die restlichen Mehrfachzugriffe verursachen Speicherbankkonflikte.

Konflikte treten ebenfalls leicht bei mehrdimensionalen Arrays im Shared Memory auf. Selbst im bereits stark beschleunigten Quellcode 4.2 lassen sich solche Konflikte finden.

```
18      As[threadIdx.y][threadIdx.x] = A[mainidx + idx];
19      Bs[threadIdx.y][threadIdx.x] = B[mainidx + idx];
20
21      // wait, until all threads finished reading
22      __syncthreads();
23
24      // calculate C[i,j]
25      float result = 0.0f;
26      for (int i=0; i<16; i++)
27      {
28          result += As[threadIdx.y][i]*Bs[threadIdx.x][i];
```

Das Einlesen der Matrizen in Zeile 18 und 19 ist unproblematisch, da immer 16 Threads eine Zeile im Shared Memory füllen (`threadIdx.y` ist bei einem Threadgitter von 16×16 innerhalb eines Half–Warps konstant). Sehr ineffizient ist jedoch die transponierte Multiplikation der Matrizen in Zeile 28. Hier lesen 16 Threads gleichzeitig eine Spalte von `Bs[][]` aus. Trotz zweier Dimensionen liegen die Daten von `Bs[][]` eindimensional im Speicher. O.B.d.A. ist dieser Zugriff also einer Adressierung wie in `data[threadIdx.x*16]` gleichzusetzen, was einen 16–fachen Speicherbankkonflikt erzeugt. Das Auslesen von `As[][]` stellt kein Problem dar, da alle Threads eines Half–Warps von der selben Adresse lesen. Dieser Zugriff findet also als Broadcast statt.

Verbesserung verspricht, die Matrix B bereits beim Einlesen in Zeile 19 zu transponieren. Zwar verursacht dies einen 16–fachen Konflikt beim Einlesen der Matrix, jedoch nicht mehr beim Auslesen während der Multiplikation. Bei 16 Multiplikationen pro einmal Einlesen, werden hier die Konflikte also bei jeder Berechnung eines Matrixelements von 256 auf 16 reduziert, was die Laufzeit im Vergleich zur konfliktreichen Berechnung mit Quellcode 4.2 spürbar verkürzt (siehe Abbildung 4.7).

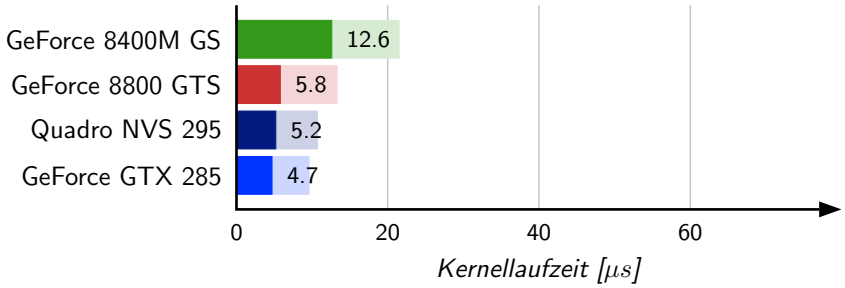


Abbildung 4.7: Vergleich der Bruttolaufzeiten nach Reduzierung der Speicherbankkonflikte auf den verschiedenen Grafikkarten.

4.2 Threadbranching

Der vorhergehende Abschnitt macht deutlich, dass eine falsche Speichernutzung die Kernellaufzeit vervielfachen kann. Unabhängig davon hält die SIMD–Architektur der GPU, wie sie bereits in Kapitel 2.3 erläutert wurde, noch einen weiteren Fallstrick für Programmierer bereit.

In Kapitel 2.3 wurde betont, dass die 32 Threads eines Warps nach dem SIMD–Prinzip absolut synchron arbeiten. Dieses Prinzip spart sich eine individuelle Programmflusssteuerung der Threads zu Gunsten von mehr Rechenleistung. Doch wie kann ein Multiprozessor, der Warps *nur* synchron bearbeiten kann, etwa den folgenden Programmcode parallel ausführen?

```
if (threadIdx.x == 0)
{
    value = 4.0f;
}
else
{
    value = d*10.0f;
}
```

Theoretisch müsste Thread 0 andere Instruktionen befolgen als der Rest des Warps. Da jedoch alle Threads eines Warps im Ablauf aneinander gebunden sind, kann die GPU diese Befehle demnach auch nicht parallelisieren.

Damit die GPU Programmcodes dieser Art verarbeiten kann, kann sie bestimmte Threads eines Warps suspendieren. Alle Threads, die von dieser Suspendierung ausgenommen sind, arbeiten trotzdem synchron. In der Praxis würde ein Multiprozessor den obigen Code also in zwei Abschnitten ausführen:

1. Verarbeite Befehl `value = 4.0f`, während die Threads 1 bis 31 suspendiert sind
2. Verarbeite Befehl `value = d*10.0f`, während Thread 0 suspendiert ist.

Diese Aufteilung des Programmcodes auf mehrere serialisierte Abschnitte nennt man *Threadbranching*.

Da beim Threadbranching immer Threads angehalten werden, wirkt sich dies natürlich negativ auf die Kernellaufzeit aus. Im Gegensatz zu Speicherzugriffen kann die GPU das Threadbranching nicht durch Umschalten zwischen aktiven Warps kompensieren und wird bei einer Aufteilung in mehrere Abschnitte immer die kumulierte Rechenzeit aller Abschnitte benötigen.

Ursachen für Threadbranching sind immer bei Fallunterscheidungen zu suchen – bedingte Schleifen, die auf dem Threadindex oder threadlokalen Variablen basieren, inbegriffen. Dabei sind die Ursachen manchmal nicht so offensichtlich, wie im obigen Quelltext. Im Beispielkernel aus Quellcode 4.2 tritt während der parallelen Summation Threadbranching auf.

```
38      for (int i=0; i<8; i++)
39      {
40          if ((idx & ((1<<(i+1))-1)) == 0)
41          {
42              sum[idx] += sum[idx+(1<<i)];
43          }
44          __syncthreads();
45      }
```

In Zeile 40 werden basierend auf der Variable `idx`, die den eindimensionalen Index im zweidimensionalen Threadgitter darstellt, die Warps künstlich "verdünnt"[6]. Für `i=0` sind 16

[6]Siehe Abbildung 4.1 auf Seite 38.

von 32 Threads eines Warps suspendiert, für `i=1` bereits 24 von 32. In beiden Fällen werden jedoch alle acht Warps[7] ausgeführt. Dabei spielt es für die Laufzeit eines Warpabschnitts keine Rolle, ob keine oder 24 Threads suspendiert sind. Insgesamt werden für die vollständige Summierung 47 Warpabschnitte benötigt (siehe Tabelle 4.1).

i	nicht suspendierte Threads pro Warp	Warpabschnitte
0	16/32	8
1	8/32	8
2	4/32	8
3	2/32	8
4	1/32	8
5	1/32	4
6	1/32	2
7	1/32	1
Warpabschnitte, kumuliert		47

Tabelle 4.1: Kumulierte Anzahl der Warpabschnitte bei intuitiver paralleler Summation.

Um das Treadbranching zu reduzieren, genügt es in diesem Beispiel, die Reihenfolge der Summierung umzustellen. Anstatt nebeneinanderliegende Werte zu addieren, werden immer zwei Werte addiert, deren Indizes in jeweils unterschiedlichen Warps liegen. Das neue Prinzip ist in Quellcode 4.3 umgesetzt und durch Abbildung 4.8 verdeutlicht.

```
1 for( int i=7; i>=0; --i)
2 {
3     int bitshift = 1 << i;
4     if (idx < bitshift)
5     {
6         sum[idx] += sum[idx+bitshift];
7     }
8     __syncthreads();
9 }
```

Quellcode 4.3: Parallele Summierung ohne Threadbranching.

Analysiert man die Summierung nach der Umstellung hinsichtlich Treadbranching, stellt man fest, dass die Anzahl der Warpabschnitte von 47 auf 12 gesunken ist (siehe Tabelle 4.2).

In diesem Beispiel ist jedoch der effektive Geschwindigkeitszuwachs des Kernels kaum messbar, da die Warpabschnitte jeweils nur eine Addition umfassen. Schleifenverwaltung und Indexberechnung benötigen wesentlich mehr Instruktionen und müssen von allen Threads durchgeführt werden.

4.3 Maximierung der GPU–Ausnutzung

Es ist bereits bekannt, dass auf einem Multiprozessor immer mehrere Blöcke gleichzeitig aktiv sein können, sofern das Blockgitter groß genug ist (o.B.d.A wird dies in diesem Abschnitt

[7] Ein Threadgitter 16×16 hat insgesamt 256 Threads, wird also in acht Warps ausgeführt.

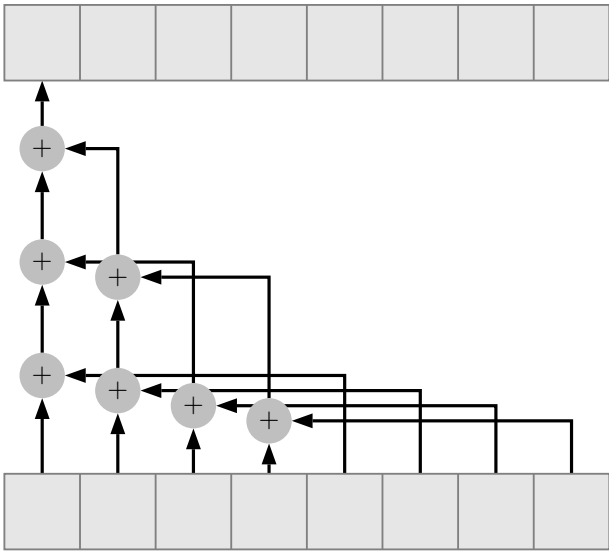


Abbildung 4.8: Parallelisierung der Aufsummierung eines Arrays mit acht Elementen ohne Threadbranching.

i	nicht suspendierte Threads pro Warp	Warpabschnitte
7	32/32	4
6	32/32	2
5	32/32	1
4	16/32	1
3	8/32	1
2	4/32	1
1	2/32	1
0	1/32	1
Warpabschnitte, kumuliert		12

Tabelle 4.2: Kumulierte Anzahl der Warpabschnitte bei optimierter paralleler Summation.

vorausgesetzt). Sei B die Anzahl dieser aktiven Blöcke. Aus der Threadblockgröße T ergibt sich die Gesamtanzahl der aktiven Threads T_{act}:

$$T_{act} = T \cdot B$$

Je nach Compute Capabilities ist eine maximale Anzahl aktiver Threads T_{max} vorgegeben. Die GPU–Ausnutzung G (sog. *Occupancy*) ist nun definiert als

$$G = \frac{T_{act}}{T_{max}} = \frac{T \cdot B}{T_{max}}$$

Um klare Aussagen über die Maximierung von G machen zu können, ist es nötig, herauszufinden, wie viele Böcke ein Multiprozessor gleichzeitig ausführen kann. Hier spielt die Größe des Device Kernels eine ausschlaggebende Rolle.

4.3.1 Die "Größe" des Kernels

Unter der Größe eines Kernels versteht man die zur Ausführung eines Threadblocks benötigten Ressourcen. Diese sind nichts anderes als die Anzahl der benötigten Register R pro Thread und das benötigte Shared Memory S pro Block in Bytes. Herausfinden kann man diese Werte, indem man den Compiler mit der Option `--ptxas-options=-v` aufruft. Der Kernel zur Berechnung von (4.1) benötigt beispielsweise nach Optimierung des Speicherzugriffs und Elimination des Threadbranchings pro Thread 11 Register und pro Block 3120 Bytes Shared Memory.

Anhand dieses Beispiels wird bereits klar, dass die Größe des Kernels oft durch das Problem gegeben wird und nur schwer modifiziert werden kann. Oft hängt das benötigte Shared Memory indirekt von der Anzahl der Threads ab. Es bietet sich in diesem Fall an, die Größe der Arrays im Shared Memory über die Anzahl der Threads per `#define` Konstante oder Ähnlichem abhängig zu machen, um anschließend S etwas variieren zu können.

Die Anzahl der benötigten Register lassen sich manchmal mit Hilfe der Compiler–Option `-maxrregcount <R>` beschränken, wobei `R` die gewünschte obere Grenze für die Anzahl der Register pro Thread darstellt. Die Einflussmöglichkeit ist jedoch auch hier ziemlich gering. Sollte der Compiler mehr Register als erlaubt benötigen, verlagert er einfach Variablen in das langsame Local Memory und verlangsamt damit gegebenfalls[8] den Kernel.

4.3.2 Ermitteln der Anzahl aktiver Blöcke

Die Anzahl der benötigten Register pro Block R_B errechnet sich durch

$$R_B = \left\lceil R \cdot \lceil T \rceil_{32} \right\rceil_{\frac{R_{max}}{32}}$$

mit R_{max} Anzahl der Register pro Multiprozessor (gegeben durch Compute Capabilities) und

$$\lceil x \rceil_y := \min_{k \in \mathbb{N}} \{ k \cdot y \mid k \cdot y \geq x \}$$

Als Anzahl aktiver Blöcke B wählt die GPU nun das maximale $b \in \mathbb{N}$ unter den Nebenbedingungen

$$\begin{aligned} b \cdot R_B = b \cdot \left\lceil R \cdot \lceil T \rceil_{32} \right\rceil_{\frac{R_{max}}{32}} &\leq R_{max} \\ b \cdot \frac{\lceil T \rceil_{32}}{32} &\leq W_{max} \\ b \cdot S &\leq S_{max} \end{aligned} \tag{4.2}$$

mit W_{max} maximale Anzahl aktiver Warps und S_{max} maximales Shared Memory (gegeben durch Device Capabilities). Bei der oben genannten Kernelgröße von $R = 11$ und $S = 3120$, ergeben sich bei einer Blockgröße von $16 \times 16 = 256$ Threads die folgende GPU–Ausnutzung G für die unterschiedlichen Device Capabilities:

[8] Sollte aufgrund der Reduzierung der Register wesentlich bessere Ausnutzung erreicht werden, könnte ein Kernel, trotz der dadurch verursachten Verlagerung einiger Variablen ins Device Memory, kürzere Laufzeiten erzielen.

Compute Capabilities 1.0 und 1.1 (GeForce 8400M GS, GeForce 8800 GTS, Quadro NVS 295):

$$G = \frac{T \cdot B}{T_{max}} = \frac{256 \cdot 2}{768} = \frac{2}{3}$$

wegen

$$\begin{aligned} 2 \cdot R_B = 2 \cdot \lceil 11 \cdot \lceil 256 \rceil_{32} \rceil_{\frac{8192}{32}} = 2 \cdot \lceil 2816 \rceil_{256} = 5632 &< 8192 = R_{max} \\ 2 \cdot \frac{256}{32} = 16 &< 24 = W_{max} \\ 2 \cdot 3120 = 6240 &< 16384 = S_{max} \end{aligned}$$

und

$$3 \cdot R_B = 3 \cdot 2816 = 8448 > 8192 = R_{max}$$

Compute Capabilities 1.3 (GeForce GTX 285):

$$G = \frac{T \cdot B}{T_{max}} = \frac{256 \cdot 4}{1024} = 1$$

wegen

$$\begin{aligned} 4 \cdot R_B = 4 \cdot \lceil 11 \cdot \lceil 256 \rceil_{32} \rceil_{\frac{16384}{32}} = 4 \cdot \lceil 2816 \rceil_{512} = 12288 &< 16384 = R_{max} \\ 4 \cdot \frac{256}{32} &= 32 = W_{max} \\ 4 \cdot 3120 = 12480 &< 16384 = S_{max} \end{aligned}$$

4.3.3 Zusammenhang zwischen GPU–Ausnutzung und Kernellaufzeit

Aus der maximalen GPU–Ausnutzung folgt die maximale Anzahl von aktiven Warps, zwischen denen ein Multiprozessor umschalten kann, um Wartezeiten zu überbrücken. Daher sollte sie stets vom Programmierer angestrebt werden.

Da die tatsächliche Kernellaufzeit jedoch von weit mehr Faktoren abhängt, dient Kernelkonfiguration mit maximalen G lediglich als Orientierung. Oft ändert sich mit einer Variation der Blockgröße auch die Anzahl der Zugriffe auf Device Memory und Shared Memory. Im Beispiel der Matrixmultiplikation müssten Speicherbereiche mehrfach vom Device Memory geladen werden, wenn der Kernel auf mehrere Blöcke aufgeteilt werden würde, um mit kleineren Blöcken die gleichen Matrizen bearbeiten zu können.

Dem Programmierer bleibt letztlich nichts anderes übrig, als Blockgrößen flexibel zu implementieren, um die optimale Kernel–Gitterkonfiguration mit Laufzeittests dem Kernel anzupassen.

4.4 Die Wahl des richtigen Blockgitters

Nachdem der Beispielkernel zur Lösung von (4.1) bzgl. Speichernutzung und Threadbranching optimiert wurde, ergeben sich die Nettolaufzeiten auf den Grafikkarten wie in Abbildung 4.9 abgebildet. Zum Vergleich wurde auch hier die Verkürzung der Nettolaufzeiten durch Codeoptimierung dargestellt.

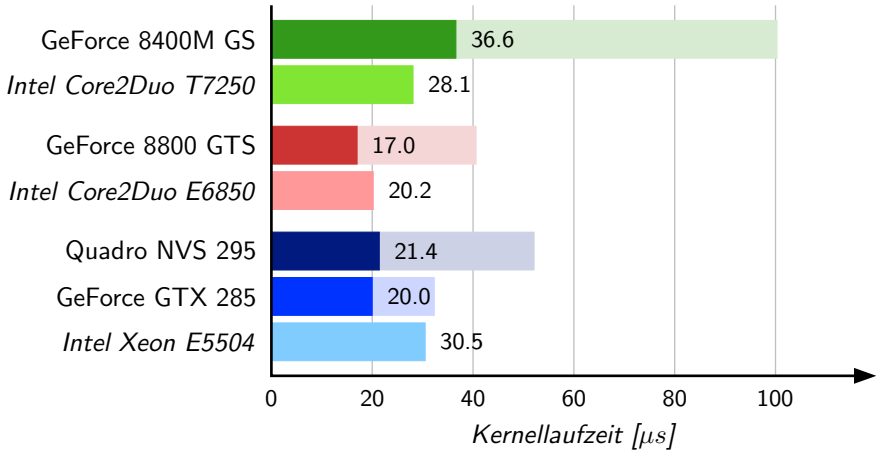


Abbildung 4.9: Verbesserung der Nettolaufzeiten durch Codeoptimierung auf den verschiedenen Grafikkarten und CPUs.

Selbst nach allen Maßnahmen zur Laufzeitverkürzung ist das Ergebnis immer noch enttäuschend. Besonders auffallend sind die fast identischen Laufzeiten bei NVS 295 und GTX 285, obwohl letztere beim Einkauf wesentlich mehr kostet. Der Grund ist, dass für diese Messungen der Kernel mit einem Blockgitter mit gerade einmal einem Element ausgeführt wurde.

Dies führt direkt zum wichtigsten Ansatz für die Codeoptimierung: Der Kernel muss so implementiert werden, dass er mit "vielen" Blöcken ausgeführt wird. Insbesondere muss natürlich der zu implementierende Algorithmus selbst auch in "viele" unabhängige Blöcke zerlegt werden können. Im weiteren Verlauf dieser Arbeit wird noch deutlich werden, dass genau diese Forderung an den zu implementierenden Algorithmus die stärkste Einschränkung bedeutet, denn die Unfähigkeit der GPU, Blöcke zu synchronisieren, setzt spezielle Strukturen im Algorithmus voraus.

In wie viele Blöcke ein Problem zerlegt werden muss, hängt von der Grafikkarte ab. In Kapitel 4.3 wurde gezeigt, dass mehrere Blöcke auf einem Multiprozessor gleichzeitig ausgeführt werden können. Daraus folgt, dass der Kernel für (4.1) beispielsweise auf der GeForce 8400M GS mit mindestens $2 \cdot 2 = 4$ (zwei Multiprozessoren, zwei Blöcke gleichzeitig bei Device Capabilities 1.1) Blöcken ausgeführt werden muss, jedoch auf der GeForce GTX 285 sogar $30 \cdot 4 = 120$ (30 Multiprozessoren, 4 Blöcke gleichzeitig bei Device Capabilities 1.3) Blöcke für eine vollständige Auslastung notwendig sind. Mit anderen Worten: Während der Laufzeitmessung in Abbildung 4.9 war die GeForce GTX 285 zu lediglich $\approx 0.8\%$ ausgelastet.

Führt man den Kernel mit mehreren Blöcken aus, wobei jeder Block i ein eigenes Tupel (A_i, B_i, C_i) als Berechnungsgrundlage hat, zeigt sich erst das wahre Potential der Grafikkarten. Abbildung 4.10 zeigt die Nettolaufzeiten dieses speziellen Kernels in Abhängigkeit der

Blockanzahl. Um auch zukünftige Grafikkarten genügend ausnutzen zu können, ist es also unvermeidbar, den Kernel auf möglichst großen Blockgittern auszuführen.

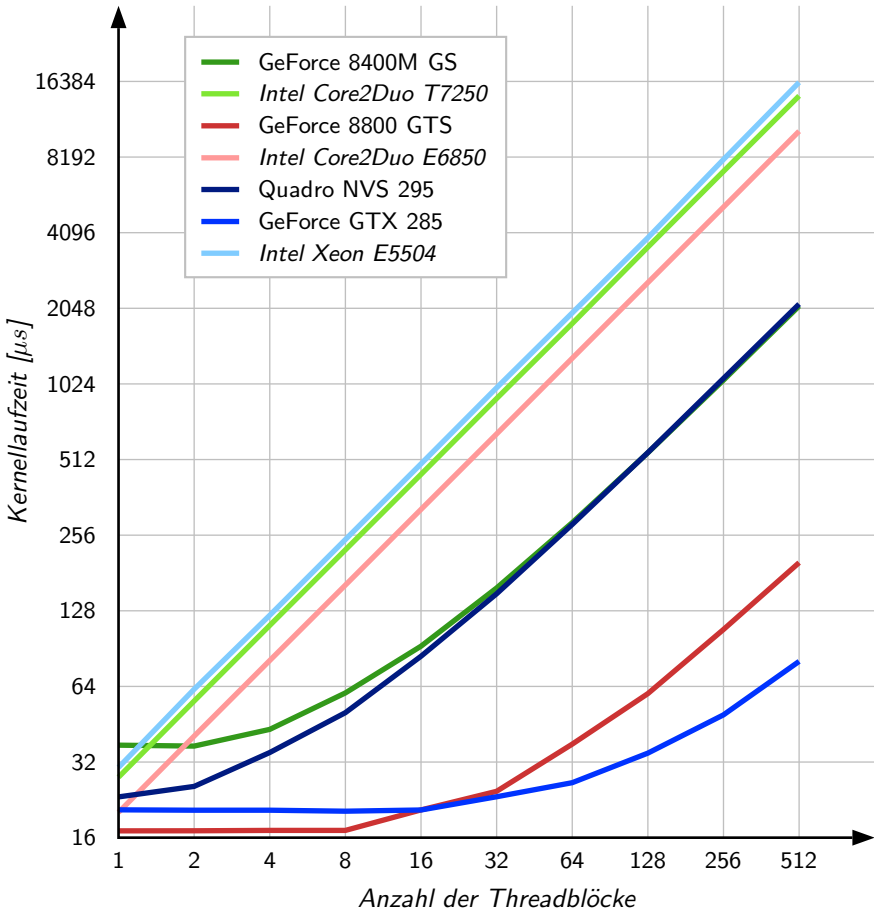


Abbildung 4.10: Vergleich der Nettolaufzeiten mit unterschiedlicher Blockanzahl auf den verschiedenen Grafikkarten und CPUs.

Teil II

Parabolische partielle Differentialgleichungen

Kapitel 5

Numerische Lösung parabolischer PDEs

Im ersten Teil dieser Arbeit wurden die technischen Kenntnisse über CUDA–Programmierung vermittelt, die einen Einstieg in die effiziente Programmierung von Device Kernels ermöglichen. Besonders im letzten Kapitel wurden auch die besonderen Anforderungen der Hardware an die Algorithmen deutlich: Gleich bleibende Instruktionen müssen auf große Datenmengen in unabhängigen Blöcken angewendet werden können, wobei nur wenig Prozessorressourcen verwendet werden dürfen.

Ob ein Algorithmus diese Vorraussetzungen erfüllen kann, ist im Vorfeld sehr schwer zu erkennen. Eine zu Beginn "übersehene" Notwendigkeit der Synchronisation der Blöcke und zu "große" Device Kernels können dem Programmierer in letzter Konsequenz einen Strich durch die Rechnung machen. Was bleibt ist eine wachsende Intuition, die dabei behilflich ist, die Eignung eines Algorithmus für die Implementierung auf CUDA–Systemen a priori einzuschätzen. Ein Fundament für diese Intuition sollen die Implementationen in dieser Arbeit legen.

> " Ich habe nicht versagt. Ich habe mit Erfolg zehntausend Wege entdeckt, die zu keinem Ergebnis führen. "
>
> Thomas Alva Edison

Algorithmen, die auf der Diskretisierung kontinuierlicher Probleme basieren, scheinen hierbei von Anfang an Erfolg zu versprechen. Die Möglichkeit, derartige Probleme über feine Diskretisierungen nahezu beliebig groß zu skalieren, könnte die Grundvoraussetzung, nämlich eine GPU mit ausreichend vielen Threadblöcken versorgen zu können, erfüllen.

Die erste exemplarische Implementierung soll demnach ein Anfangs–/Randwert–Problem einer parabolischen partiellen Differentialgleichung zweiter Ordnung berechnen. In diesem Kapitel wird die zugehörige Numerik im Hinblick auf die spätere Implementierung näher betrachtet.

5.1 Eine Problemstellung

Ein Anfangs–/Randwert–Problem einer parabolischen partiellen Differentialgleichung zweiter Ordnung ist folgendermaßen definiert:

Definition 5.1
Sei $\Omega \subset \mathbb{R}^p$ ein offenes, beschränktes Ortsgebiet und Γ der Rand von Ω. Für eine parabolische partielle Differentialgleichung ist das Anfangs–/Randwert–Problem zweiter Ordnung definiert durch

$$\begin{aligned} \frac{\partial y}{\partial t}(x,t) + Ly(x,t) &= \tilde{f}(x,t) \quad \forall (x,t) \in \Omega \times (0,T] \\ y(x,t) &= \begin{cases} 0, & (x,t) \in \Gamma \times [0,T] \\ r(x), & (x,t) \in \Omega \times \{0\} \end{cases} \end{aligned}$$

bei gegebenem $\tilde{f} : \Omega \times (0,T] \to \mathbb{R}$ und $r : \Omega \to \mathbb{R}$, wobei $y : \bar{\Omega} \times (0,T] \to \mathbb{R}$ gesucht wird. L ist für alle $(x,t) \in \Omega \times (0,T]$ ein Differentialoperator zweiter Ordnung der Form

$$Ly(x,t) = -\sum_{i,j=1}^{p} \alpha^{ij}(x,t) \frac{\partial^2 y}{\partial x_i \partial x_j}(x,t) + \sum_{i=1}^{p} \beta^i(x,t) \frac{\partial y}{\partial x_i}(x,t) + \gamma(x,t) y(x,t)$$

mit gegebenen Koeffizienten $\alpha^{ij}(x,t), \beta^i(x,t), \gamma(x,t) \in \mathbb{R}$, $i,j = 1,\ldots,p$, für alle $(x,t) \in \Omega \times (0,T]$. Das Problem heißt "semilinear", falls $\tilde{f}$ zusätzlich von $y(x,t)$ abhängt. Die Bedingung $y(x,t) = 0$ für $(x,t) \in \Gamma \times [0,T]$ heißt Dirichlet–Randbedingung.

Praktisch betrachtet kann mit diesem Anfangs–/Randwert–Problem zum Beispiel die zeitliche Entwicklung einer chemischen Konzentration in einem Gebiet modelliert werden. Dabei wird die Ausbreitung (Diffusion) der Konzentration durch die Koeffizienten $\alpha^{ij}(x,t)$, der Transport (Advektion) durch $\beta^i(x,t)$ und die lokale Veränderung (Reaktion) durch $\gamma(x,t)$ ausgedrückt. Der Anfangszustand der Konzentration ist durch r gegeben. Die Dirichlet–Randbedingung schreibt vor, dass die Konzentration am Rand des Gebiets immer 0 ist.

Für die spätere Implementierung wird als semilineare parabolische partielle Differentialgleichung eine Wärmeleitungsgleichung mit $p = 2$ und zusätzlicher Reaktions– und Advektionskomponente betrachtet. Das Gebiet Ω wird auf die offene Menge $(0,w_1) \times (0,w_2) \subset \mathbb{R}^2$ mit $w_1, w_2 > 0$ festgelegt. Mit $\alpha^{ij}(x,t) \equiv 0.1\delta_{ij}$, $\beta^i(x,t) \equiv 1$, $\gamma(x,t) \equiv 0$ und $\tilde{f}(x,t,y) = 10(y - y^3)$ ergibt sich das Anfangs–/Randwert–Problem

$$\begin{aligned} \frac{\partial y}{\partial t}(x,t) - 0.1\Delta y(x,t) + \sum_{i=1}^{2} \frac{\partial y}{\partial x_i}(x,t) & \qquad \forall\, (x,t) \in \Omega \times (0,T] \\ = 10(y(x,t) - y(x,t)^3) & \\ y(x,t) = \begin{cases} 0 & \forall\, (x,t) \in \Gamma \times [0,T] \\ r(x) & \forall\, (x,t) \in \Omega \times \{0\} \end{cases} & \end{aligned} \tag{5.1}$$

Die Lösung dieses Problems existiert und ist eindeutig (Beweis siehe [16]).

Nach Definition 5.1 ist die auf den Zeitpunkt t eingeschränkte Lösung eine Funktion $y : \Omega \to \mathbb{R}$. Das Problem ist also ∞–dimensional. Um es mit numerischen Methoden auf der Grafikkarte lösen zu können, muss das Problem auf eine endlichdimensionale Approximation transformiert werden.

5.2 Methode der finiten Differenzen

Die Art und Weise, wie das ∞–dimensionale Problem (5.1) auf eine endliche Dimension transformiert werden kann, hängt stark vom gewählten Ortsgebiet Ω ab. Wenn das Gebiet mit $(0, w_1)\times\cdots\times(0, w_p)$ sehr einfach strukturiert ist, bietet sich die in diesem Fall sehr einfach zu implementierende Methode der finiten Differenzen an. Es wird im folgenden Abschnitt angenommen, dass Ω diese Struktur besitzt.

Das Grundprinzip dieser Methode ist die Approximation der Ableitungen im Differentialoperator L (gemäß Definition 5.1) durch entsprechende Differenzenquotienten. Gleichzeitig wird anstatt des kontinuierlichen Ortsgebiets Ω nur noch eine diskrete Teilmenge $G \subset \Omega$ betrachtet. Für eine effiziente Implementierung wird G äquidistant diskretisiert. Als Schrittweite h_i des Differenzenquotienten wählt man den jeweiligen Abstand der Gitterpunkte in G.

Das Resultat ist ein großes approximierendes System gewöhnlicher Differentialgleichungen, die mit bekannten Algorithmen gelöst werden kann. Da diese gewöhnliche Differentialgleichung immer noch kontinuierlich in der Zeit ist und lediglich das Ortsgebiet diskretisiert wurde, spricht man von einer *Semidiskretisierung*.

Die folgende mathematische Darstellung der Methode der finiten Differenzen weicht geringfügig von den gängigen Definitionen in der Fachliteratur[1] ab. Die hier aufgeführte Darstellung wird sich jedoch in der späteren Implementierung in Kapitel 6.3 als sehr nützlich erweisen, da sich die hier beschriebenen Definitionen sehr gut auf die CUDA–Hardware übertragen lassen.

Definition 5.2
Gegeben ist ein offenes Ortsgebiet $\Omega = (0, w_1) \times \cdots \times (0, w_p) \subset \mathbb{R}^p$ *und* $N_1, \ldots, N_p \in \mathbb{N}$. *Die Punktmenge*

$$G := \left\{ \sum_{l=1}^{p} (k_l + 1)h_l \cdot e_l \;\middle|\; k_i = 0, \ldots N_i - 1,\ i = 1, \ldots, p \right\} \subset \Omega$$

wobei e_i *i–ter Einheitsvektor im* $\mathbb{R}^p$ *und*

$$h_i := \frac{w_i}{N_i + 1}, \quad i = 1, \ldots, p$$

ist, heißt Ortsdiskretisierung von Ω.

Definition 5.3
Sei G *Ortsdiskretisierung von* Ω, $T > 0$ *und* $y : \bar{\Omega} \times (0, T] \to \mathbb{R}$ *gegeben. Für alle* $k_1, \ldots, k_p \in$

[1] Eine Beschreibung der Methode der finiten Differenzen findet sich z.B. in [5, 15].

$\mathbb{Z}$ *ist* $y_{k_1,\ldots,k_p} : (0,T] \to \mathbb{R}$ *definiert als*

$$y_{k_1,\ldots,k_p}(\cdot) := \begin{cases} y(x,\cdot) & x \in G \\ 0 & \textit{sonst} \end{cases}$$

mit

$$x = \sum_{i=1}^{p} (k_i + 1) h_i \cdot e_i$$

Korollar 5.4
Sei G Ortsdiskretisierung von Ω. Aus $x \in G$ folgt: Es gibt eindeutig definierte $(k_1,\ldots,k_p) \in \{0,\ldots,N_1-1\} \times \ldots \times \{0,\ldots,N_p-1\}$ mit

$$x = \sum_{i=1}^{p} (k_i + 1) h_i \cdot e_i$$

Beweis: Folgt direkt aus Definition von G und der linearen Unabhängigkeit der e_i.

□

Definition 5.5
Ein Punkt $x \in G$ heißt genau dann randnah, wenn es ein $i \in \{1,\ldots,p\}$ gibt mit $x + h_i \cdot e_i \notin G$ oder $x - h_i \cdot e_i \notin G$.

Satz 5.6
Sei G Ortsdiskretisierung von Ω, $x \in G$, $T > 0$, $y : \bar{\Omega} \times (0,T] \to \mathbb{R}$ viermal stetig differenzierbar und $(k_1,\ldots,k_p) \in \{0,\ldots,N_1-1\} \times \ldots \times \{0,\ldots,N_p-1\}$ gemäß Korollar 5.4 für das gegebene x eindeutig definiert. Für $\tilde{x} \in \Gamma$ ist $y(\tilde{x},\cdot) \equiv 0$, wobei Γ der Rand von Ω ist. Dann existiert eine beschränkte Konstante $C > 0$, so dass für alle $t \in (0,T]$ und $i = 1,\ldots,p$ gilt

$$\left| \frac{\partial y}{\partial x_i}(x,t) - \frac{y_{k_1,\ldots,k_i+1,\ldots,k_p}(t) - y_{k_1,\ldots,k_i-1,\ldots,k_p}(t)}{2h_i} \right| < C h_i^2$$

und für $i,j = 1,\ldots,p$

$$\left| \frac{\partial^2 y}{\partial x_i \partial x_j}(x,t) - \left(\frac{y_{k_1,\ldots,k_i+1,\ldots,k_p}(t) - y_{k_1,\ldots,k_i+1,\ldots,k_j-1,\ldots,k_p}(t)}{h_i h_j} + \frac{y_{k_1,\ldots,k_j-1,\ldots,k_p}(t) - y_{k_1,\ldots,k_p}(t)}{h_i h_j} \right) \right| < C h_i h_j$$

wenn $i \neq j$ ($i < j$, o.B.d.A) und sonst

$$\left| \frac{\partial^2 y}{\partial x_i^2}(x,t) - \frac{y_{k_1,\ldots,k_i+1,\ldots,k_p}(t) - 2y_{k_1,\ldots,k_p}(t) + y_{k_1,\ldots,k_i-1,\ldots,k_p}(t)}{h_i^2} \right| < C h_i^2$$

Beweis: Folgt direkt aus der Definition von $y_{k_1,\ldots,k_p}$ und der Taylorreihenentwicklung (siehe [15]).

Näher zu betrachten ist der Fall, dass x ein randnaher Punkt ist. In diesem Fall werden für die Differenzenquotienten neben Punkten aus G auch Punkte aus Γ verwendet, wobei Γ und G disjunkt sind. Nach Konstruktion gilt aber für $x + h_i \cdot e_i \in \Gamma$:

$$y_{k_1,\ldots,k_i+1,\ldots,k_p}(\cdot) = y(x + h_i \cdot e_i, \cdot) \equiv 0$$

und für $x - h_i \cdot e_i \in \Gamma$:

$$y_{k_1,\ldots,k_i-1,\ldots,k_p}(\cdot) = y(x - h_i \cdot e_i, \cdot) \equiv 0$$

Die Differenzenquotienten sind somit wohldefiniert.

□

Es wird angenommen, dass die Lösung von (5.1) die Bedingungen von Satz 5.6 erfüllt. Dann kann (5.1) mit Hilfe einer Ortsdiskretisierung G durch die folgende diskrete Darstellung approximiert werden.

$$\begin{aligned}
\frac{\partial}{\partial t} y_{k_1,k_2}(t) = \; & 0.1 \left(\frac{y_{k_1+1,k_2}(t) - 2y_{k_1,k_2}(t) + y_{k_1-1,k_2}(t)}{h_1^2} + \right. \\
& \left. \frac{y_{k_1,k_2+1}(t) - 2y_{k_1,k_2}(t) + y_{k_1,k_2-1}(t)}{h_2^2} \right) - \\
& \frac{y_{k_1+1,k_2}(t) - y_{k_1-1,k_2}(t)}{2h_1} - \\
& \frac{y_{k_1,k_2+1}(t) - y_{k_1,k_2-1}(t)}{2h_2} + \\
& 10 \left(y_{k_1,k_2}(t) - y_{k_1,k_2}(t)^3 \right) \\
& \forall k_1 = 0, \ldots, N_1 - 1, \; k_2 = 0, \ldots, N_2 - 1, \; t \in (0, T] \\
y_{k1,k2}(0) = \; & r \begin{pmatrix} (k_1 + 1)h_1 \\ (k_2 + 1)h_2 \end{pmatrix}
\end{aligned} \tag{5.2}$$

Das neue diskrete Problem kann nun als Anfangswertproblem eines autonomen gewöhnlichen Differentialgleichungssystems der Dimension $N_1 \times N_2$ behandelt werden. Für $h_1, h_2 \to 0$ bzw. $N_1, N_2 \to \infty$ konvergiert die diskrete Lösung gegen die kontinuierliche Lösung von (5.1) (siehe [5]).

Man beachte, dass die Dirichlet Randbedingung bereits implizit durch die Definition von $y_{k1,k2}(\cdot)$ erfüllt ist.

5.3 Stabilisierte explizite Runge-Kutta Methode

Sei nun mit $n := N_1 \cdot N_2$ und

$$\begin{aligned}
\hat{y}(t) \; := (\; & y_{0,0}(t), \ldots, y_{0,N_2-1}(t), \\
& y_{1,0}(t), \ldots, y_{1,N_2-1}(t),
\end{aligned}$$

$$\vdots$$
$$y_{N_1-1,0}(t), \ldots, y_{N_1-1,N_2-1}(t)\ )^T \in \mathbb{R}^n$$

das Problem (5.2) durch die autonome gewöhnliche Differentialgleichung

$$\dot{\hat{y}}(t) = f(\hat{y}(t))$$

der Dimension n dargestellt, f sei hierbei die rechte Seite von (5.2).

Gewöhnliche Differentialgleichungen, die mit Hilfe der Methode der finiten Differenzen konstruiert wurden, können bei feinen Gitterdiskretisierungen sehr steif sein. Für die numerische Lösung steifer Probleme sind oft implizite Runge–Kutta–Methoden die erste Wahl.

Im Hinblick auf die Absicht, die Lösung auf der CUDA–Grafikkarte berechnen zu lassen, scheinen jedoch implizite Methoden weniger geeignet. Die Implementierung eines Lösers für nichtlineare Gleichungssysteme und der damit verbundene enorme Speicherbedarf (quadratisch zur Problemgröße), könnte bereits bei relativ groben Diskretisierungen an die Grenzen der GPU–Ressourcen stoßen.

Eine vielversprechende Alternative ist die stabilisierte explizite Runge–Kutta–Methode, die von Prof. Assyr Abdulle unter dem Namen ROCK4 veröffentlicht wurde. Es handelt sich hierbei um ein Verfahren vierter Ordnung mit Schrittweitensteuerung, das hauptsächlich für die Lösung von parabolischen PDEs, die mit der Methode der finiten Differenzen diskretisiert wurden, verwendet wird. Die Grundlage bilden Tschebyscheff–Methoden, die sich besonders gut für rechte Seiten f mit negativen Eigenwerten eignen (siehe [1]).

Die Motivation des ROCK4–Algorithmus ist, basierend auf Tschebyscheff–Polynomen, ein Runge–Kutta–Tableau zu konstruieren, das zur Laufzeit je nach Problemgestalt in der Anzahl der Stufen s variiert. Die Praxis zeigt, dass derart diskretisierte PDEs mit expliziten Methoden besser gelöst werden können, wenn die Methoden viele Stufen besitzen. Allerdings würde ein s–stufiges explizites Runge–Kutta–Verfahren mindestens s Vektoren für die Speicherung der Zwischensummen benötigen. Daher ist eine spezielle Gestalt des Runge–Kutta–Tableaus notwendig, die unabhängig von den Runge–Kutta–Stufen eine feste Anzahl von Vektoren ermöglicht.

In diesem Abschnitt soll der ROCK4–Algorithmus nach [1] als gegeben gelten und lediglich die Aspekte erarbeitet werden, die für die Konzeption einer CUDA–Implementierung relevant sind.

Definition 5.7
Seien $s \geq 1 \in \mathbb{N}$, $a_{i,j}$, b_i gegeben durch ein Tableau.

$$\begin{array}{c|ccc} & & & \\ a_{2,1} & & & \\ \vdots & \ddots & & \\ a_{s,1} & \cdots & a_{s,s-1} & \\ \hline b_1 & \cdots & b_{s-1} & b_s \end{array}$$

Eine Funktion

$$\mathbb{R}^+ \setminus \{0\} \times \mathbb{R}^n \ \rightarrow \ \mathbb{R}^n$$

$$(\hat{h}, \hat{y}) \;\mapsto\; \hat{y} + \hat{h} \sum_{i=1}^{s} b_i \hat{k}_i$$

mit

$$\hat{k}_i \;:=\; f\left(\hat{y} + \hat{h} \sum_{j=1}^{i-1} a_{i,j} \hat{k}_j\right)$$

heißt explizite Tableau–Abbildung bzgl. autonomer rechter Seite f.

Satz+Definition 5.8
Sei $s > 4 \in \mathbb{N}$, $\tilde{a}_{i,j}$, $\tilde{b}_i$, $\hat{a}_{i,j}$ und $\hat{b}_i$ gegeben. Für die expliziten Tableau–Abbildungen WP, definiert durch

$\tilde{a}_{2,1}$							
$\vdots$	$\ddots$						
$\tilde{a}_{s-4,1}$	$\cdots$	$\tilde{a}_{s-4,s-5}$					
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$				
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$	$\hat{a}_{2,1}$			
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$	$\hat{a}_{3,1}$	$\hat{a}_{3,2}$		
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$	$\hat{a}_{4,1}$	$\hat{a}_{4,2}$	$\hat{a}_{4,3}$	
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$	$\hat{b}_1$	$\hat{b}_2$	$\hat{b}_3$	$\hat{b}_4$

P, definiert durch

$\tilde{a}_{2,1}$			
$\vdots$	$\ddots$		
$\tilde{a}_{s-4,1}$	$\cdots$	$\tilde{a}_{s-4,s-5}$	
$\tilde{b}_1$	$\cdots$	$\tilde{b}_{s-5}$	$\tilde{b}_{s-4}$

und W, definiert durch

$\hat{a}_{2,1}$			
$\hat{a}_{3,1}$	$\hat{a}_{3,2}$		
$\hat{a}_{4,1}$	$\hat{a}_{4,2}$	$\hat{a}_{4,3}$	
$\hat{b}_1$	$\hat{b}_2$	$\hat{b}_3$	$\hat{b}_4$

bzgl. einer autonomen rechten Seite f gilt

$$WP(\hat{h}, \cdot) = W(\hat{h}, P(\hat{h}, \cdot))$$

Beweis: Sei $\hat{y} \in \mathbb{R}^n$.

$$\begin{aligned} WP(\hat{h},\hat{y}) &= \hat{y} + \hat{h}\sum_{i=1}^{s-4} \tilde{b}_i \hat{k}_i + \hat{h}\sum_{i=s-3}^{s} \hat{b}_{i-(s-4)}\hat{k}_i \\ &= P(\hat{h},\hat{y}) + \hat{h}\sum_{i=s-3}^{s} \hat{b}_{i-(s-4)}\hat{k}_i \end{aligned}$$

mit

$$\begin{aligned} \hat{k}_i &:= f\left(\hat{y} + \hat{h}\sum_{j=1}^{i-1} \tilde{a}_{i,j}\hat{k}_j\right), && i = 1,\ldots,s-4 \\ \hat{k}_i &:= f\left(\hat{y} + \hat{h}\sum_{j=1}^{s-4} \tilde{b}_j\hat{k}_j + \hat{h}\sum_{j=s-3}^{i-1} \hat{a}_{i,j-(s-4)}\hat{k}_j\right) \\ &= f\left(P(\hat{h},\hat{y}) + \hat{h}\sum_{j=s-3}^{i-1} \hat{a}_{i,j-(s-4)}\hat{k}_j\right), && i = s-3,\ldots,s \end{aligned}$$

Sei nun $\tilde{k}_i := \hat{k}_{i+(s-4)}$, $i = 1,\ldots,4$, dann gilt

$$\begin{aligned} WP(\hat{h},\hat{y}) &= P(\hat{h},\hat{y}) + \hat{h}\sum_{i=s-3}^{s} \hat{b}_{i-(s-4)}\hat{k}_i \\ &= P(\hat{h},\hat{y}) + \hat{h}\sum_{i=1}^{4} \hat{b}_i\tilde{k}_i = W(\hat{h},P(\hat{h},\hat{y})) \end{aligned}$$

mit

$$\tilde{k}_i = f\left(P(\hat{h},\hat{y}) + \hat{h}\sum_{j=1}^{i-1} \hat{a}_{i,j}\tilde{k}_j\right), \qquad i = 1,\ldots,4$$

□

Das s–stufige, durch WP definierte Verfahren lässt sich also in eine Kaskade aus einem $(s-4)$–stufigen und einem vierstufigen Verfahren zerlegen. Eine spezielle Wahl der Koeffizienten $\tilde{a}_{i,j}$ und $\tilde{b}_i$ erlaubt die angestrebte effiziente Auswertung des $(s-4)$–stufigen Verfahrens:

Satz 5.9
Seien $s > 4 \in \mathbb{N}$, rechte Seite $f : \mathbb{R}^n \to \mathbb{R}^n$, μ_i, ν_i, $\kappa_i \in \mathbb{R}$ mit $i = 1,\ldots,s-4$, $\hat{y} \in \mathbb{R}^n$ und Zeitschrittweite $\hat{h} > 0$ gegeben. Die Koeffizienten $\tilde{a}_{i,j}$, $\tilde{b}_i$ des Tableaus von P erfüllen die folgenden Bedingungen:

$$\begin{aligned} \tilde{a}_{i,i-1} &= -\mu_{i-1}, && \text{wenn } 2 \le i \le s-3 \\ \tilde{a}_{i,j} &= -\nu_{i-1}\tilde{a}_{i-1,j} - \kappa_{i-1}\tilde{a}_{i-2,j}, && \text{wenn } 2 \le j \le i-2,\ i \le s-3 \\ \tilde{a}_{i,j} &= 0, && \text{wenn } j \ge i \\ \tilde{b}_{i,j} &= \tilde{a}_{s-3,j}, && \text{wenn } 1 \le j \le s-4 \end{aligned}$$

$g_i \in \mathbb{R}^n$, $i = 0, \ldots, s-4$ *seien folgendermaßen rekursiv definiert:*

$$\begin{aligned} g_0 &:= \hat{y} \\ g_1 &:= \hat{y} + \hat{h}\mu_1 f(g_0) \\ g_i &:= \hat{h}\mu_i f(g_{i-1}) - \nu_i g_{i-1} - \kappa_i g_{i-2}, \qquad i = 2, \ldots, s-4 \end{aligned} \tag{5.3}$$

dann gilt

$$g_{s-4} = P(\hat{h}, \hat{y})$$

Beweis: siehe [1].

□

Sollte das Tableau von P also die spezielle Gestalt aus Satz 5.9 besitzen, kann $P(\hat{h}, \cdot)$ unabhängig von der variablen Stufenanzahl $s-4$ mit nur drei Vektoren zur Zwischenspeicherung über eine rekursive Formel berechnet werden.

Zu Beginn des Runge–Kutta–Schritts mit Anfangszustand $\hat{y}$ und mit Zeitschrittweite $\hat{h}$ wird gemäß [1] die Anzahl der Stufen s so gewählt, dass mit dem Spektralradius $\rho(\cdot)$ der Jacobimatrix von f die Bedingung

$$\hat{h} \cdot \rho\left(D_{\hat{y}} f(\hat{y})\right) \leq 0.35 s^2$$

erfüllt ist.

Für eine effiziente Schrittweitensteuerung wird die durch W definierte Methode um eine Stufe erweitert. Die eingebettete Methode $\bar{W}$ ist somit durch das Tableau

$$\begin{array}{c|cccc} & & & & \\ \hat{a}_{2,1} & & & & \\ \hat{a}_{3,1} & \hat{a}_{3,2} & & & \\ \hat{a}_{4,1} & \hat{a}_{4,2} & \hat{a}_{4,3} & & \\ \hat{b}_1 & \hat{b}_2 & \hat{b}_3 & \hat{b}_4 & \\ \hline \bar{b}_1 & \bar{b}_2 & \bar{b}_3 & \bar{b}_4 & \bar{b}_5 \end{array}$$

definiert. Die Schrittweite für den nächsten Schritt wird basierend auf

$$\varepsilon := \frac{\left\| W(\hat{h}, P(\hat{h}, \hat{y})) - \bar{W}(\hat{h}, P(\hat{h}, \hat{y})) \right\|_2}{\mathrm{tol} \cdot \sqrt{n}} \tag{5.4}$$

gemäß [4] berechnet.

Die Koeffizienten μ_i, ν_i, κ_i, $\hat{a}_{i,j}$, $\hat{b}_i$ und $\bar{b}_i$ hängen von s ab. In [1] wird konstruktiv gezeigt, dass die Koeffizienten existieren, so dass mit Hilfe der Sätze 5.9 und 5.8 $W(\hat{h}, P(\hat{h}, \cdot))$ und $\bar{W}(\hat{h}, P(\hat{h}, \cdot))$ als Methoden dritter bzw. vierter Ordnung konstruiert werden können. Zu beschreiben, wie diese Koeffizienten berechnet werden, würde den Rahmen dieser Arbeit sprengen. In der Fortran Implementierung des ROCK4–Algorithmus von Prof. Assyr Abdulle werden die Koeffizienten einer vorberechneten Tabelle entnommen, die auch für die folgende Implementierung auf der GPU verwendet wird.

Kapitel 6

Implementierung des ROCK4–Algorithmus

Für die Implementierung des ROCK4–Algorithmus kommt von allen in der Arbeit verwendeten Grafikkarten ausschließlich die GeForce GTX 285 in Frage. Die drei anderen Karten sind ungeeignet, da sie lediglich Gleitkommazahlen mit einfacher Präzision beherrschen. Die damit verbundene Genauigkeit von 7 Dezimalstellen würde die Stabilität des Algorithmus erheblich beeinträchtigen.

Der hier implementierte Algorithmus soll die gewöhnliche Differentialgleichung (5.2) lösen, die eine Diskretisierung der zweidimensionalen parabolischen partiellen Differentialgleichung (5.1) darstellt. Die Größe des Ortsdiskretisierungsgitters sei in diesem Kapitel als $N_1 \times N_2$ festgelegt, woraus für die gewöhnliche Differentialgleichung die Dimension $n = N_1 \cdot N_2$ folgt.

Das Hauptaugenmerk liegt auf dem Laufzeitvergleich zwischen der CUDA–Implementation und einer äquivalenten seriellen Implementation auf der CPU. Ein Algorithmus wie der ROCK4–Algorithmus, der bei einer steifen Differentialgleichung wie (5.2) am Rande der numerischen Stabilität arbeitet, eignet sich jedoch auch hervorragend, um die Rechengenauigkeit der GPU im Vergleich zur CPU zu betrachten.

Doch zu Beginn ist ein für die CUDA–Programmierung elementares Problem zu lösen: Die Rechenlast muss auf die vielen Prozessoren einer GPU verteilt werden, was sich hinsichtlich der speziellen Architektur der GPU alles andere als trivial gestaltet.

6.1 Der Abhängigkeitsgraph als Hilfsmittel

Ein Algorithmus hat die Eigenschaft, dass aus Eingabedaten in einer endlichen Folge von Rechenschritten (im Folgenden *Tasks* genannt) ein Endergebnis berechnet wird. Jeder Task liest Daten ein, verknüpft sie untereinander und gibt anschließend ein Teilergebnis aus. In einem Algorithmus werden endlich viele Tasks in der Art miteinander verknüpft, dass die Eingabedaten der Tasks von den Ausgaben anderer Tasks oder zumindest von den Eingabedaten des Algorithmus abhängen. Fasst man die Menge der Tasks als Knoten und die Abhängigkeiten als gerichtete Kanten auf, dann lässt sich ein Algorithmus als ein gerichteter Graph darstellen. Dieser Graph erweist sich bei einer Analyse hinsichtlich paralleler Ausführung von Tasks als sehr nützlich.

Definition 6.1
Gegeben seien eine Menge von Tasks $T := \{t_i, \ldots, t_N\}$ *und eine Menge von Tupeln* $D := \{(t_{k_i}, t_{l_i})_i \mid k_i, l_i \in \{1, \ldots, N\},\ k_i \neq l_i,\ i = 1, \ldots, M\}$*, wobei die Elemente von* D *paarweise verschieden sind. Die Menge* T *sei genau die Menge der Tasks eines Algorithmus* A*. Es gelte*

$$\begin{array}{c}\textit{In Algorithmus } A \textit{ benötigt Task } t_j \\ \textit{die Ausgabedaten von Task } t_i\end{array} \Leftrightarrow (t_i, t_j) \in D$$

Das Tupel (T, D) *heißt Abhängigkeitsgraph von* A*.*

Definition 6.2
Gegeben sei ein Abhängigkeitsgraph (T, D)*. Eine Teilmenge*

$$\{(t_{i_1}, t_{i_2}), (t_{i_2}, t_{i_3}), \ldots, (t_{i_{k-2}}, t_{i_{k-1}}), (t_{i_{k-1}}, t_{i_k})\} \subset D$$

mit $t_{i_1} = t_a$ *und* $t_{i_k} = t_b$*,* $a, b \in \{1, \ldots, N\}$ *heißt Weg von* t_a *nach* t_b *in* D*.*

Definition 6.3
Gegeben sei ein Abhängigkeitsgraph (T, D) *und* $t_i, t_j \in T$*.* t_j *heißt unabhängig von* $t_i \Leftrightarrow$ *Es existiert kein Weg von* t_i *nach* t_j *in* D*.*

Da zwei unabhängige Tasks keine gegenseitigen Ausgabedaten benötigen, liegt es auf der Hand, dass diese Tasks parallel ausgeführt werden können, sofern beim Start der parallelen Ausführung bereits die Eingabedaten der beiden Tasks vorliegen. Unabhängige Tasks können also von zwei unterschiedlichen Threads ausgeführt werden. Um die Struktur der Threadblöcke berücksichtigen zu können, muss die Gruppierung von Tasks formalisiert werden.

Definition 6.4
Gegeben sei ein Abhängigkeitsgraph (T, D)*. Ein Tupel* $(\hat{T}, \hat{D})$ *mit den Eigenschaften*

- $\hat{T} \subset T$ *und* $\hat{D} \subset D$
- *Für alle* $(t_i, t_j) \in D$ *gilt:* $t_i, t_j \in \hat{T} \Leftrightarrow (t_i, t_j) \in \hat{D}$
- *Sei* $t_a, t_b \in \hat{T}$*. Für alle Wege* $\{(t_{i_1}, t_{i_2}), (t_{i_2}, t_{i_3}), \ldots, (t_{i_{k-1}}, t_{i_k})\} \subset D$ *mit* $t_{i_1} = t_a$ *und* $t_{i_k} = t_b$ *gilt* $t_{i_j} \in \hat{T}$*,* $j = 1, \ldots, k$*.*

heißt separater Teilgraph von (T, D)*.*

Definition 6.5
Gegeben seien Abhängigkeitsgraph (T, D) *und die separaten Teilgraphen* $(\hat{T}_1, \hat{D}_1)$ *und* $(\hat{T}_2, \hat{D}_2)$ *von* (T, D)*. Es gelte*

- $\hat{T}_1 \cap \hat{T}_2 = \emptyset$
- *Für alle* $t_1 \in \hat{T}_1$*,* $t_2 \in \hat{T}_2$ *gilt: Es existiert kein Weg von* t_1 *nach* t_2 *in* D*.*

Dann heißt $(\hat{T}_2, \hat{D}_2)$ *unabhängig von* $(\hat{T}_1, \hat{D}_1)$*.*

Die wichtige Eigenschaft eines separaten Teilgraphen ist, dass es keinen Weg geben kann, der teilweise außerhalb des Teilgraphen liegt. Es kann also niemals eine wechselseitig gerichtete Abhängigkeit zwischen zwei separaten Teilgraphen $(\hat{T}_1, \hat{D}_1)$ und $(\hat{T}_2, \hat{D}_2)$ mit $\hat{T}_1 \cap \hat{T}_2 = \emptyset$ geben, weshalb die Teilgraphen grundsätzlich nacheinander abgearbeitet werden können, bzw. bei Unabhängigkeit sogar parallel.

Diese Struktur lässt sich nun nahezu direkt auf die CUDA–Hardwarearchitektur abbilden: Alle Tasks werden zu separaten Teilgraphen gruppiert (mit paarweise disjunkten Taskmengen). Jeder separate Teilgraph wird in der Praxis auf einem Threadblock ausgeführt, während die darin enthaltenen Tasks von den Threads im jeweiligen Threadblock bearbeitet werden. Die Gruppierung muss so gewählt werden, dass möglichst viele Teilgraphen, sowie möglichst viele darin enthaltene Tasks, unabhängig und sich zugleich strukturell sehr ähnlich sind (siehe *Threadbranching*, Kapitel 4.2). Das Limit von 512 Threads pro Block und alle weiteren Aspekte im Zusammenhang mit GPU–Ressourcen (limitierter Zwischenspeicher und Register, Zugriffe auf Device Memory minimieren, usw...) sind bei der Konzeption ebenfalls im Auge zu behalten. Auf diese Weise werden zwei essentielle Fragen beantwortet, die vor jeder Programmierung von Device Kernels gestellt werden müssen:

1. *Welche verschiedenen Device Kernels werden benötigt?* Für jede *unterschiedliche* Struktur eines separaten Teilgraphen wird ein eigener Device Kernel benötigt.

2. *Mit welchem Blockgitter und welcher Threadblockgröße wird ein Device Kernel aufgerufen?* Als Blockgitter ist eine Nummerierung der unabhängigen separaten Teilgraphen des Device Kernels nach Punkt 1 zu wählen. Die Blockgröße wird so groß gewählt, dass so viele Tasks wie möglich parallel bearbeitet werden können.

Im Gegensatz zu einem *Nassi–Shneiderman–Diagramm* können in einem Abhängigkeitsgraphen Fallunterscheidungen und Schleifen nicht dargestellt, sondern lediglich beispielsweise über Wiederholungen oder alternative Graphen angedeutet werden.

Es bleibt abschließend noch zu erwähnen, dass ein Abhängigkeitsgraph zwar für eine konkrete Implementation eines Algorithmus eindeutig ist, im Allgemeinen aber nicht für den Algorithmus selbst. Durch geeignete Umstrukturierung eines Algorithmus kann (wie am Beispiel der parallelen Summation in Kapitel 4) der Abhängigkeitsgraph stark variieren und ggf. eine breitere Parallelisierung ermöglicht werden. Konzepte für derartige Umstrukturierungen zur besseren parallelen Auswertung von Algorithmen finden sich z.B. in [2] und [14]. Zudem ist die Einteilung eines Abhängigkeitsgraphen lediglich als Entwurf zu werten. Es kann sich im Laufe der Implementierung eine leichte Anpassung der Einteilung als günstig erweisen, um Strukturen der Architektur effizienter ausnutzen zu können.

6.2 Konzept zur Kernelgestaltung

Die Erarbeitung eines Kernelkonzepts erfolgt natürlich anhand eines Abhängigkeitsgraphen des ROCK4–Algorithmus. Ist dieser Graph konstruiert, kann sich anschließend über die Aufteilung der Rechenschritte auf die verschiedenen Threadblöcke und ggf. unterschiedliche Kernels Gedanken gemacht werden.

6.2.1 Abhängigkeitsgraph eines Runge–Kutta–Schritts

Ein Runge–Kutta–Schritt wird im ROCK4–Algorithmus basierend auf Zustand $\hat{y} \in \mathbb{R}^n$ und Zeitschrittweite $\hat{h}$ wird wie folgt berechnet:

1. Der Spektralradius wird durch Gerschgorin–Kreise abgeschätzt und die Anzahl der Stufen s ermittelt. Die Berechnung der n Kreise erfolgt unabhängig voneinander.

2. Die Dreifachterm–Rekursion wird $(s-4)$ mal berechnet (Methode P). Die Komponenten von g_0 und g_1 können noch unabhängig voneinander bestimmt werden. Für die Berechnung der nächsten Iterierten g_i nach (5.3) ist eine Auswertung von f notwendig. Die darin enthaltene Berechnung des zweidimensionalen Differenzenquotienten benötigt in einer Komponente vier weitere, in der Ortsdiskretisierung benachbarte, Komponenten von g_{i-1}. Abbildung 6.1 verdeutlicht diese Abhängigkeiten. Das gewichtete Addieren dieser Funktionsauswertung mit den Iterierten g_{i-1} und g_{i-2} erfolgt komponentenweise unabhängig voneinander. Die Iterierten g_i müssen in jedem Schritt neu berechnet werden.

3. Das vierstufige Runge–Kutta–Schema mit eingebetteter Methode wird angewendet (Methode W bzw. $\bar{W}$). Hinsichtlich der Abhängigkeiten ist diese Berechnung der Dreifachterm-Rekursion relativ ähnlich. Für die Berechnung der i–ten Stufe des Runge–Kutta–Schemas

$$\hat{k}_i := f\left(\hat{y} + \hat{h}\sum_{j=1}^{i-1} \hat{a}_{i,j}\hat{k}_j\right) \tag{6.1}$$

muss erneut eine Summe über n–dimensionale Vektoren gebildet werden. Die Auswertung der einzelnen Komponenten erfolgt ebenfalls unabhängig voneinander, es werden allerdings $\hat{k}_1, \ldots, \hat{k}_{i-1}$ benötigt. Auch für die Berechnung von f sind wie bereits in Abbildung 6.1 vier in der Ortsdiskretisierung benachbarte Komponenten nötig. Man beachte, dass sich die Koeffizienten $\hat{a}_{i,j}$ mit jedem Schritt ändern können und auch alle Variablen $\hat{k}_i$ in jedem Schritt neu definiert werden.

4. ε wird gemäß (5.4) berechnet. Die einzelnen Summanden werden dabei unabhängig voneinander bearbeitet.

5. Eine neue Schrittweite wird bestimmt und der Schritt ggf. akzeptiert.

Insgesamt ergibt sich für einen Runge–Kutta–Schritt der Abhängigkeitsgraph, wie er auf Abbildung 6.2 dargestellt wird. Man beachte, dass z.B. Abhängigkeiten wie $\hat{k}_1, \ldots, \hat{k}_{i-2} \to \hat{k}_i$ für (6.1) nicht abgebildet werden, da sie bereits indirekt über die Abhängigkeit von $\hat{k}_{i-1}$ gegeben sind. Ebenso werden aus Gründen der Übersichtlichkeit nur zwei der vier Nachbarschafts–Abhängigkeiten aus Abbildung 6.1 berücksichtigt. Die Problematik bleibt dadurch unverändert.

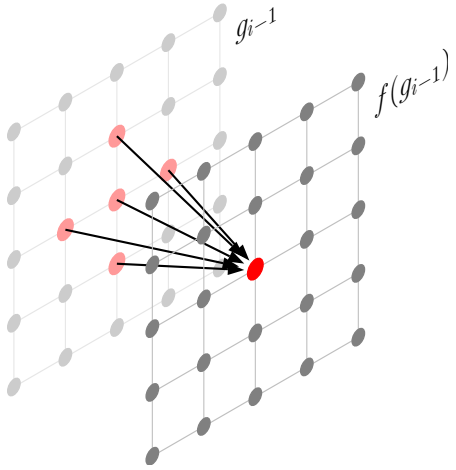


Abbildung 6.1: Veranschaulichung der Abhängigkeiten in der Ortsdiskretisierung bei der Berechnung eines zweidimensionalen Differenzenquotienten durch f am Beispiel der Dreifachterm–Rekursion.

6.2.2 Abbildung auf die CUDA–Hardware

Es gilt nun herauszufinden, wie die Tasks des Graphen in Threadblöcke eingeteilt werden können. Hierfür wurde als Hilfsmittel auf Seite 66 der Begriff des unabhängigen separaten Teilgraphen eingeführt.

Da die Hardware erwartet, dass die Tasks in einem Threadblock möglichst unabhängig sind und aus den gleichen Instruktionen bestehen sollen (siehe *SIMD*, Kapitel 2.3), kommt hinsichtlich dieser Anforderung in diesem Fall nur eine zeilenweise Separierung des Graphen in Frage. Wegen der kreuzenden Abhängigkeiten in Abbildung 6.2 existieren jedoch Wege, die zwischen den Zeilen hin und her wechseln. Eine zeilenweise Separierung würde demnach keine separaten Teilgraphen per Definition erzeugen – erst recht keine unabhängigen Teilgraphen.

Wenn also die Dimension der Differentialgleichung nicht zufällig kleiner als 512 ist, so dass der gesamte Graph als unabhängiger Teilgraph betrachtet und somit in einem einzelnen Threadblock ausgeführt werden kann, müssen zusätzlich zu den zeilenweisen auch noch spaltenweise Aufteilungen des Graphen durchgeführt werden.

Die roten Linien in Abbildung 6.3 deuten die Gruppierungen der Tasks zu separaten Teilgraphen an, deren Abhängigkeiten durch rote Pfeile signalisiert werden. Die spaltenweise angeordneten Teilgraphen sind unabängig und können somit parallel berechnet werden. Bei den kreuzenden Abhängigkeiten zwischen den Teilgraphen müssen die Blöcke synchronisiert werden. Dies kann nur durch Beenden und Neustart eines Device Kernels erreicht werden.

Insgesamt gibt es in dieser Einteilung neun unterschiedliche Strukturen der separaten Teilgraphen; es können also bis zu neun verschiedene Device Kernels nötig sein. Vorausgesetzt, es gibt $T_1, T_2, B_1, B_2 \in \mathbb{N}$, so dass $N_1 = T_1 \cdot B_1$ und $N_2 = T_2 \cdot B_2$, so ist gemäß der Einteilung in Abbildung 6.3 ein Blockgitter der Größe $B_1 \times B_2$ nötig. Die Threadblöcke haben die Größe $T_1 \times T_2$, wobei jeder Thread (i, j) des Blocks (k, l) das Element $(kT_1 + i, lT_2 + j)$ der Ortsdiskretisierung bearbeitet.

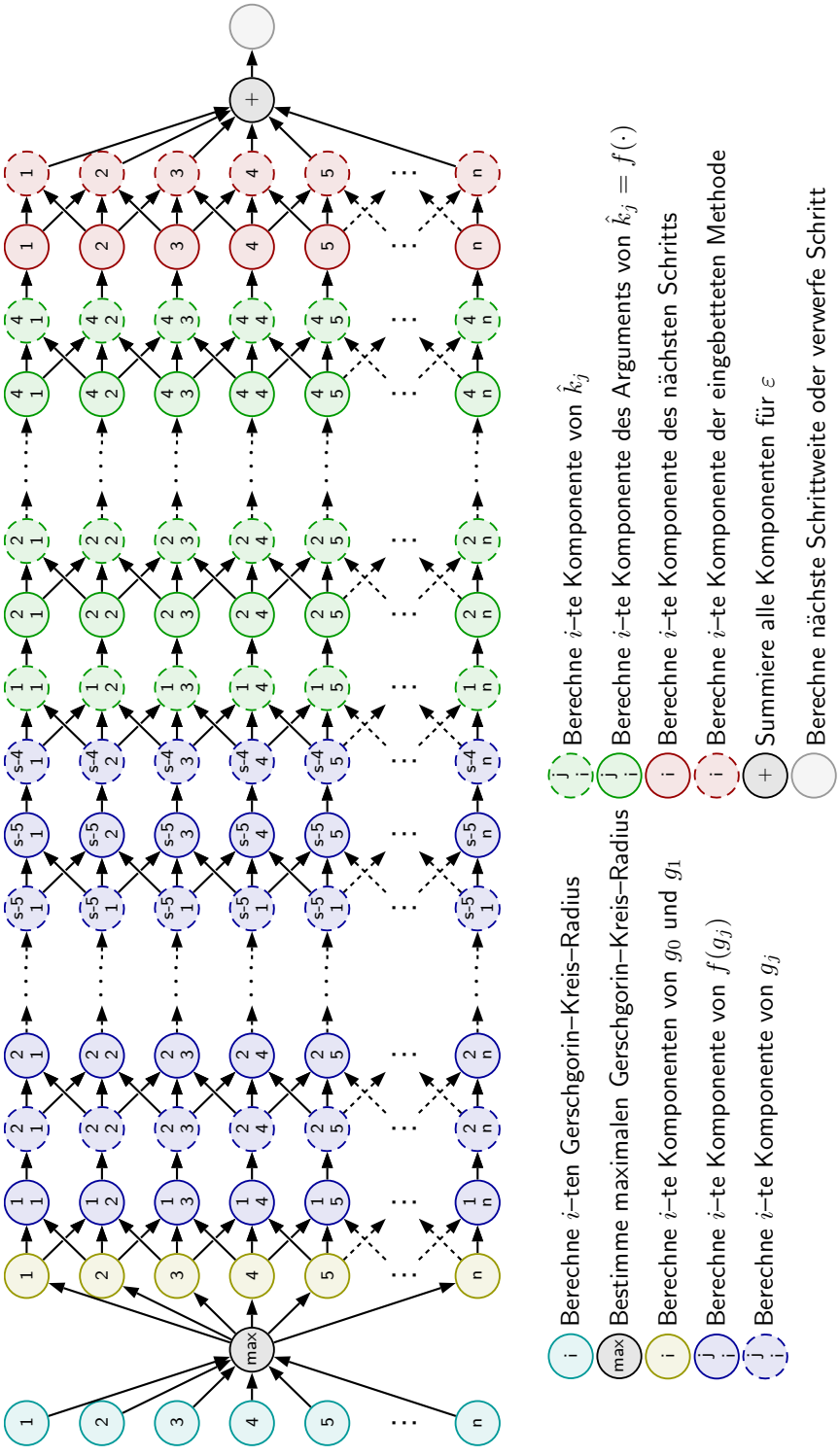


Abbildung 6.2: Abhängigkeitsgraph eines Runge–Kutta–Schritts der s–stufigen Methode $W(\hat{h}, P(\hat{h}, \cdot))$ inkl. Schrittweitensteuerung.

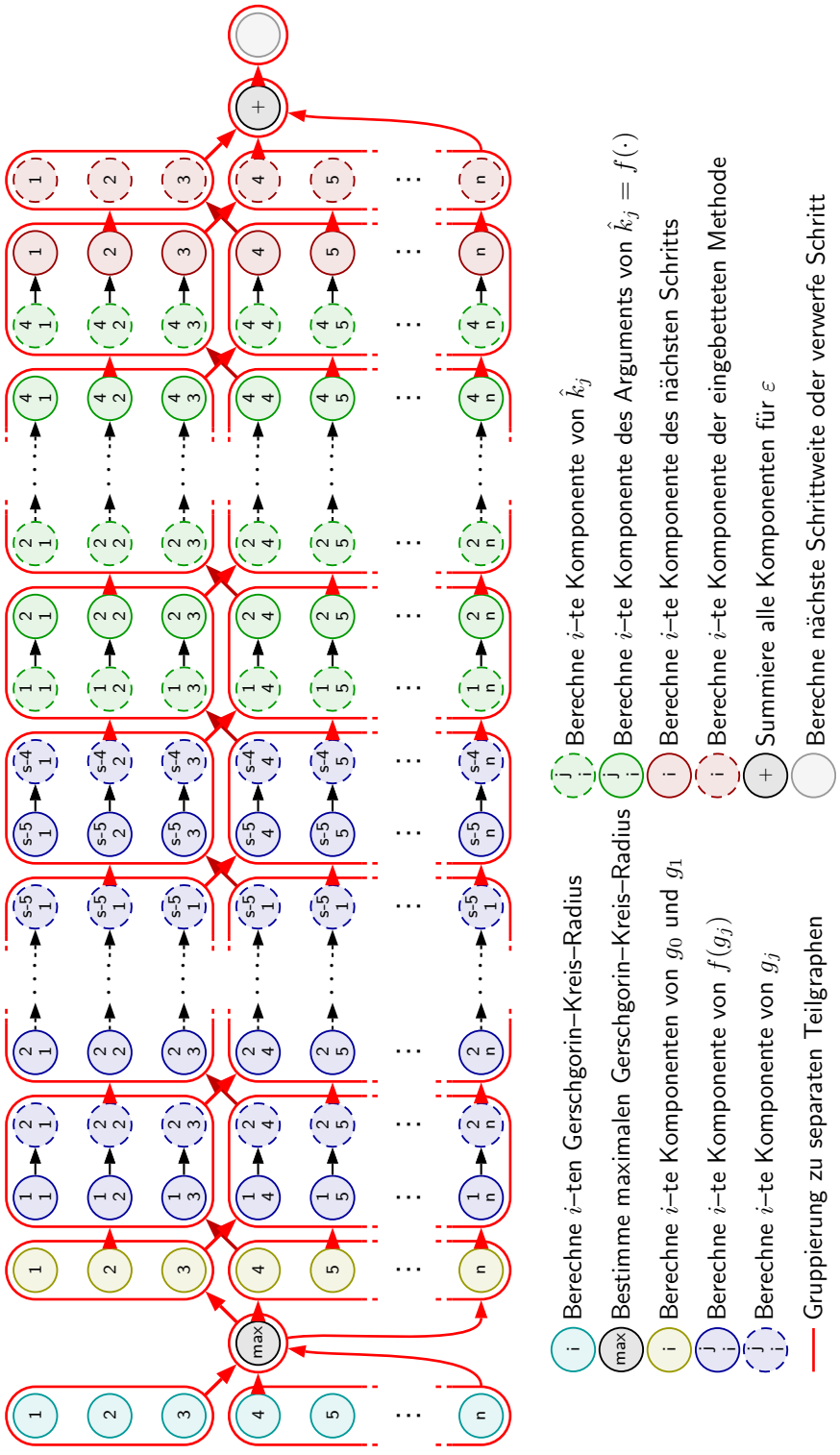


Abbildung 6.3: Abhängigkeitsgraph aus Abbildung 6.2, in separate Teilgraphen eingeteilt.

6.3 Exemplarische Umsetzung

Alle CUDA–spezifischen Methoden des ROCK4–Algorithmus werden in einer Datei namens `cudarock4kernel.cu` implementiert, die mit dem *nvcc*–Compiler kompiliert werden kann. Der Abhängigkeitsgraph von Abbildung 6.3 hat bereits deutlich gezeigt, dass mehrere verschiedene Device Kernels programmiert werden müssen.

Aus Platzgründen wird in dieser Arbeit nicht auf die Implementierung des Host Codes eingegangen. Der vollständige kommentierte Quellcode kann jedoch auf der beigelegten CD[1] eingesehen werden.

Da der Device Code keine Functionpointer unterstützt, muss die zu berechnende diskretisierte partielle Differentialgleichung (5.2) fest im Code angegeben werden. Für eine andere rechte Seite der Differentialgleichung müssen demnach auch andere Kernels geschrieben werden.

6.3.1 Speicherung der Konstanten

Die Parameter der PDE werden gemäß (5.2) mit `#define`–Direktiven festgelegt.

```
10 #define NU 0.1      // Diffusion
11 #define MU 10.0     // Reaction
12 #define ADVX 1.0    // 2-dim Advection
13 #define ADVY 1.0
```

Ebenso wird die Größe der Threadblöcke auf diese Art und Weise definiert. Dies erlaubt eine spätere Modifikation zur Ermittlung der optimalen Blockgröße. Die Größe der Threadblöcke wird in Zweierpotenzen gewählt, um später Berechnungen mittels Bitoperationen beschleunigen zu können.

```
15 #define TSHIFTX 4   // Set blocksize to 16x16
16 #define TSHIFTY 4
17 #define TSIZEX (1<<TSHIFTX)
18 #define TSIZEY (1<<TSHIFTY)
```

Die Koeffizienten der Runge–Kutta–Tableaus werden in vorberechneter Form der Fortran–Code Vorlage von Prof. Assyr Abdulle entnommen. Die Koeffizienten sind für 50 verschiedene Stufenanzahlen s vorberechnet und werden im separaten Headerfile `cudarock4kernel_constants.h` gesetzt.

```
20 __constant__ double fpa[300];   // Coeff. for 4-stages method
21 __constant__ double fpb[200];   // Weights for 4-stages method
22 __constant__ double fpbe[250];  // Weights for embedded method
23 __constant__ double recf[4382]; // Coeff. for three-term-recursion
```

Somit werden bei acht Bytes pro `double`–Variable 41056 Bytes von 64kB Constant Memory reserviert.

[1] Siehe Dateiverzeichnis ab Seite 141.

6.3.2 Abschätzung des Spektralradius

Die Berechnung der Gerschgorin–Kreis–Radien und die Bestimmung deren Maximums können mit einem kleinen Trick in einem einzigen Kernel zusammengefasst werden. Die Berechnung des Maximums kann wie in Kapitel 4 als Binärbaum parallelisiert werden. Hinderlich ist jedoch auf den ersten Blick, dass das Maximum erst ermittelt werden kann, wenn *alle* Kreisradien berechnet wurden, was jedoch auf mehrere Blöcke aufgeteilt ist. Abbildung 6.4 verdeutlicht die parallele Auswertung und deren Blockabhängigkeiten.

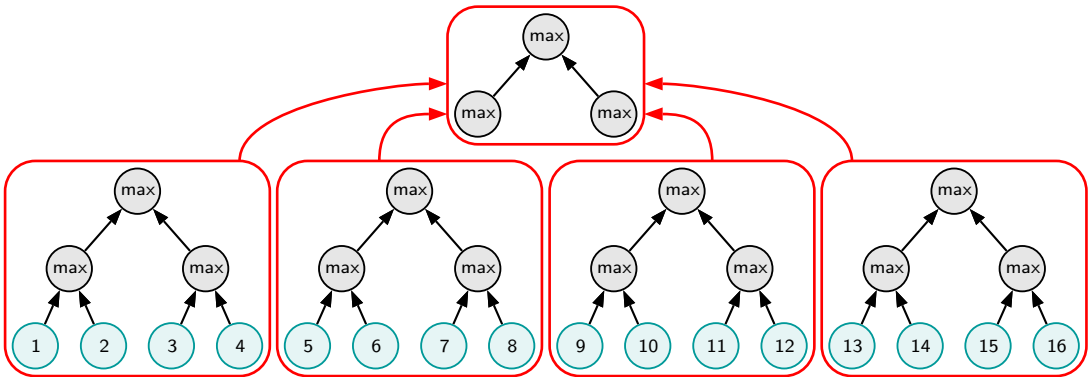


Abbildung 6.4: Separate Teilgraphen einer parallelen Maximumbestimmung.

Dieses Problem (wie auch alle anderen Probleme, bei denen ein Threadblock auf die Terminierung aller anderen Threadblöcke warten muss) lässt sich mit einem Zähler (sog. *Semaphore*) im Device Memory lösen. Jeder Block inkrementiert nach der Bestimmung des Maximums über alle Werte des Blocks diesen Zähler um eins. Sollte der Wert des Zählers anschließend der Anzahl aller Blöcke entsprechen, kann davon ausgegangen werden, dass der aktuelle Block der letzte ist und die abschließende Berechnung des Maximums über alle Blöcke durchgeführt werden kann.

U.a. zur Bestimmung des Maximums aller Elemente eines Threadblocks wurde die Routine `blockaction()` implementiert. Sie berechnet parallel wie in Kapitel 4 beschrieben das Maximum (für `action=1`) oder die Summe (für `action=0`) des übergebenen Arrays `double * temp` und speichert den Wert im ersten Element des Arrays.

Methodenparameter von `blockaction()`

`double * temp`	Zu bearbeitendes Datenarray (Shared Memory)
`int action`	Wahl der durchzuführenden Aktion: 0=Summierung, 1=Maximum bestimmen

```
90  __device__ void blockaction(double * temp, int action)
91  {
92      int thidx = (threadIdx.y << TSHIFTX) + threadIdx.x;
93      for(int i=TSHIFTY+TSHIFTX-1; i>=0; --i)
94      {
95          __syncthreads();
96
97          int shift = 1 << i;
98          if (thidx < shift)
99          {
100             if (action == 0)
```

```
101             {
102                 temp[thidx] += temp[thidx+shift];
103             } else
104             {
105                 if (temp[thidx+shift] > temp[thidx])
106                 {
107                     temp[thidx] = temp[thidx+shift];
108                 }
109             }
110         }
111     }
112     __syncthreads();
113 }
```

Quellcode 6.1: Berechnung eines Maximums bzw. einer Summe in einem Threadblock.

Die hinsichtlich aller Threadblöcke globale Berechnung des Maximums findet letztendlich in der Methode `globalaction()` statt.

Methodenparameter von `globalaction()`	
`double * field`	Array der Länge `gridDim.x*gridDim.y` (Device Memory)
`double * temp`	In diesem Array legt jeder Threadblock Daten ab, deren Maximum oder Summe über alle Blöcke hinweg bestimmt werden soll (Shared Memory)
`uint * semaphore`	Mit 0 initialisierte Variable zur Ermittlung des letzten aktiven Blocks (Device Memory)
`int action`	Wahl der durchzuführenden Aktion: 0=Summierung, 1=Maximum bestimmen

```
115 __device__ bool globalaction(double * field, double * temp,
116                              unsigned int * semaphore, int action)
117 {
118     __shared__ double val;          // the return value
119     __shared__ int blocksx;
120     __shared__ int blocksy;
121     __shared__ bool calcaction;
122 
123     int thidx = (threadIdx.y << TSHIFTX) + threadIdx.x;
124     int size = gridDim.x*gridDim.y;
125 
126     // do action for current block
127     blockaction(temp, action);
128 
129     if (threadIdx.x == 0 && threadIdx.y == 0)
130     {
131         // save value to device memory
132         field[blockIdx.y*gridDim.x + blockIdx.x] = temp[0];
133 
134         // wait until value is available to all Blocks
135         __threadfence();
136 
137         // check if this block is the last running one
138         int thisblock = atomicInc(semaphore, size-1);
139         calcaction = (thisblock == size-1);
140     }
141     __syncthreads();
142 
143     if (calcaction)
144     {
145         // It IS the last one, build sum of device memory array
146         if (threadIdx.x == 0 && threadIdx.y == 0)
147         {
148             // check how many iterations are needed to handle
149             // all blocks by 16x16 partial fields
150             blocksx = gridDim.x;
```

```
151
152             // is gridDim.x a multiple of TSIZEX?
153             if ((blocksx & (TSIZEX-1)) != 0)
154             {
155                 //no? then increase it to a multiple of TSIZEX
156                 blocksx  = (blocksx | (TSIZEX-1))+1;
157             }
158             blocksx >>= TSHIFTX;    // we just need the factor
159
160             blocksy = gridDim.y;
161             if ((blocksy & (TSIZEY-1)) != 0)
162             {
163                 blocksy  = (blocksy | (TSIZEY-1))+1;
164             }
165             blocksy >>= TSHIFTY;
166
167             val = 0.0;
168         }
169
170         __syncthreads();
171
172         for (int by = 0; by < blocksy; ++by)
173         {
174             for (int bx = 0; bx < blocksx; ++bx)
175             {
176                 // compute every partial field
177                 int idx_y = by*TSIZEY + threadIdx.y;
178                 int idx_x = bx*TSIZEX + threadIdx.x;
179
180                 if (idx_x < gridDim.x && idx_y < gridDim.y)
181                 {
182                     // read value of field to shared memory
183                     temp[thidx] = field[idx_y*gridDim.x + idx_x];
184                 } else
185                 {
186                     // if fieldsize is not a muliple of 16, fill
187                     // increase to a multiple of 16 by reading zeroes.
188                     temp[thidx] = 0.0;
189                 }
190
191                 // do action for current partial field
192                 blockaction(temp, action);
193
194                 // apply computed value to return value
195                 if (thidx == 0)
196                 {
197                     if (action == 0)
198                     {
199                         val += temp[0];
200                     } else
201                     {
202                         if (temp[0] > val)
203                         {
204                             val = temp[0];
205                         }
206                     }
207                 }
208
209                 __syncthreads();
210             }
211         }
212
213         // store return value to first element of "temp"
214         if (thidx == 0)
215         {
216             temp[0] = val;
217         }
218         __syncthreads();
```

```
219     }
220
221     // tell kernel, if current block is the last one
222     return calcaction;
223 }
```

Quellcode 6.2: Berechnung eines Maximums bzw. einer Summe global über alle Threadblöcke.

In der Methode `globalaction()` wird zuerst von jedem Threadblock das Maximum (bzw. die Summe) der Daten in `double * temp` des jeweiligen Threadblocks mit `blockaction()` berechnet und der Wert an entsprechender Stelle im Array `double * field` abgelegt. Ist der aktuelle Threadblock der letzte Block, wird erneut `blockaction()` aufgerufen, diesmal jedoch, um die Werte von `double * field` zu bearbeiten. Da `blockaction()` nur Arrays der Länge `TSIZEX*TSIZEY` bearbeiten kann, muss `blockaction()` ggf. mehrfach *nacheinander* (denn es ist nur noch ein Block aktiv) mit Teilen von `double * field` aufgerufen werden. Ist die Länge von `double * field` kein Vielfaches von `TSIZEX*TSIZEY`, wird das Feld entsprechend mit Nullen aufgefüllt.

Der Rückgabewert von `globalaction()` ist `false`, wenn der aktuelle Block nicht der letzte war. Ist der Rückgabewert `true`, war der aktuelle Block der letzte und das über sämtliche Daten bestimmte Maximum, bzw. deren Summe wurde in `temp[0]` abgelegt.

Mithilfe von `globalaction()` kann nun der Device Kernel `rho()` für die Abschätzung des Spektralradius implementiert werden.

Methodenparameter von `rho()`

`double * _y`	Aktueller Zustand $\hat{y}$ (Device Memory)
`double hx, hy`	Schrittweiten der zweidimensionalen Ortsdiskretisierung h_1 und h_2
`double * res`	Array der Länge `gridDim.x*gridDim.y` (Device Memory)
`uint * semaphore`	Mit 0 initialisierte Variable zur Ermittlung des letzten aktiven Blocks (Device Memory)

```
240 __global__ void rho( double * _y, double hx, double hy,
241                      double * res, unsigned int * semaphore )
242 {
243     __shared__ double result[TSIZEY][TSIZEX];
244
245     // Workingelement of the current thread
246     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
247                 (blockDim.x*blockIdx.x + threadIdx.x);
248     double sqrhx = 1.0/(hx*hx);
249     double sqrhy = 1.0/(hy*hy);
250
251     // Read the Element from device memory
252     double y = _y[idx];
253
254     // Compute gershgorin radius
255     result[threadIdx.y][threadIdx.x] =
256                   fabs(ADVX/hx + ADVY/hy - NU*2.0*(sqrhx + sqrhy) +
257                             MU*(1.0-3.0*y*y))
258                 + fabs(-ADVX/hx + NU*sqrhx)
259                 + fabs(-ADVY/hy + NU*sqrhy)
260                 + fabs(NU*sqrhx);
261                 + fabs(NU*sqrhy);
262
263     // compute global maximum
264     if (globalaction(res, &result[0][0], semaphore, 1))
265     {
266         // this is the last block, return value to host code
```

```
267         if (threadIdx.x== 0 && threadIdx.y == 0)
268         {
269             res[0] = result[0][0];
270         }
271     }
272 }
```

Quellcode 6.3: Device Kernel für die Abschätzung des Spektralradius im ROCK4–Algorithmus.

Im Kernel wird der Komponentenindex des Elements in der Ortsdiskretisierung anhand des Index des aktuellen Threads berechnet und das entsprechende Element lokal zwischengespeichert. Der Radius des Gerschgorin–Kreises, der dem Komponentenindex des Elements entspricht, lässt sich dank der regelmäßigen Struktur der Jacobimatrix einfach berechnen.

Der Radius wird an entsprechender Stelle im Shared Memory Array `double result[][]` gespeichert und anschließend `globalaction()` zur Bestimmung des Maximums aufgerufen. `double * res` wird dabei als Zwischenspeicher im Device Memory verwendet. Ist der aktuelle Block der letzte aktive Block, wird das Ergebnis in `result[0][0]` über `double * res` an den Host Code zurückgeliefert.

6.3.3 Die erste Auswertung der rechten Seite

Zu Beginn des Runge–Kutta–Schritts wird die rechte Seite der Funktion einmal ausgewertet. In der Implementierung erledigt dies der Kernel `calc_stepinit()`.

Methodenparameter von `calc_stepinit()`	
`double * _y`	Aktueller Zustand $\hat{y}$ zu Beginn des Runge–Kutta–Schritts (Device Memory)
`double * _fn`	Die Auswertung der rechten Seite mit diesem Zustand wird hier gespeichert (Device Memory)
`double hx, hy`	Schrittweiten der Ortsdiskretisierung h_1 und h_2

```
275 __global__ void calc_stepinit(double * _yn, double * _fn,
276                 double hx, double hy)
277 {
278     // Workingelement of the current thread
279     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
280                 (blockDim.x*blockIdx.x + threadIdx.x);
281     __shared__ double yn[TSIZEY+2][TSIZEX+2];
282 
283     readstate(_yn, yn, idx);                    // Read state to shared memory
284     _fn[idx] = eval_pde(yn, hx, hy);    // evaluate right handed side
285 }
```

Quellcode 6.4: Device Kernel für eine einfache Auswertung der rechten Seite im ROCK4–Algorithmus.

Die Methode `readstate()` liest den Speicherbereich, den der aktuelle Block bearbeiten muss, ins Shared Memory. Das zweidimensionale Array im Shared Memory ist in jeder Dimension um zwei Elemente größer, da die Ränder der angrenzenden Blöcke ebenfalls eingelesen werden, was in unterschiedlichen Warps geschieht, um Threadbranching zu vermeiden.

Methodenparameter von `readstate()`

`double * _y`	Einzulesendes Array (Device Memory)
`double y[][]`	Zwischenspeicher, in den ein Teil von _y eingelesen wird (Shared Memory)
`int idx`	Index des Elements im Ortsdiskretisierungsgitter, das dem aktuellen Thread zugeordnet ist

```
27 __device__ void readstate(double * _y, double y[TSIZEY+2][TSIZEX+2], int idx)
28 {
29
30     int local_xidx = threadIdx.x+1;
31     int local_yidx = threadIdx.y+1;
32     int width = blockDim.x*gridDim.x;
33
34     y[local_yidx][local_xidx] = _y[idx];
35
36     // read boundaries
37
38     // left boundary
39     if (threadIdx.y == 2)                        // second warp
40     {
41         if (blockIdx.x == 0)
42         {
43             y[local_xidx][0] = 0.0;              // most left block
44         } else
45         {
46             y[local_xidx][0] =
47                 _y[idx - 1 - threadIdx.x + (threadIdx.x - 2)*width ];
48         }
49     }
50     // right boundary
51     else if (threadIdx.y == 4)                   // third warp
52     {
53         if (blockIdx.x == gridDim.x-1)
54         {
55             y[local_xidx][blockDim.x+1] = 0.0;   // most right block
56         } else
57         {
58             y[local_xidx][blockDim.x+1] =
59                 _y[idx - threadIdx.x + TSIZEX + (threadIdx.x - 4)*width ];
60         }
61     }
62
63     // upper boundary
64     else if (threadIdx.y==0)                     // first warp
65     {
66         if (blockIdx.y == 0)
67         {
68             y[local_yidx-1][local_xidx] = 0.0;   // most upper block
69         } else
70         {
71             y[local_yidx-1][local_xidx] = _y[idx-width];
72         }
73     }
74
75     // bottom boundary
76     else if (threadIdx.y==blockDim.y-1)          // last warp
77     {
78         if (blockIdx.y == gridDim.y-1)
79         {
80             y[local_yidx+1][local_xidx] = 0.0;   // most bottom block
81         } else
82         {
83             y[local_yidx+1][local_xidx] = _y[idx+width];
84         }
85     }
86
```

```
87      __syncthreads();
88  }
```

Quellcode 6.5: Hilfsmethode zum Einlesen eines Bereichs ins Shared Memory im ROCK4–Algorithmus.

Mit `eval_pde()` wird das dem jeweiligen Thread entsprechende Element der rechten Seite ausgewertet. Die Berechnung entspricht einer Komponente von (5.2).

Methodenparameter von `eval_pde()`	
`double y[][]`	Teilbereich des Zustands $\hat{y}$, der vom aktuellen Threadblock bearbeitet wird (Shared Memory)
`double hx, hy`	Schrittweiten der Ortsdiskretisierung h_1 und h_2

```
226 __device__ double eval_pde(double y[TSIZEY+2][TSIZEX+2], double hx, double hy)
227 {
228     int idx_x = threadIdx.x+1;
229     int idx_y = threadIdx.y+1;
230
231     return  -ADVX*(y[idx_y][idx_x+1] - y[idx_y][idx_x])/hx
232             -ADVY*(y[idx_y+1][idx_x] - y[idx_y][idx_x])/hy
233
234         + NU*((y[idx_y][idx_x+1] - 2.0*y[idx_y][idx_x] + y[idx_y][idx_x-1])/(hx*hx)
235         + (y[idx_y+1][idx_x] - 2.0*y[idx_y][idx_x] + y[idx_y-1][idx_x])/(hy*hy))
236
237         + MU*y[idx_y][idx_x]*(y[idx_y][idx_x] + 1.0)*(1.0 - y[idx_y][idx_x]);
238 }
```

Quellcode 6.6: Auswerten der rechten Seite der diskretisierten Wärmeleitungsgleichung im ROCK4–Algorithmus.

6.3.4 Dreifachterm–Rekursion

Die Dreifachterm–Rekursion wird in zwei verschiedenen Kernels durchgeführt. Im ersten Kernel `calc_rfstepfirst()` werden die Iterierten g_0 und g_1 gemäß (5.3) berechnet.

Methodenparameter von `calc_rfstepfirst()`	
`double * _y`	Array zum Zwischenspeichern von g_1 (Device Memory)
`double * _yn`	Aktueller Zustand zu Beginn der Rekursion (Device Memory)
`double * _yjm1`	Array zum Zwischenspeichern von g_1, wenn $s-4>1$ (Device Memory)
`double * _yjm2`	Array zum Zwischenspeichern von g_0, wenn $s-4>1$ (Device Memory)
`double * _fn`	Bereits ausgewertete rechte Seite (Device Memory)
`int mdeg`	Gesamtanzahl der Rekursionsstufen $s-4$
`int mr`	Startindex der Rekursionskoeffizienten in `double recf[]`
`double h`	Aktuelle Zeitschrittweite $\hat{h}$

```
286
287 __global__ void calc_rfstepfirst(double * _y, double * _yn, double * _yjm1,
288                                  double * _yjm2, double * _fn, int mdeg, int mr, double h)
289 {
```

```
290     // Workingelement of the current thread
291     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
292                 (blockDim.x*blockIdx.x + threadIdx.x);
293     double yjm1,yjm2;
294
295     yjm2 = _yn[idx];                          // g_0
296     yjm1 = yjm2 + h*recf[mr] * _fn[idx];      // g_1
297
298     if (mdeg > 1)                 // store g_0 and g_1 to device memory, if
299     {                             // s-4 is greater than 1
300         _yjm1[idx] = yjm1;
301         _yjm2[idx] = yjm2;
302     }
303
304     _y[idx] = yjm1;               // store g_1
305 }
```

Quellcode 6.7: Device Kernel zur Berechnung von g_0 und g_1 der Dreifachterm–Rekursion im ROCK4–Algorithmus.

Alle weiteren Rekursionsstufen werden durch den Kernel `calc_rfstepmdeg()` berechnet.

Methodenparameter von `calc_rfstepmdeg()`

`double * _y`	Array zum Zwischenspeichern von g_i (Device Memory)
`double * _yjm1`	Array mit g_{i-1} (Device Memory)
`double * _yjm2`	Array mit g_{i-2} (Device Memory)
`int stage`	Zu berechnende Iterationsstufe i
`int mdeg`	Gesamtanzahl der Rekursionsstufen $s-4$
`int mr`	Startindex der Rekursionskoeffizienten in `double recf[]`
`double h`	Aktuelle Zeitschrittweite $\hat{h}$
`double hx, hy`	Schrittweiten der Ortsdiskretisierung h_1 und h_2

```
307 __global__ void calc_rfstepmdeg(double * _y, double * _yjm1, double * _yjm2,
308                 int stage, int mdeg, int mr, double h,
309                 double hx, double hy)
310 {
311     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
312                 (blockDim.x*blockIdx.x + threadIdx.x);
313
314     double temp1, temp2, temp3, y;
315     __shared__ double yjm1[TSIZEY+2][TSIZEX+2];
316
317     temp1 = h * recf[mr + ( ( stage-1 ) << 1 ) + 1];
318     temp3 = -recf[mr + ( ( stage-1 ) << 1 ) + 2];
319     temp2 = 1.0 - temp3;
320
321     readstate(_yjm1, yjm1, idx);     // read g_{i-1}
322     y = eval_pde(yjm1, hx, hy);      // compute f(g_{i-1})
323
324     __syncthreads();
325
326     // compute g_i
327     _y[idx] = temp1 * y + temp2*yjm1[threadIdx.y+1][threadIdx.x+1] +
328                 temp3 * _yjm2[idx];
329 }
```

Quellcode 6.8: Device Kernel zur Berechnung der Iterierten g_i für $i = 2, \ldots, s-4$ der Dreifachterm–Rekursion im ROCK4–Algorithmus.

6.3.5 Implementierung einer Stufe der Methode W

Wegen der kreuzenden Abhängigkeiten zwischen den unterschiedlichen Runge–Kutta–Stufen der Methode W (siehe Definition 5.8) ist für jede Stufe ein eigener Device Kernel nötig. Die vier Stufen werden demnach in den Kernels `calc_rfstep_s1()`, `calc_rfstep_s2()`, `calc_rfstep_s3()` und `calc_rfstep_s4()` implementiert.

In jedem dieser Kernels findet eine Auswertung der rechten Seite statt und zugleich die Berechnung des Arguments für die Auswertung im darauf folgenden Device Kernel.

Methodenparameter von `calc_rfstep_s2()`

`double * _y`	Array mit g_{s-4}, also mit der Lösung der Methode P (Device Memory)
`double * _yjm1`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm2`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm3`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm4`	Hilfsarray für Zwischenergebnisse (Device Memory)
`int mz`	Startindex der Tableaukoeffizienten $\hat{a}_{i,j}$ in `double fpa[]`
`double h`	Aktuelle Zeitschrittweite $\hat{h}$
`double hx, hy`	Schrittweiten der Ortsdiskretisierung h_1 und h_2

```
350 __global__ void calc_rfstep_s2(double * _y, double * _yjm1, double * _yjm2,
351                 double * _yjm3, double * _yjm4, int mz, double h,
352                 double hx, double hy)
353 {
354     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
355                 (blockDim.x*blockIdx.x + threadIdx.x);
356
357     __shared__ double yjm3[TSIZEY+2][TSIZEX+2];
358     double yjm2;
359
360     readstate(_yjm3, yjm3, idx);     // read argument from Device Memory
361     yjm2 = eval_pde(yjm3, hx, hy);   // compute RHS
362
363     _yjm2[idx] = yjm2;               // store result to Device Memory
364
365     // compute argument for RHS of the next kernel and store to Device Memory
366     _yjm4[idx] = _y[idx] + h*(fpa[mz + 50]*_yjm1[idx] + fpa[mz + 100]*yjm2);
367 }
```

Quellcode 6.9: Device Kernel zur Berechnung der zweiten Stufe der Methode W im ROCK4–Algorithmus.

Die Kernels `calc_rfstep_s1()`, `calc_rfstep_s3()` und `calc_rfstep_s4()` führen im Prinzip die gleichen Schritte aus wie `calc_rfstep_s2()`. Es werden jedoch die Hilfsarrays für die Zwischenergebnisse jeweils anders verwendet.

6.3.6 Das eingebettete Verfahren

Zur Schrittweitenberechnung muss die Methode $\bar{W}$ ausgewertet und ε gemäß (5.4) ermittelt werden. Die anschließende Summierung über alle Elemente wird mithilfe des gleichen Tricks wie in Kapitel 6.3.2 zu einem Kernel `calc_rfstep_err()` zusammengefasst.

Methodenparameter von `calc_rfstep_err()`

`double * _y`	Lösung der Methode WP (Device Memory)
`double * _yjm1`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm2`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm3`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _yjm4`	Hilfsarray für Zwischenergebnisse (Device Memory)
`double * _fnt`	Enthält anschließend die Funktionsauswertung von `double * _y` für den nächsten Runge–Kutta–Schritt (Device Memory)
`double rtol, atol`	Absolute und relative Toleranz für Fehlerberechnung
`int mz`	Startindex der Tableaukoeffizienten $\hat{b}_i$ und $\bar{b}_i$ in `double fpb[]` bzw. `double fpbe[]`
`double h`	Aktuelle Zeitschrittweite $\hat{h}$
`double hx, hy`	Schrittweiten der Ortsdiskretisierung h_1 und h_2
`double * err`	Hilfsarray der Länge `gridDim.x*gridDim.y` zur Fehlerberechnung (Device Memory)
`uint * semaphore`	Mit 0 initialisierte Variable zur Ermittlung des letzten aktiven Blocks (Device Memory)

```
407 __global__ void calc_rfstep_err(double * _y, double * _yjm1, double * _yjm2,
408                 double * _yjm3, double * _yjm4, double * _fnt, double rtol,
409                 double atol, int mz, double h, double hx, double hy,
410                 double * err, unsigned int * semaphore)
411 {
412
413     int idx = (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y + threadIdx.y) +
414                 (blockDim.x*blockIdx.x + threadIdx.x);
415     int size = gridDim.x*gridDim.y;
416
417     __shared__ double y[TSIZEY+2][TSIZEX+2];
418     double * sum = &y[0][0];     // use y[][] also as summation array
419     double fnt;
420     double sqr;
421
422     // Read solution of method W.
423     readstate(_y, y, idx);
424
425     // Compute RHS of solution. This will be k_5 of Method \bar{W}
426     fnt = eval_pde(y, hx, hy);
427     _fnt[idx] = fnt;
428
429     // Compute component difference of both solutions divided by
430     // component tolerance factor
431     sqr = h*(   (fpbe[mz] - fpb[mz]) * _yjm1[idx] +
432             (fpbe[mz + 50] - fpb[mz + 50]) * _yjm2[idx] +
433             (fpbe[mz + 100] - fpb[mz + 100]) * _yjm3[idx] +
434             (fpbe[mz + 150] - fpb[mz + 150]) * _yjm4[idx] +
435             fpbe[mz + 200] * fnt
436         ) / ( atol + fabs ( y[threadIdx.y+1][threadIdx.x+1] ) *rtol );
437
438     __syncthreads();
439
440     // square the component difference ...
441     sum[(threadIdx.y << TSHIFTX)+threadIdx.x] = sqr*sqr;
442
443     // ... and build sum of arrays of all blocks
444     if (globalaction(err, sum, semaphore, 0))
445     {
446         if (threadIdx.x== 0 && threadIdx.y == 0)
447         {
448             // the last block: divide by size and extract the root
449             err[0] = sqrt ( sum[0] / (double)(size*TSIZEX*TSIZEY) );
450         }
```

```
451     }
452 }
```

Quellcode 6.10: Device Kernel zur Berechnung der eingebetteten Methode $\bar{W}$ und des Fehlers ε im ROCK4–Algorithmus.

Ergänzend zu (5.4) wird im Kernel zusätzlich eine komponentenweise relative Toleranz berücksichtigt.

Bei der Bildung der Differenz wird die Lösung von Methode W erneut berechnet, obwohl sie durch das Array `double * _y` bereits zur Verfügung gestellt werden würde. Auf diese Weise wird jedoch der Ausgangszustand $P(\hat{h}, \hat{y})$ eliminiert:

$$\begin{aligned} & W(\hat{h}, P(\hat{h}, \hat{y})) - \bar{W}(\hat{h}, P(\hat{h}, \hat{y})) \\ = \; & P(\hat{h}, \hat{y}) + \hat{h} \sum_{i=1}^{4} \hat{b}_i \hat{k}_i - P(\hat{h}, \hat{y}) - \hat{h} \sum_{i=1}^{5} \bar{b}_i \hat{k}_i \\ = \; & \hat{h} \left(\sum_{i=1}^{4} \left(\hat{b}_i - \bar{b}_i \right) \hat{k}_i - \bar{b}_5 \hat{k}_5 \right) \\ = \; & \hat{h} \left(\sum_{i=1}^{4} \left(\hat{b}_i - \bar{b}_i \right) \hat{k}_i - \bar{b}_5 f \left(W(\hat{h}, P(\hat{h}, \hat{y})) \right) \right) \end{aligned}$$

Der Wert für $f\left(W(\hat{h}, P(\hat{h}, \hat{y}))\right)$ wurde bereits innerhalb des Kernels ausgewertet. Der Kernel benötigt dadurch nicht mehr den Wert von $g_{s-4} = P(\hat{h}, \hat{y})$, was den Speicherbedarf reduziert und einen Zugriff auf Device Memory innerhalb des Kernels einspart.

Der Wert von ε ist anschließend in `err[0]` abgelegt.

6.3.7 Feinabstimmung der Kernelgrößen

Nachdem alle Kernels fertig implementiert wurden, sollten sie, wie in Kapitel 4.3 beschrieben, auf maximale GPU–Ausnutzung hin überprüft werden. Hilfreich ist hierbei der Compilerparameter `--ptxas-options=-v`, der eine Ausgabe der benötigten Ressourcen aller Kernels bewirkt. In Tabelle 6.1 werden die Größen der einzelnen Kernels mit der jeweiligen GPU–Ausnutzung G aufgelistet.

Offensichtlich wird die GPU bei fast allen Kernels nur zur Hälfte ausgenutzt. Es sollte also versucht werden, durch Variation des benötigten Shared Memory, der Anzahl der Threads pro Block oder der benötigten Register die Ausnutzung zu verbessern. Der Kernel `calc_rfstepfirst()` wird hierbei nicht betrachtet, da er bereits unter maximaler GPU–Ausnutzung ausgeführt wird.

Das benötigte Shared Memory lässt sich in den Kernels nicht beliebig ändern, sondern ist hier im Allgemeinen mit `(SIZEX+1)*(SIZEY+1)` fest an die Blockgröße gekoppelt. Um einen Hinweis zu erhalten, wie die GPU–Ausnutzung optimiert werden kann, wird nun für eine Auswahl von unterschiedlichen Blockgrößen – es kommen hier aufgrund der Implementation nur Zweierpotenzen als Blockgrößen in Frage – und Registern die GPU–Ausnutzung berechnet. Die Ergebnisse sind Tabelle 6.2 aufgelistet.

Device Kernel	Register	Memory [Bytes] *Shared*	*Local*	Blockgröße	G
`rho()`	28	2105	0	16×16	0.5
`calc_stepinit()`	26	2624	0	16×16	0.5
`calc_rfstepfirst()`	9	56	0	16×16	1
`calc_rfstepmdeg()`	28	2656	0	16×16	0.5
`calc_rfstep_s1()`	26	2648	0	16×16	0.5
`calc_rfstep_s2()`	26	2664	0	16×16	0.5
`calc_rfstep_s3()`	26	2672	0	16×16	0.5
`calc_rfstep_s4()`	26	2672	0	16×16	0.5
`calc_rfstep_err()`	31	2721	0	16×16	0.5

Tabelle 6.1: Benötigte Ressourcen der einzelnen Kernels der ROCK4–Implementation.

Register \ **Blockgrößen**	32	64	128	256	512
20	0.25	0.5	0.75	0.75	0.5
21	0.25	0.5	0.625	0.5	0.5
22	0.25	0.5	0.625	0.5	0.5
23	0.25	0.5	0.625	0.5	0.5
24	0.25	0.5	0.625	0.5	0.5
25	0.25	0.5	0.5	0.5	0.5
26	0.25	0.5	0.5	0.5	0.5
27	0.25	0.5	0.5	0.5	0.5
28	0.25	0.5	0.5	0.5	0.5
29	0.25	0.5	0.5	0.5	0.5
30	0.25	0.5	0.5	0.5	0.5
31	0.25	0.5	0.5	0.5	0.5

Tabelle 6.2: GPU–Ausnutzung der Kernels des ROCK4–Algorithmus bei unterschiedlichen Register– und Blockgrößenkonfigurationen.

Gemäß der Tabelle ergibt sich eine geringe Steigerung der GPU–Ausnutzung, wenn die Größe der Threadblöcke von 256 auf 128 Threads reduziert werden würde. Zugleich müsste die Registerzahl mithilfe des Compilerparameters `-maxrregcount` auf 24 beschränkt werden. Eine Verringerung der Blockgröße ist jedoch nicht in Betracht zu ziehen, da dies die Anzahl der Zugriffe auf langsames Device Memory, die beim Einlesen der Bereichsränder in der Methode `readstate()` verursacht werden, stark vergrößern würde. Der Vorteil der geringfügig besseren GPU–Ausnutzung wäre somit hinfällig.

Um die Kernels auf Threadblöcken mit 256 Threads effizienter ausführen zu können, müsste die Registerzahl auf maximal 20 beschränkt werden. Diese drastische Beschränkung hat jedoch zur Folge, dass der Compiler Rechenergebnisse im langsamen Local Memory auslagert. Die Menge des allokierten Local Memory kann der Tabelle 6.3 entnommen werden.

Es werden also aufgrund der Beschränkung auf 20 Register fast die Hälfte aller lokalen Variablen von den vier Byte großen Registern ins Local Memory verlagert. Auch diese

Device Kernel	Register	Memory [Bytes] *Shared*	*Local*	Blockgröße	G
`rho()`	20	2105	20	16×16	0.75
`calc_stepinit()`	20	2624	64	16×16	0.75
`calc_rfstepfirst()`	9	56	0	16×16	1
`calc_rfstepmdeg()`	20	2656	72	16×16	0.75
`calc_rfstep_s1()`	20	2648	64	16×16	0.75
`calc_rfstep_s2()`	20	2664	64	16×16	0.75
`calc_rfstep_s3()`	20	2672	64	16×16	0.75
`calc_rfstep_s4()`	20	2672	64	16×16	0.75
`calc_rfstep_err()`	19	2721	88	16×16	0.75

Tabelle 6.3: Benötigte Ressourcen der einzelnen Kernels der ROCK4–Implementation mit Beschränkung auf 20 Register.

Möglichkeit würde demnach trotz besserer GPU–Ausnutzung die Kernellaufzeit verlängern und ist somit nicht sinnvoll[2].

Die Analyse der Kernelgröße hat folglich ergeben, dass die bisherige Konfiguration (keine Registerbeschränkung, 256 Threads pro Block) für die gegebene Problemstellung trotz lediglich 50% GPU–Ausnutzung optimal bezüglich der Laufzeit ist.

6.4 Vergleiche mit der CPU

Um die Ausführung auf der Grafikkarte mit einer Ausführung auf der CPU vergleichen zu können, wurden die Kernels äquivalent für die CPU programmiert und die Kernelaufrufe gegen Schleifen ausgetauscht, die den Blockgittern und Threadblöcken entsprechen. Zwar werden die CPU–Methoden nur auf einem Prozessorkern ausgeführt, die Implementation dieser Methoden ist jedoch für die CPU optimiert um möglichst realistische Laufzeitvergleiche durchführen zu können.

Beide Varianten berechnen die Lösung des Anfangswertproblems der diskretisierten Wärmeleitungsgleichung (5.2) zum Zeitpunkt $t = 1$ bzgl. des Startzustands $r(\cdot) \equiv 1$. Der Gitterabstand wird dabei auf $h_1 = h_2 = 10^{-2}$ festgelegt. Abbildung 6.5 zeigt das Resultat des ROCK4–Algorithmus, berechnet auf der Grafikkarte mit der Ortsdiskretisierung $N_1 = N_2 = 256$.

6.4.1 Vergleich der Laufzeiten

Es soll nun ermittelt werden, wie sich die Größe des Ortsdiskretisierungsgitters (und somit die Gesamtanzahl der Threadblöcke im Blockgitter) auf die Laufzeiten auswirkt. Es werden also $N_1 = N_2$ variiert, während alle anderen Parameter konstant gehalten werden. Interessant ist hierbei der *Speedup*, der als

$$\frac{\text{CPU–Laufzeit}}{\text{GPU–Laufzeit}}$$

[2]Die Nutzung von mehr als acht Bytes Local Memory als Registerersatz hatte in zahlreichen Tests mit der CUDA–Hardware trotz besserer GPU–Ausnutzung stets längere Laufzeiten ergeben.

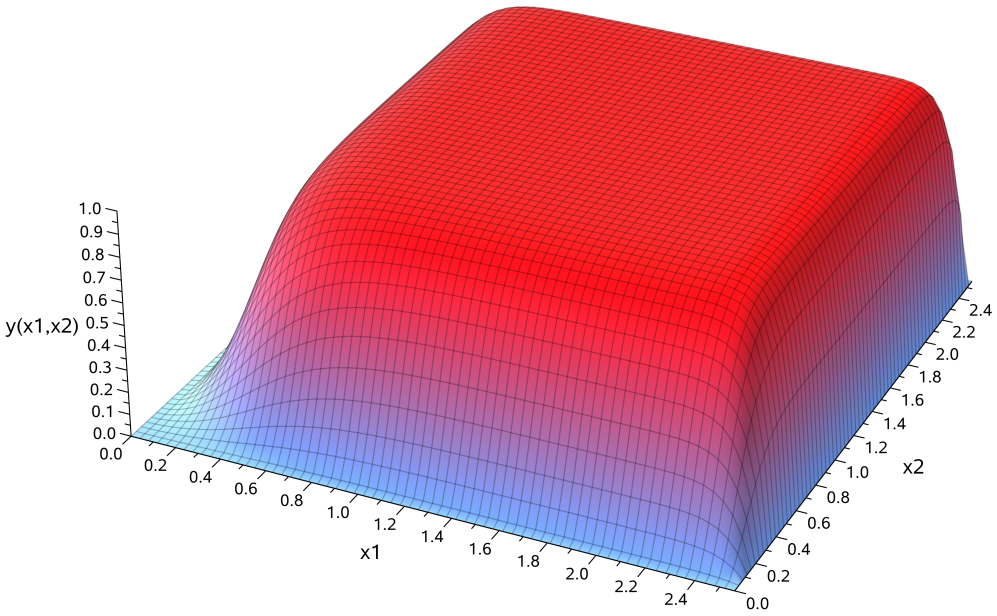


Abbildung 6.5: Lösung der Wärmeleitungsgleichung (5.1) zum Zeitpunkt $t = 1$, berechnet mithilfe der Grafikkarte.

definiert ist und den Faktor beschreibt, um wieviel schneller die Grafikkarten–Berechnung gegenüber der CPU–Berechnung ist. In Tabelle 6.4 sind die Messergebnisse aufgeführt und in Abbildung 6.6 entsprechend grafisch dargestellt.

N_i	CPU [s]	GPU [s]	Speedup	N_i	CPU [s]	GPU [s]	Speedup
16	0.051	0.137	×0.372	144	4.368	0.304	×14.349
32	0.217	0.132	×1.639	160	5.221	0.393	×13.298
48	0.485	0.138	×3.505	176	6.482	0.462	×14.031
64	0.893	0.134	×6.653	192	7.228	0.486	×14.887
80	1.266	0.138	×9.189	208	9.211	0.616	×14.944
96	1.896	0.213	×8.883	224	10.357	0.663	×15.633
112	2.647	0.216	×12.265	240	12.491	0.732	×17.054
128	3.398	0.288	×11.796	256	12.800	0.843	×15.187

Tabelle 6.4: Laufzeiten des ROCK4–Algorithmus für die Berechnung der Lösung zum Zeitpunkt $t = 1$ mit verschiedenen Gittergrößen.

Der maximale Speedup von ca. ×16 stellt sich ab einem Blockgitter mit $N_1 = N_2 = 256$ ein. Obwohl dies ein klarer Vorteil gegenüber der CPU–Implementation ist, wird die Erwartungshaltung etwas getrübt, wenn man bedenkt, dass dieser Vorteil auf der Grafikkarte von 30×8 Prozessoren erarbeitet wurde. Betrachtet man die Kernels des Algorithmus wird der Grund schnell klar: Nahezu alle Zwischenergebnisse werden im Device Memory ausgelagert.

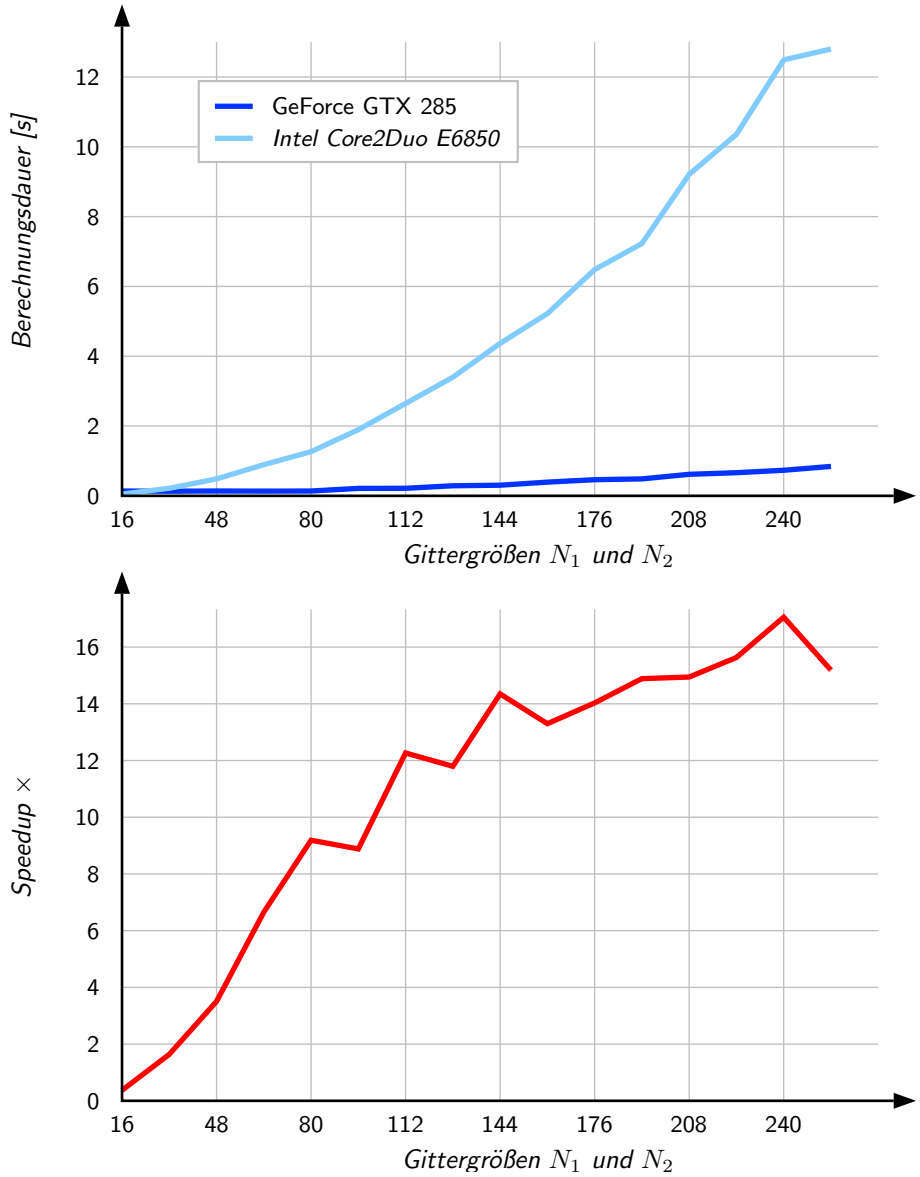


Abbildung 6.6: Laufzeiten des ROCK4–Algorithmus für die Berechnung der Lösung zum Zeitpunkt $t = 1$ mit verschiedenen Gittergrößen.

Die Kernels sind dagegen relativ kurz mit wenig "richtigen" Berechnungen pro Thread, so dass die GPU ihre Stärken nur schwer ausspielen kann und die meiste Zeit mit dem Kopieren von Speicherinhalten verbringt.

6.4.2 Numerische Effekte

Bei der Betrachtung der grafischen Darstellung des Speedup–Verlaufs, fällt das sehr unregelmäßige und nicht monotone Verhalten auf. Obwohl CPU– und GPU–Implementierung exakt die gleichen Rechenschritte durchführen, scheint das Verhältnis der Laufzeiten nicht nachvollziehbar zu variieren. Eine Erklärung hierfür liefert eine genauere Betrachtung der

Anzahl aller berechneten Runge–Kutta–Schritte in beiden Verfahren (siehe Abbildung 6.7).

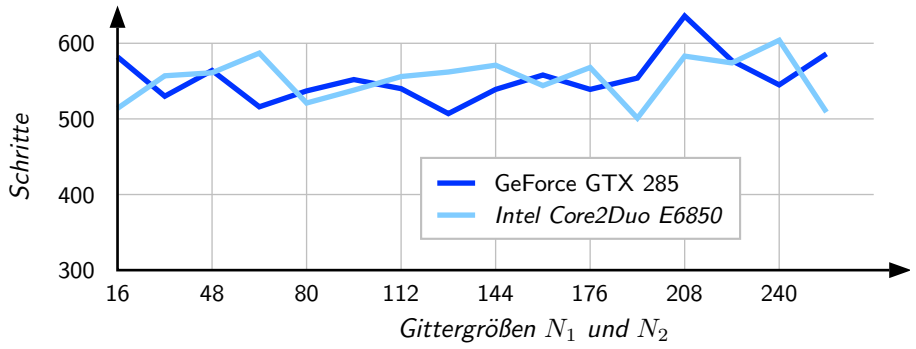


Abbildung 6.7: Anzahl der Runge–Kutta–Schritte des ROCK4–Algorithmus für die Berechnung der Lösung zum Zeitpunkt $t = 1$ mit verschiedenen Gittergrößen.

Entgegen der Erwartung, dass beide Implementationen exakt die gleichen Schritte berechnen, besitzen die berechneten Trajektorien beider Implementationen offensichtlich unterschiedliche Zeitdiskretisierungen. Dieser Effekt kann durch Eigenschaften der Gleitkommaarithmetik erklärt werden.

Während der Parallelisierung ändert sich die Reihenfolge, mit der die Summanden bei der Berechnung von ε addiert werden. In der Gleitkommaarithmetik existieren jedoch keine *echte* Assoziativität und Kommutativität, weshalb sich auch die daraufhin berechneten neuen Schrittweiten bei CPU– und GPU–Implementierung leicht unterscheiden.

Ebenso wirkt sich eine bestimmte Codeoptimierung des *nvcc*–Compilers auf die Berechnungen aus. Die GPU besitzt spezielle Prozessorbefehle, mit denen sich Multiplikationen und Additionen in einem Schritt zusammenfassen lassen, um Rechenzeit zu sparen. Die Ergebnisse dieser sog. *fused multiply–add* Instruktionen (FMAD) unterscheiden sich oft leicht von den konventionell berechneten Ergebnissen.

Da sich die Abweichungen in beiden Fällen im Rahmen der Double–Genauigkeit bewegen, sind sie normalerweise vernachlässigbar. Der ROCK4–Algorithmus ist jedoch trotz seiner Stabilisierung ein explizites Verfahren. Je steifer die zu lösende Differentialgleichung ist, desto stärker treten also numerische Effekte in den Vordergrund. Bei einer Diskretisierungsschrittweite von $h_1 = h_2 = 10^{-2}$ ist bereits ein Steifheitsgrad erreicht, bei dem leichte Abweichungen, die durch FMAD–Instruktionen und Parallelisierung generiert werden, deutlich werden. Bei einer gröberen Ortsdiskretisierung berechnen sowohl GPU– als auch die CPU–Implementation identische Trajektorien.

Teil III

Modellprädiktive Regelung

Kapitel 7

Modellprädiktive Regelung

Die Verwendung der Grafikkarte zur numerischen Berechnung der Lösungen von parabolischen partiellen Differentialgleichungen hat sich als sinnvoll erwiesen. Das ∞–dimensionale Problem kann näherungsweise durch ein quasi beliebig groß skalierbares endlichdimensionales Problem approximiert werden. Dieses Prinzip hatte bereits zu Beginn darauf schließen lassen, dass sich die CUDA–Hardware zur Berechnung der Lösung eignen würde.

In diesem Kapitel soll nun die GPU dazu verwendet werden, eine nichtlineare Feedbackregelung zu berechnen. Als Konzept zur Berechnung dieser Feedbackregelung wird die *Modellprädiktive Regelung* (*Model Predictive Control, MPC*) verwendet[1]. Anders als bei PDEs scheint hier anfangs nichts darauf hinzuweisen, dass sich passende Algorithmen effizient auf der CUDA–Hardware implementieren lassen könnten. Der relative große Aufwand des MPC–Konzepts bietet jedoch ausreichend Anlass dafür, genauere Analysen bzgl. der Möglichkeit einer Implementierung auf CUDA–Systemen durchzuführen.

7.1 Das Konzept MPC

Die Grundlage für die modellprädiktive Regelung bildet – dem Namen gemäß – eine mathematische Modellierung eines zu regelnden Systems. Der Zustand des Systems habe die Dimension n, die Kontrolle die Dimension m. Es wird angenommen, dass der Zustand des realen Systems für die Feedbackregelung vollständig vorliegt und daher ebenfalls die Dimension n hat.

Die Regelung findet in diskreten Zeitschritten der Länge T statt. Dabei wird zum k–ten Zeitpunkt der Zustand des Systems $x(k) \in \mathbb{R}^n$ gemessen und die MPC–Feedbackkontrolle $u(k) := F(x(k)) \in \mathbb{R}^m$ bis zum nächsten Zeitschritt $k+1$ angewandt. Hieraus ergibt sich eine stückweise konstante Steuerung. Das Modell wird durch eine Modellfunktion

$$\Phi : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n \tag{7.1}$$

beschrieben, wobei $\Phi(x(k), u(k))$ den Zustand des Modellsystems ausgehend vom Zustand $x(k)$ bei angewandter Steuerung $u(k)$, zum $(k+1)$–ten Zeitpunkt prädiziert.

[1] Eine detailierte Darstellung der Modellprädiktive Regelung kann in [13] gefunden werden.

7.1.1 Berechnen der Feedbackkontrolle

Definition 7.1
Sei die Horizontlänge N und $\mathbb{U} \subset \mathbb{R}^m$ gegeben.

- *Ein Vektor $u_N \in \mathbb{R}^{Nm}$ mit*

$$u_N(i) := \begin{pmatrix} u_{N,i\cdot m+1} \\ \vdots \\ u_{N,i\cdot m+m} \end{pmatrix} \in \mathbb{U}, \quad i = 0, \ldots, N-1$$

  heißt Kontrollsequenz der Länge N.

- *$\mathcal{U}_N$ ist die Menge aller Kontrollsequenzen der Länge N.*

- *$\mathbb{U}$ heißt Menge der zulässigen Kontrollen.*

Definition 7.2
Sei $u_N \in \mathcal{U}_N$, Φ eine Modellabbildung gemäß (7.1) und der Systemzustand $x \in \mathbb{R}^n$ gegeben. Die Funktion

$$x_{u_N} : \{0, \ldots, N\} \times \mathbb{R}^n \to \mathbb{R}^n$$

ist definiert durch

$$\begin{aligned} x_{u_N}(0, x) &:= x \\ x_{u_N}(i+1, x) &:= \Phi(x_{u_N}(i, x), u_N(i)), \quad i = 0, \ldots, N-1 \end{aligned}$$

$x_{u_N}(\cdot, x)$ heißt Prädiktion bezüglich des Startzustands x, der Kontrollsequenz u_N und der Modellabbildung Φ.

Definition 7.3
Gegeben seien die Funktionen $l : \mathbb{R}^n \times \mathbb{U} \to \mathbb{R}_0^+$ und $L : \mathbb{R}^n \to \mathbb{R}_0^+$. Die Funktion $J_N : \mathbb{R}^n \times \mathcal{U}_N \to \mathbb{R}_0^+$, definiert durch

$$J_N(x, u_N) := \sum_{i=0}^{N-1} l(x_{u_N}(i, x), u_N(i)) + L(x_{u_N}(N, x))$$

mit x_{u_N} gemäß Definition 7.2 heißt Zielfunktion mit Endkosten.

Definition 7.4
Gegeben sei eine Zielfunktion mit Endkosten J_N gemäß Definition 7.3 und $\mathbb{X} \subset \mathbb{R}^n$. Es existiere eine Kontrollsequenz $u_N \in \mathcal{U}_N$ s.d. gilt

$$x_{u_N}(i, x) \in \mathbb{X}, \quad i = 1, \ldots, N \tag{7.2}$$

Die MPC–Feedbackfunktion $F : \mathbb{X} \to \mathbb{U}$ *ist definiert als*

$$F(x) := \hat{u}_N(0)$$

mit

$$\hat{u}_N := \underset{u_N \in \mathcal{U}_N}{\operatorname{argmin}} J_N(x, u_N) \in \mathcal{U}_N \tag{7.3}$$

unter den Nebenbedingungen (7.2)*. Die Menge* $\mathbb{X}$ *heißt Menge der zulässigen Zustände.*

Die MPC–Feedbackregelung $u(k) = F(x(k))$ (ein *MPC–Schritt*) erfordert also die Bearbeitung der folgenden zwei Aufgaben:

1. Für den aktuellen Zustand $x(k)$ wird ein Optimalsteuerungsproblem auf dem endlichen Zeithorizont von $k \cdot T$ bis $(k+N) \cdot T$ gelöst, indem $x(k)$ mithilfe von Φ und unter Einwirkung einer Kontrollsequenz u_N N Zeitschritte in die Zukunft prädiziert und die dadurch entstehende diskrete Trajektorie über die Kostenfunktion J_N bewertet wird. Es wird davon ausgegangen, dass das Minimierungsproblem (7.3) mindestens eine zulässige Lösung besitzt. Die Lösung dieses Optimalsteuerungsproblems ist die optimale Kontrollsequenz $\hat{u}_N$.

2. $u(k) := \hat{u}_N(0)$ wird als Steuerung angewandt.

Auf Stabilität dieser Feedbackregelung und die damit zusammenhängenden Bedingungen an Startzustand, Horizontlänge und Zielfunktion wird in u.a. in [3, 6, 13] eingegangen. Abbildung 7.1 stellt drei aufeinanderfolgende MPC–Schritte in idealisierter Form dar.

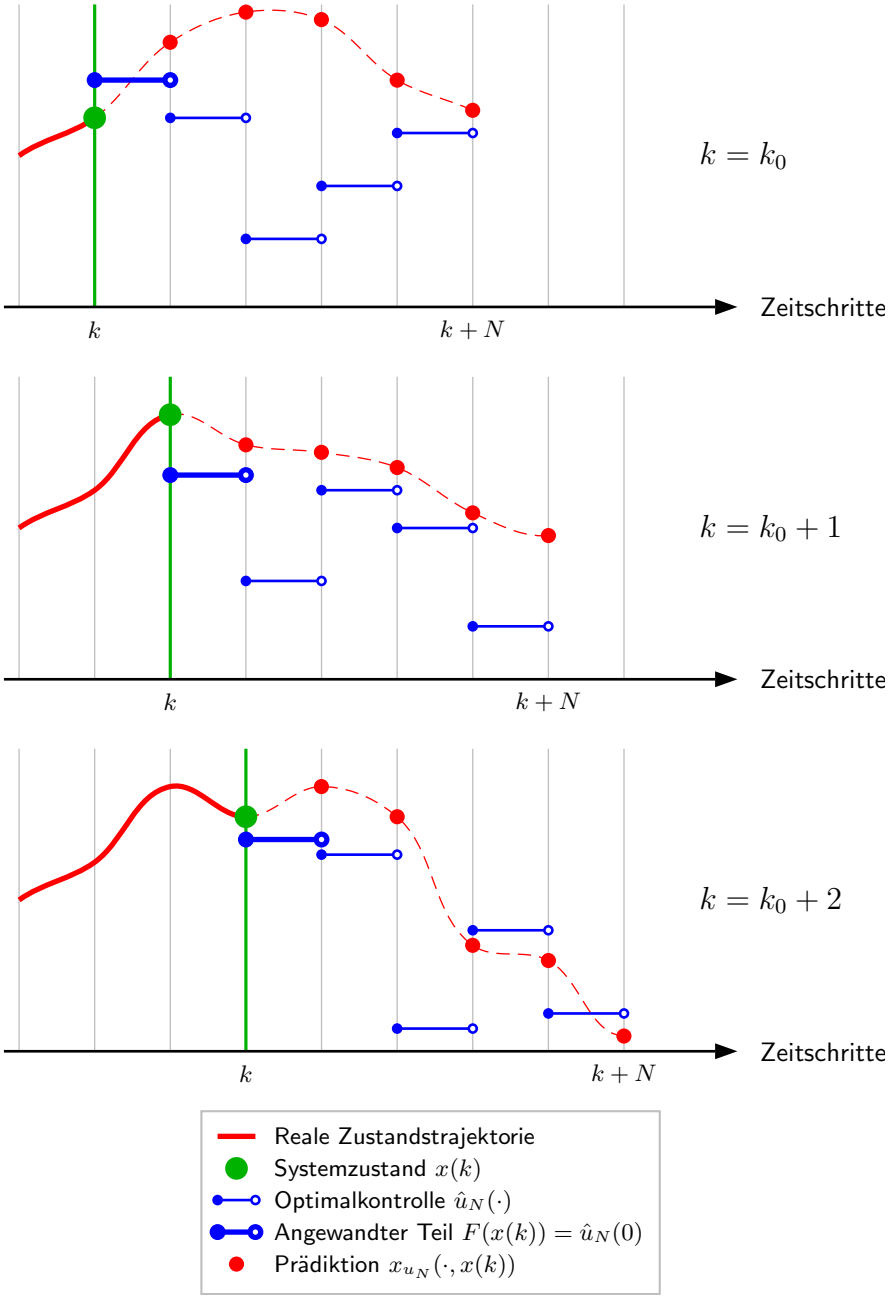


Abbildung 7.1: Exemplarischer Ablauf von drei MPC–Schritten.

7.1.2 Umsetzung in der Praxis

Der wesentliche Aufwand zur Bestimmung von $F(x(k))$ liegt in der Lösung des $(N \cdot m)$–dimensionalen Minimierungsproblems (7.3) unter den Nebenbedingungen (7.2). In der Praxis wird dafür in dieser Arbeit der nichtlineare Optimierer *Ipopt*[2] verwendet, dem die Menge

[2]Webseite der COIN–OR Initiative: https://projects.coin-or.org/Ipopt (12.05.2010).

der zulässigen Kontrollen $\mathbb{U}$ als implizite Restriktionen in Form von *boxed constraints*

$$u^u \leq \begin{pmatrix} u_{N,i\cdot m+1} \\ \vdots \\ u_{N,i\cdot m+m} \end{pmatrix} \leq u^o, \quad i = 0, \ldots, N-1$$

mit $u^u, u^o \in \mathbb{R}^m$ übergeben werden kann[3].

Dem Optimierer können zusätzlich explizite Restriktionen der Form

$$g(u_N) \geq 0_{\mathbb{R}^r}$$

mit $g : \mathbb{R}^{(N\cdot m)} \to \mathbb{R}^r$, $r > 0$ vorgeschrieben werden. Mithilfe dieser expliziten Restriktionen werden die Nebenbedingungen (7.2) formuliert:

Die bisher nicht näher beschriebene Menge der zulässigen Zustände $\mathbb{X}$ sei im Folgenden als

$$\mathbb{X} := \{x \in \mathbb{R}^n \mid x^u \leq x \leq x^o, \tilde{g}(x) \geq 0_{\mathbb{R}^{r'}}\}$$

mit $\tilde{g} : \mathbb{R}^n \to \mathbb{R}^{r'}$, $r' > 0$, $x^u, x^o \in \mathbb{R}^n$ festgelegt. Es besteht also die Möglichkeit für die Zustände, zusätzlich zu boxed constraints beliebige nichtlineare Beschränkungen über $\tilde{g}$ vorzuschreiben. Die Restriktion g für das nichtlineare Programm sei nun definiert als

$$g(u_N) := \begin{pmatrix} x_{u_N}(1, x(k)) - x^u \\ x^o - x_{u_N}(1, x(k)) \\ \tilde{g}\left(x_{u_N}(1, x(k))\right) \\ \vdots \\ x_{u_N}(N, x(k)) - x^u \\ x^o - x_{u_N}(N, x(k)) \\ \tilde{g}\left(x_{u_N}(N, x(k))\right) \end{pmatrix} \tag{7.4}$$

wodurch $r := N \cdot (2n + r')$ festgelegt ist. Es ist offensichtlich, dass gilt

$$x_{u_N}(1, x(k)), \ldots, x_{u_N}(N, x(k)) \in \mathbb{X} \;\Leftrightarrow\; g(u_N) \geq 0_{\mathbb{R}^r}$$

7.2 Abhängigkeitsgraph eines MPC-Schrittes

Das MPC–Konzept soll mithilfe eines Abhängigkeitsgraphen auf seine Parallelisierbarkeit hin überprüft werden. Für die Konstruktion eines Abhängigkeitsgraphen, genügt es, die Tasks zur der Lösung des Minimierungsproblems zu betrachten.

Der *Ipopt*–Optimierer ist ein *nichtlineares Innere–Punkte–Verfahren* und erwartet somit nach [7] zweimalige stetige Differenzierbarkeit bzgl. der Kontrollsequenz sowohl der Zielfunktion J_N als auch der Restriktionsfunktion g. Um dies zu gewährleisten wird zweimalige

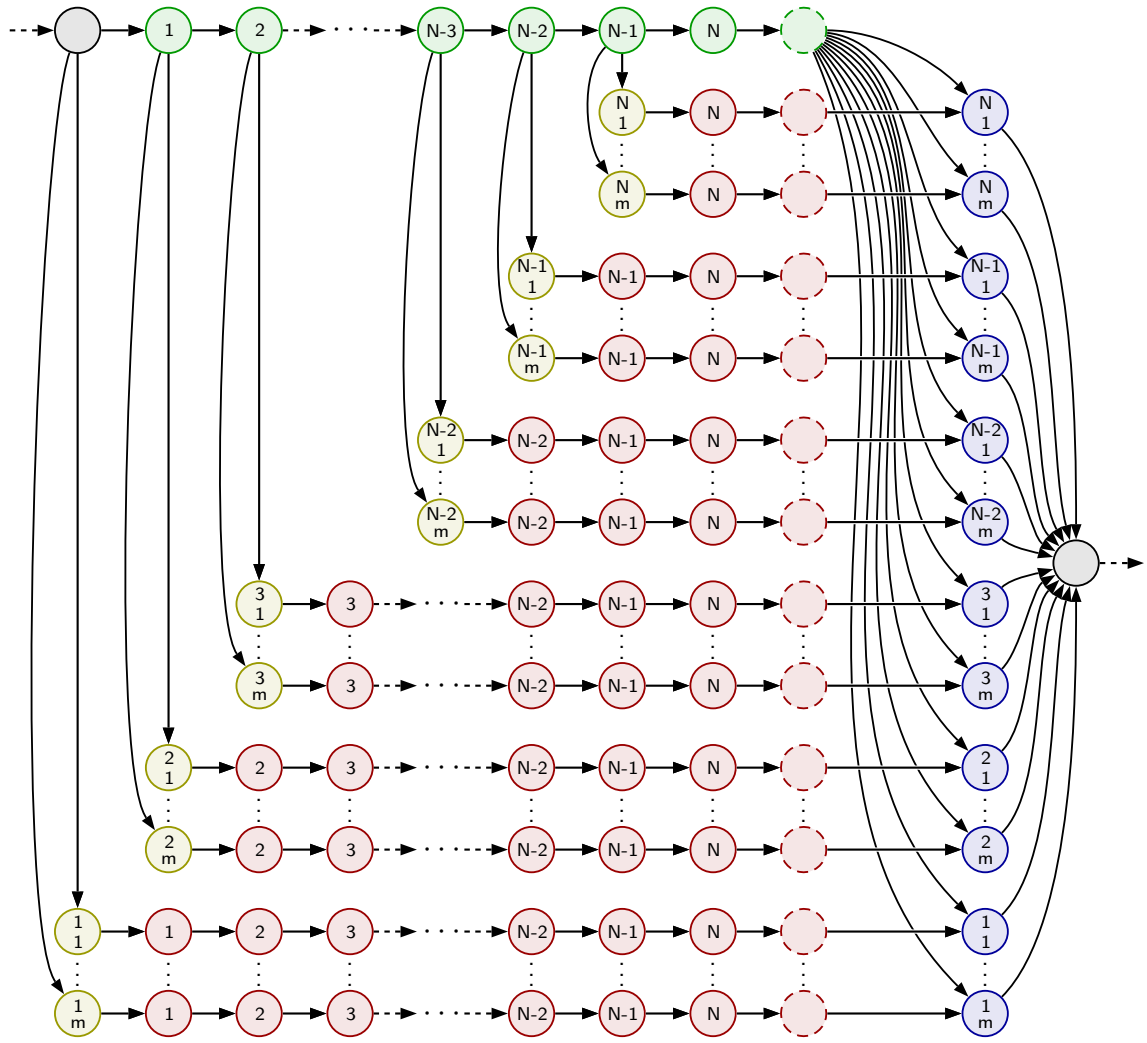
[3] "$\leq, \geq, <, >$" sind hier, wie auch im weiteren Verlauf dieser Arbeit, bei mehrdimensionalen Ausdrücken komponentenweise zu verstehen.

stetige Differenzierbarkeit gemäß den Definitionen 7.3 und 7.2 für die Funktionen $l(\cdot,\cdot)$, $L(\cdot)$ und $\Phi(\cdot,\cdot)$ vorausgesetzt. Da die Werte von J_N und g von Prädiktionen abhängen, deren Dynamik beliebig kompliziert werden kann, müssen Gradienten und Jacobimatrizen im Allgemeinen über numerisches Differenzieren mit Differenzierschrittweite $\delta_u > 0$ approximiert werden:

$$\begin{aligned}\frac{\partial}{\partial u_{N,i}} J_N(x(k), u_N) &\approx \frac{J_N(x(k), u_N + e_i \cdot \delta_u) - J_N(x(k), u_N)}{\delta_u} \in \mathbb{R}, \quad i = 1, \ldots, Nm \\ \frac{\partial}{\partial u_{N,i}} g(u_N) &\approx \frac{g(u_N + e_i \cdot \delta_u) - g(u_N)}{\delta_u} \in \mathbb{R}^r, \quad i = 1, \ldots, Nm\end{aligned}$$

$u_{N,i}$ ist hierbei die i–te Komponente der Kontrollsequenz $u_N \in \mathbb{R}^{Nm}$ und e_i der i–te Einheitsvektor im $\mathbb{R}^{Nm}$. Für die Berechnung von J_N, g und deren numerische Ableitungen ist es erforderlich, dass $Nm + 1$ Prädiktionen über jeweils N Zeitschritte ausgehend vom abgetasteten Zustand $x(k)$ stattfinden, wovon Nm Prädiktionen basierend auf gestörten Kontrollsequenzen $u_N + e_i \cdot \delta_u$ berechnet werden. Letztere Prädiktionen können jedoch ggf. bereits berechnete Werte der Trajektorie mit ungestörter Kontrollsequenz verwenden, da Variationen durch gestörte Kontrollsequenzen $u_N + e_i \cdot \delta_u$ ($i \in \{1, \ldots, Nm\}$) erst in Prädiktionen $x(j, x(k))$ mit $j \geq \lfloor (i-1)/m \rfloor + 1$ auftreten.

Insgesamt ergibt sich für die Berechnungen zwischen zwei Iterationen eines Minimierers ein Abhängigkeitsgraph, wie er in Abbildung 7.2 dargestellt wird.



Berechne Iteration des *Ipopt*–Optimierers (*Ipopt*–Bibliothek), initialisiere $\tilde{x}$

Berechne $l(\tilde{x}, u_N(i-1))$, $\tilde{x} := x_{u_N}(i, x(k))$ und $g(u_N)_{(i-1)(2n+r')+1}, \ldots, g(u_N)_{i(2n+r')}$

Berechne $L(x_{u_N}(N, x(k)))$

Lege lokale Kopie $u'_N := u_N + e_{(i-1)\cdot m+j} \cdot \delta_u$ an

Wie (i), die Prädiktion basiert jedoch auf einer lokalen Kopie u'_N

Wie (), die Prädiktion basiert jedoch auf einer lokalen Kopie u'_N

Berechne $\partial / \partial u_{N,(i-1)\cdot m+j}$ für $\Delta J_N(x(k), \cdot)$ und $Dg(\cdot)$ mit Differenzierschrittweite δ_u

Abbildung 7.2: Abhängigkeitsgraph der Auswertung von Zielfunktion, Restriktion und deren Ableitungen während der Ausführung des *Ipopt*–Optimierers.

7.3 Geeignete Problemstellungen

Die Parallelisierung des in Abbildung 7.2 illustrierten Ablaufs hängt stark vom zugrundeliegenden Modell ab. Der allgemein konstruierte Graph kann den Ablauf nur grob darstellen, da eine einzelne Prädiktion aus Platzgründen nur als einzelner Task abgebildet ist, dieser aber je nach Dimension der durch Φ modellierten Dynamik ebenfalls parallel ausgewertet werden kann. Eine detailierte Analyse kann demzufolge erst anhand eines konkret gegebenen Problems mit bekannten Funktionen l, L und Φ erfolgen.

Die Struktur des Graphen legt nahe, dass jeweils die Tasks einer Spalte parallel ausgeführt werden. Betrachtet man lediglich die in der Abbildung gelb, rot und blau dargestellten, mehrheitlich vorkommenden Tasks und setzt man voraus, dass eine Prädiktion mit einer Dynamik der Dimension n mit $p \in \mathbb{N}$ in n/p parallelen Threads erfolgen kann, ergibt sich insgesamt für die Berechnung der Prädiktionen eine Parallelisierungsbandbreite von $Nm \cdot n/p$ Threads. Über eine parallele Auswertung der Restriktionsfunktion und Zielfunktion kann erst dann eine klare Aussage gemacht werden, wenn ein konkretes Problem gegeben ist. Für eine allgemeine Aussage bezüglich der Eignung eines Problems wird daher lediglich die parallele Prädiktion betrachtet.

Die Laufzeitmessungen eines Testkernels auf Seite 52 haben gezeigt, dass für eine effiziente Nutzung der CUDA–Hardware eine Gesamtanzahl parallel auszuführender Blöcke von mehr als *Anzahl aktiver Blöcke* $\times$ *Anzahl Multiprozessoren* zwingende Voraussetzung ist. Für die Ausführung auf der Grafikkarte GTX285 müsste demnach bei beispielsweise vier aktiven Blöcken, einer Blockgröße von 64 Threads und einer numerisch realistischen Horizontgröße $N = 5$ mindestens gelten:

$$m \cdot \frac{n}{p} \geq \frac{4 \cdot 30 \cdot 64}{5} = 1536$$

Dieses einfache Beispiel zeigt deutlich, dass für die Implementierung der Funktions– und Restriktionsauswertung als Device Kernels zwischen zwei Minimierer–Schritten nur spezielle Probleme mit großen Zustands– und/oder Kontroll–Dimensionen in Frage kommen.

Kapitel 8

MPC eines Objektschwarms

Es soll nun mithilfe des MPC–Algorithmus aus Kapitel 7 ein *Schwarm* von M identischen Objekten gesteuert werden, jedes davon mit relativ geringer Zustandsdimension $\bar{n}$ und geringer Kontrolldimension $\bar{m}$. In der Gesamtheit ergibt sich damit ein Kontrollsystem mit $n := \bar{n}M$ und $m := \bar{m}M$. Bei genügend großer Objektanzahl M kann diese Problemstellung demnach das notwendige Kriterium zur effizienten Implementierung auf CUDA–Hardware aus Kapitel 7.3 erfüllen.

Die starke Nutzung von Device Memory in den Kernels des ROCK4–Algorithmus hatte einen unerwartet schwachen Geschwindigkeitsvorteil gegenüber der CPU zur Folge. Ziel dieses Kapitels ist es nun, Device Kernels für die Berechnung von J_N, g eines Objektschwarms und deren numerische Ableitungen zu konstruieren, in denen die Nutzung von Device Memory im Verhältnis zu tatsächlichen Rechenoperationen relativ gering ist und deren Laufzeit wieder mit einer äquivalent programmierten Variante für die CPU zu vergleichen.

Da die Rechenergebnisse der Grafikkarte für numerisches Differenzieren verwendet werden sollen, kommt hier ebenfalls (wie bei der Implementierung des ROCK4–Algorithmus) nur die Grafikkarte GTX285 mit Device Capabilities 1.3 als ausführende Hardware in Frage.

8.1 Aufstellung des Minimierungsproblems

Der Einfachheit halber wird in diesem Kapitel die sehr allgemeine Problemklasse der optimalen Schwarmsteuerung so weit eingeschränkt, dass folgendes gilt:

1. Alle Objekte besitzen die gleiche Modellfunktion $\bar{\Phi} : \mathbb{R}^{\bar{n}} \times \mathbb{R}^{\bar{m}} \to \mathbb{R}^{\bar{n}}$, siehe Seite 91.

2. Die Optimalsteuerungen der einzelnen Objekte werden jeweils durch die gleiche Zielfunktion $\bar{J}_N$ (gemäß Definition 7.3) vorgeschrieben, die unabhängig von den anderen Objekten ist.

3. Alle Objekte besitzen die gleichen boxed constraints $\bar{x}^u, \bar{x}^o \in \mathbb{R}^{\bar{n}}$ und $\bar{u}^u, \bar{u}^o \in \mathbb{R}^{\bar{m}}$.

4. Für alle zulässigen Zustände $x^{(i)}, x^{(j)}$ von Objekt i bzw. Objekt j mit $i \neq j$ gilt

$$\bar{g}_1(x^{(i)}, x^{(j)}) \geq 0_{\mathbb{R}_{r_1}}$$

bei gegebener Funktion $\bar{g}_1 : \mathbb{R}^{\bar{n}} \times \mathbb{R}^{\bar{n}} \to \mathbb{R}^{r_1}$ mit $r_1 > 0$. $\bar{g}_1$ ist symmetrisch, d.h. $\bar{g}_1(x^{(i)}, x^{(j)}) = \bar{g}_1(x^{(j)}, x^{(i)})$. Mithilfe von $\bar{g}_1$ lassen sich paarweise konkurrierende Bedingungen der Objekte, wie etwa der Minimalabstand zweier Objekte vorschreiben.

5. Für alle zulässigen Zustände $x^{(i)}$ von Objekt i gilt

$$\bar{g}_2(x^{(i)}) \geq 0_{\mathbb{R}_{r_2}}$$

bei gegebener Funktion $\bar{g}_2 : \mathbb{R}^{\bar{n}} \to \mathbb{R}^{r_2}$ mit $r_2 > 0$. Mithilfe von $\bar{g}_2$ lassen sich zusätzliche nichtlineare Zustandsrestriktionen beschreiben, die für alle Objekte gleichermaßen gelten.

Unter diesen Einschränkungen können nun J_N und g des Schwarmkontrollproblems definiert werden. Die Zustandsvektoren der einzelnen Objekte sowie deren Kontrollsequenzen werden hierfür im Zustand bzw. in der Kontrollsequenz des Schwarms angeordnet.

Definition 8.1 (Anordnung der Objektzustände im Schwarmzustand)
$x \in \mathbb{R}^n$ sei ein Zustand eines Objektschwarms mit $n := \bar{n}M$. Der Zustand $x^{(j)}$ eines Objekts j mit $j = 0, \ldots, M-1$ wird definiert durch

$$x^{(j)} := \begin{pmatrix} x_{j\bar{n}+1} \\ \vdots \\ x_{j\bar{n}+\bar{n}} \end{pmatrix} \in \mathbb{R}^{\bar{n}}$$

Definition 8.2 (Anordnung der Objektkontrollsequenzen in der Schwarmkontrollsequenz)
$u_N \in \mathcal{U}_N$ sei eine Kontrollsequenz gemäß Definition 7.1 des Schwarms mit Horizontlänge N und Kontrolldimension $m := \bar{m}M$. Die Kontrollsequenz $u_N^{(j)}$ eines Objekts j mit $j = 0, \ldots, M-1$ wird definiert durch

$$u_N^{(j)}(i) := \begin{pmatrix} u_N(i)_{j\bar{m}+1} \\ \vdots \\ u_N(i)_{j\bar{m}+\bar{m}} \end{pmatrix} \in \mathbb{R}^{\bar{m}}, \quad i = 0, \ldots, N-1$$

Als Zielfunktion der Schwarmkontrolle wird nun die Summe der Zielfunktionen aller Objektkontrollen gewählt:

$$\begin{aligned} J_N(x, u_N) &:= \sum_{j=0}^{M-1} \bar{J}_N\left(x^{(j)}, u_N^{(j)}\right) \\ &= \sum_{j=0}^{M-1} \left(\sum_{i=0}^{N-1} \bar{l}\left(\bar{x}_{u_N^{(j)}}\left(i, x^{(j)}\right), u_N^{(j)}(i)\right) + \bar{L}\left(x_{u_N^{(j)}}\left(N, x^{(j)}\right)\right) \right) \end{aligned} \tag{8.1}$$

$\bar{x}_{u_N^{(j)}}(\cdot, x^{(j)})$ sei hierbei eine Prädiktion des Zustands von Objekt j bezüglich der Modellfunktion $\bar{\Phi}$ mit der Kontrollsequenz $u_N^{(j)}$.

Aufgrund der Anordnung der Objektzustände $x^{(j)} \in \mathbb{R}^{\bar{n}}$ im Schwarmzustand $x \in \mathbb{R}^n$ können die Schranken für die boxed constraints des Schwarmzustands als

$$\begin{aligned} x^u &:= (\bar{x}^u, \ldots, \bar{x}^u) \in \mathbb{R}^n \\ x^o &:= (\bar{x}^o, \ldots, \bar{x}^o) \in \mathbb{R}^n \end{aligned} \tag{8.2}$$

definiert werden. Analog werden die boxed contraints für die Schwarmkontrolle durch

$$\begin{aligned} u^u &:= (\bar{u}^u, \ldots, \bar{u}^u) \in \mathbb{R}^m \\ u^o &:= (\bar{u}^o, \ldots, \bar{u}^o) \in \mathbb{R}^m \end{aligned} \tag{8.3}$$

festgelegt. Um letztendlich die Restriktionsfunktion g wie in (7.4) definieren zu können, müssen die Restriktionen $\bar{g}_1$ und $\bar{g}_2$ in $\tilde{g} : \mathbb{R}^n \to \mathbb{R}^{r'}$, $r' > 0$ zusammengefasst werden:

$$\begin{aligned} \tilde{g}(x) := (\ & \bar{g}_1(x^{(0)}, x^{(1)}), && \bar{g}_1(x^{(0)}, x^{(2)}), && \ldots, && \bar{g}_1(x^{(0)}, x^{(M-1)}), && \bar{g}_2(x^{(0)}), \\ & \bar{g}_1(x^{(1)}, x^{(2)}), && \ldots, && \bar{g}_1(x^{(1)}, x^{(M-1)}), && \bar{g}_2(x^{(1)}), \\ & && \ddots && \vdots && \vdots \\ & && && \bar{g}_1(x^{(M-2)}, x^{(M-1)}), && \bar{g}_2(x^{(M-2)}), \\ & && && && \bar{g}_2(x^{(M-1)})\ )^T \end{aligned} \tag{8.4}$$

mit

$$r' = \binom{M}{2} \cdot r_1 + M \cdot r_2 = \frac{M!}{(M-2)! \cdot 2} \cdot r_1 + M \cdot r_2 = M \left(\frac{M-1}{2} \cdot r_1 + r_2 \right)$$

8.2 Hardwarefreundliche Restriktionsfunktion

Die wie in (7.4) konstruierte Restriktionsfunktion g kann einen sehr hochdimensionalen Bildraum besitzen. Als Beispiel soll ein Schwarm von 64 Objekten gesteuert werden, wobei die einzelnen Objekte die Zustandsdimension $\bar{n} = 4$ und die Kontrolldimension $\bar{m} = 2$ besitzen. Die Horizontlänge sei auf $N = 5$ festgelegt. Für die Funktionen $\bar{g}_1$ und $\bar{g}_2$ gelte $r_1 = r_2 = 1$. Der Bildraum der Funktion g hat demnach die Dimension

$$\begin{aligned} r &= N \cdot (2n + r') = N \cdot \left(2M\bar{n} + M \left(\frac{M-1}{2} \cdot r_1 + r_2 \right) \right) \\ &= 5 \cdot \left(2 \cdot 64 \cdot 4 + 64 \left(\frac{63}{2} + 1 \right) \right) \\ &= 12960 \end{aligned}$$

Die Dimension des Kontrollvektors u_N ist

$$Nm = NM\bar{m} = 5 \cdot 64 \cdot 2 = 640$$

Die zu berechnende Jacobimatrix Dg hat somit $640 \times 12960 = 8294400$ Einträge, was bei einer einzigen Auswertung von Dg ebenso viele Schreibzugriffe auf Device Memory zur Folge hätte. Die meisten Komponenten von g werden nach (7.4) jeweils durch eine einfache Subtraktion berechnet. Dem Vorsatz zu Beginn dieses Kapitels, Speicherzugriffe auf Device Memory im Verhältnis zu Rechenoperationen sehr gering zu halten, scheint der momentane Ansatz zur Implementierung der Restriktionen offensichtlich nicht gerecht zu werden.

Dieses einfache Beispiel motiviert die Idee, mehrere Komponenten von g durch geeignete Umformung zu einer einzigen Komponente zusammenzufassen und so die Anzahl der Speicherzugriffe erheblich zu reduzieren. Hierfür werden zwei Hilfsfunktionen eingeführt.

Satz 8.3
Gegeben sind die parameterisierten Funktionen $\alpha_{a,b}, \beta_{a,b} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_0^-$ *mit* $a < b$ *durch*

$$\begin{aligned} \alpha_{a,b}(\xi) &:= \min\left\{ -\frac{64}{(b-a)^6}\left((\xi-a)(\xi-b)\right)^3,\ 0 \right\} \\ \beta_{a,b}(\xi) &:= \min\left\{ \frac{64}{(b-a)^6}\left((\xi-a)(\xi-b)\right)^3,\ 0 \right\} \end{aligned}$$

Dann gilt

1. $\alpha_{a,b}, \beta_{a,b}$ *sind zweimal stetig partiell differenzierbar.*

2.

$$\begin{gathered} \xi_i \in [a_i, b_i] \subset \mathbb{R}, \quad \xi_j \in (-\infty, a_j] \cup [b_j, \infty) \subset \mathbb{R} \\ \Updownarrow \\ \sum_i \alpha_{a_i,b_i}(\xi_i) + \sum_j \beta_{a_j,b_j}(\xi_j) = 0 \end{gathered}$$

Beweis: *Zeige 1.:* Da alle Funktionen in den Minimumausdrücken aus $\mathcal{C}^\infty$ sind, muss lediglich noch das Polynom und dessen erste Ableitung auf horizontale Tangenten in den Nullstellen a und b untersucht werden:

$$\begin{aligned} \beta'_{a,b}(\xi) &= \frac{64}{(b-a)^6} \cdot 3(\xi-b)^2(\xi-a)^2(2\xi-a-b) \\ \beta''_{a,b}(\xi) &= \frac{64}{(b-a)^6} \cdot 6(\xi-b)(\xi-a)(5\xi^2 - 5\xi(a+b) + a^2 + b^2 + 3ab) \end{aligned}$$

Sowohl die erste, als auch die zweite Ableitung von $\beta_{a,b}$ haben Nullstellen in a und b. Somit besitzen $\beta_{a,b}$ und $\beta'_{a,b}$ horizontale Tangenten in a und b, woraus die zweimalige stetige Differenzierbarkeit folgt. Analog kann das gleiche für $\alpha_{a,b}$ gezeigt werden.
Zeige 2.:

$$\begin{aligned} &\sum_i \alpha_{a_i,b_i}(\xi_i) + \sum_j \beta_{a_i,b_i}(\xi_j) = 0 \\ &\quad \Leftrightarrow\ \forall i,j:\ \left((\xi_i - a_i)(\xi_i - b_i)\right)^3 \le 0,\ \ \left((\xi_j - a_j)(\xi_j - b_j)\right)^3 \ge 0 \\ &\quad \Leftrightarrow\ \forall i,j:\ \xi_i \in [a_i, b_i], \quad \xi_j \in (-\infty, a_j] \cup [b_j, \infty) \end{aligned}$$

□

Mithilfe der Funktionen aus Satz 8.3 können die boxed constraints für Schwarmzustände mit lediglich einem eindimensionalen Term beschrieben werden:

$$\begin{pmatrix} x - x^u \\ x^o - x \end{pmatrix} \ge 0_{\mathbb{R}^{2n}} \ \Leftrightarrow\ \bar{x}_i^u < x_i^{(j)} < \bar{x}_i^o,\ \forall\, i = 1, \ldots, \bar{n},\ j = 0, \ldots, M-1$$

$$\Leftrightarrow \quad \sum_{i=1}^{\bar{n}} \sum_{j=0}^{M-1} \alpha_{\bar{x}_i^u, \bar{x}_i^o} \left(x_i^{(j)} \right) = 0$$

Die Restriktionen für weitere prädizierte Schwarmzustände $x_{u_N}(i, x(k))$ können mit zusätzlichen Additionen hinzugefügt werden.

Der Skalierungsfaktor $64/(b-a)^6$ in $\alpha_{a,b}$ sorgt dafür, dass die Restriktionen unabhängig von den jeweiligen Intervalllängen $b-a$ gleich gewichtet sind, was bei sehr langen Summen verhindert, dass einzelne Restriktionen mit kleinen Intervallen aufgrund der Ungenaukgkeit der Gleitkommaarithmetik vernachlässigt werden.

Sollte es die Problemstellung der optimalen Schwarmkontrolle zulassen, dass auch $\tilde{g}$ mit den Hilfsfunktionen aus Satz 8.3 ausgedrückt werden kann, dann würden sich letztendlich sämtliche Restriktionen in einer einzigen Komponente beschreiben lassen. Die Anzahl der Einträge der Jakobimatrix $D_{u_N} g$ würde beim obigen Beispiel von $640 \times 12960 = 8294400$ auf $640 \times 1 = 640$ reduziert und die Anzahl der Zugriffe auf Device Memory um den Faktor $\times 12960$ verringert werden.

8.3 Kernelkonzept eines konkreten Beispiels

Für die Umsetzung des Algorithmus auf der GPU seien die folgenden Schwarmeigenschaften gegeben:

$$\begin{aligned} \bar{n} &:= 4 \\ \bar{m} &:= 2 \\ \bar{\Phi}(\bar{x}, \bar{u}) &:= \begin{pmatrix} \bar{x}_1 + T\bar{x}_2 + \frac{1}{2}T^2\bar{u}_1 \\ \bar{x}_2 + T\bar{u}_1 \\ \bar{x}_3 + T\bar{x}_4 + \frac{1}{2}T^2\bar{u}_2 \\ \bar{x}_4 + T\bar{u}_2 \end{pmatrix} \end{aligned}$$

Es handelt sich hierbei um eine Gruppe von Objekten, die jeweils in der zweidimensionalen Ebene in jede Richtung beschleunigt werden können. Jedes Objekt des Schwarms soll zu einem Referenzpunkt $\bar{x}^{(ref)} \in \mathbb{R}^{\bar{n}}$ gesteuert werden, wobei die Ortskomponenten $\bar{x}_1^{(ref)}$ und $\bar{x}_3^{(ref)}$ von höherer Priorität sind. Die Anzahl der Objekte M soll aus $\{32, 64, 128, 256\}$ gewählt werden können, der Kontrollhorizont N und die Intervalllänge T der Zeitdiskretisierung ist variabel.

Als Kontrollrestriktionen werden $\bar{u}_1^u = \bar{u}_2^u = -12$ und $\bar{u}_1^o = \bar{u}_2^o = 12$ gewählt. Die Objektzustände sollen nicht durch boxed constraints beschränkt sein; stattdessen gelten die Bedingungen:

- Für zwei Objekte i und j $(i \neq j)$ gilt der ortsmäßige Mindestabstand

$$\left\| \begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} - \begin{pmatrix} x_1^{(j)} \\ x_3^{(j)} \end{pmatrix} \right\| \geq d$$

- Die Maximalgeschwindigkeit von Objekt i ist betragsmäßig beschränkt durch

$$\left\| \begin{pmatrix} x_2^{(i)} \\ x_4^{(i)} \end{pmatrix} \right\| \leq v_{max}$$

- Für den Ort von Objekt i gilt

$$\begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} \notin B_{0.3} \begin{pmatrix} 1.4 \\ 0.4 \end{pmatrix} \cup B_{0.3} \begin{pmatrix} 1.4 \\ -0.4 \end{pmatrix} \cup B_{0.3} \begin{pmatrix} 2.1 \\ 0 \end{pmatrix}$$

mit

$$B_r(\tilde{x}) := \{\tilde{x} + \lambda \in \mathbb{R}^2 \mid \|\lambda\| \leq r\}$$

8.3.1 Die Zielfunktion eines Schwarmobjekts

Die Zielfunktion eines einzelnen Schwarmobjekts wird gemäß (8.1) in Form einer gewichteten Summe von Abstandsquadraten durch

$$\begin{aligned} \bar{l}(\bar{x}) &:= (\bar{x}_1 - \bar{x}_1^{(ref)})^2 + (\bar{x}_3 - \bar{x}_3^{(ref)})^2 + \\ &\qquad \frac{1}{50}\left((\bar{x}_2 - \bar{x}_2^{(ref)})^2 + (\bar{x}_4 - \bar{x}_4^{(ref)})^2\right) \\ \bar{L}(\bar{x}) &:= 20 \cdot \bar{l}(\bar{x}) \end{aligned}$$

definiert, wobei $\bar{l}$ hier im Gegensatz zur allgemeinen Definition nur von $\bar{x} \in \mathbb{R}^{\bar{n}}$ abhängt.

8.3.2 Bestimmen der Restriktionsfunktion

Die Normen der drei geforderten Beschränkungen sollen mit den Funktionen aus Satz 8.3 in eine passende Restriktionsfunktion übertragen werden. Der folgende Satz gewährleistet, dass die zweimalige stetige Differenzierbarkeit der Restriktionsfunktion trotz Verwendung der Normen unter gewissen Voraussetzungen erhalten bleibt.

Satz 8.4
Seien $\alpha_{-a,a}$ und $\beta_{-a,a}$ mit $a > 0$ wie in Satz 8.3 gegeben. Die Funktion $\gamma : \mathbb{R} \to \mathbb{R}$ habe die folgenden Eigenschaften:

- $\gamma(0) = 0$.
- *γ ist stetig auf $\mathbb{R}$.*
- *γ ist zweimal stetig partiell differenzierbar auf $\mathbb{R} \setminus \{0\}$.*
- $-\gamma'(\xi) = \gamma'(-\xi)$, $\forall\, \xi \in \mathbb{R} \setminus \{0\}$.
- $|\lim_{\xi \to 0} \gamma'(\xi)| < \infty$ *und* $|\lim_{\xi \to 0} \gamma''(\xi)| < \infty$.

Dann sind $\alpha_{-a,a} \circ \gamma$ und $\beta_{-a,a} \circ \gamma$ auf ganz $\mathbb{R}$ zweimal stetig differenzierbar.

Beweis: $\beta_{-a,a} \circ \gamma$ ist stetig aufgrund der Stetigkeit von $\beta_{-a,a}$ und γ. Da $\beta_{-a,a}$ und γ auf $\mathbb{R} \setminus \{0\}$ zweimal stetig differenzierbar sind, gilt dies wegen der Kettenregel auch für $\beta_{-a,a} \circ \gamma$. Es bleibt somit lediglich noch die zweimalige stetige Differenzierbarkeit von $(\beta_{-a,a} \circ \gamma)(\xi)$ im Punkt $\xi = 0$ zu zeigen.

Wegen der Stetigkeit von γ existiert ein $\delta > 0$, so dass für alle ξ mit $|\xi| < \delta$ die Bedingung $|\gamma(\xi)| < a$ erfüllt ist. Es gilt also für $|\xi| < \delta$

$$(\beta_{-a,a} \circ \gamma)(\xi) = \frac{1}{a^6} \left((\gamma(\xi) + a)(\gamma(\xi) - a)\right)^3$$

und wegen $\gamma(0) = 0$, $-\gamma'(\xi) = \gamma'(-\xi)$ sowie der Beschränktheit von $\gamma'(\cdot)$ und $\gamma''(\cdot)$

$$\begin{aligned}
\lim_{\xi \to +0} (\beta_{-a,a} \circ \gamma)'(\xi) &= \lim_{\xi \to +0} -\frac{6}{a^6} \gamma(\xi)(a - \gamma(\xi))^2 (a + \gamma(\xi))^2 \gamma'(\xi) \\
&= 0 = \lim_{\xi \to -0} (\beta_{-a,a} \circ \gamma)'(\xi)
\end{aligned}$$

$$\begin{aligned}
\lim_{\xi \to +0} (\beta_{-a,a} \circ \gamma)''(\xi) &= \lim_{\xi \to +0} -\frac{6}{a^6} \underbrace{(a - \gamma(\xi))}_{\to a} \underbrace{(a + \gamma(\xi))}_{\to a} \underbrace{(a^2 \gamma''(\xi)\gamma(\xi)}_{\to 0} + a^2 \gamma'(\xi)^2 - \\
&\qquad \underbrace{\gamma''(\xi)\gamma(\xi)^3}_{\to 0} - \underbrace{5\gamma(\xi)^2 \gamma'(\xi)^2)}_{\to 0} \\
&= \lim_{\xi \to +0} -\frac{6}{a^2} \gamma'(\xi)^2 \\
&= \lim_{\xi \to +0} -\frac{6}{a^2} (-\gamma'(-\xi))^2 = \lim_{\xi \to +0} -\frac{6}{a^2} \gamma'(-\xi)^2 \\
&= \lim_{\xi \to -0} (\beta_{-a,a} \circ \gamma)''(\xi)
\end{aligned}$$

woraus die zweimalige stetige Differenzierbarkeit von $\beta_{-a,a} \circ \gamma$ folgt.

$\alpha_{-a,a} \circ \gamma$ ist wie auch $\beta_{-a,a} \circ \gamma$ auf $\mathbb{R} \setminus \{0\}$ zweimal stetig differenzierbar. Da $\alpha_{-a,a}(\gamma(\xi)) \equiv 0$ für $|\xi| < \delta$ ist, folgt hier sofort die zweimalige stetige Differenzierbarkeit auch für $\xi = 0$.

□

Normen erfüllen die Bedingungen für γ in Satz 8.4. Sofern das Beschränkungsintervall $[a, b]$ symmetrisch zum Ursprung ist, können Normen also als Argumente in $\alpha_{a,b}$ und $\beta_{a,b}$ eingesetzt werden, ohne dabei die zweimalige stetige Differenzierbarkeit der Restriktionsfunktion zu verletzen. Somit lässt sich die Restriktionsfunktion $g : \mathbb{R}^{NM\bar{m}} \to \mathbb{R}_0^-$ definieren durch

$$\begin{aligned}
\bar{g}_1(x^{(i)}, x^{(j)}) &:= \beta_{-d,d} \left(\left\| \begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} - \begin{pmatrix} x_1^{(j)} \\ x_3^{(j)} \end{pmatrix} \right\| \right) \\
\bar{g}_2(x^{(i)}) &:= \alpha_{-v_{max}, v_{max}} \left(\left\| \begin{pmatrix} x_2^{(i)} \\ x_4^{(i)} \end{pmatrix} \right\| \right) + \beta_{-0.3,0.3} \left(\left\| \begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} - \begin{pmatrix} 1.4 \\ 0.4 \end{pmatrix} \right\| \right) \\
&\quad + \beta_{-0.3,0.3} \left(\left\| \begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} - \begin{pmatrix} 1.4 \\ -0.4 \end{pmatrix} \right\| \right) + \beta_{-0.3,0.3} \left(\left\| \begin{pmatrix} x_1^{(i)} \\ x_3^{(i)} \end{pmatrix} - \begin{pmatrix} 2.1 \\ 0.0 \end{pmatrix} \right\| \right)
\end{aligned}$$

$$\begin{aligned}\tilde{g}(x) &:= \sum_{i=0}^{M-2}\sum_{j=i+1}^{M-1}\bar{g}_1\left(x^{(i)},x^{(j)}\right)+\sum_{i=0}^{M-1}\bar{g}_2\left(x^{(i)}\right)\\ g(u_N) &:= \sum_{i=1}^{N}\tilde{g}\left(x_{u_N}(i,x(k))\right)\end{aligned}$$

mit $\bar{g}_1 : \mathbb{R}^{\bar{n}} \times \mathbb{R}^{\bar{n}} \to \mathbb{R}_0^-$, $\bar{g}_2 : \mathbb{R}^{\bar{n}} \to \mathbb{R}_0^-$ und $\tilde{g} : \mathbb{R}^{M\bar{n}} \to \mathbb{R}_0^-$.

8.3.3 Einteilung in Threadblöcke

Als Motivation für die nähere Betrachtung der Optimalsteuerung eines Schwarms diente in Kapitel 7.3 die Feststellung, dass

$$Nm \cdot \frac{n}{p}$$

Threads für die Berechnung der Prädiktionen in den partiellen Ableitungen parallel arbeiten können. Legt man fest, dass alle Komponenten der Modellfunktion $\bar{\Phi}$ in einem einzigen Thread berechnet werden ($p := \bar{n}$), ergibt sich eine Gesamtanzahl von $NM\bar{m} \cdot M\bar{n}/\bar{n} = NM^2\bar{m}$ Threads.

Die Problemvorgabe $M \in \{32, 64, 128, 256\}$ legt nahe, dass M als Threadblockgröße gewählt wird. Somit lässt sich der Abhängigkeitsgraph aus Abbildung 7.2 in separate Teilgraphen einteilen, wie es in Abbildung 8.1 dargestellt wird. Die Einteilung verdeutlicht die Notwendigkeit, zwei verschiedene Kernels nacheinander auszuführen.

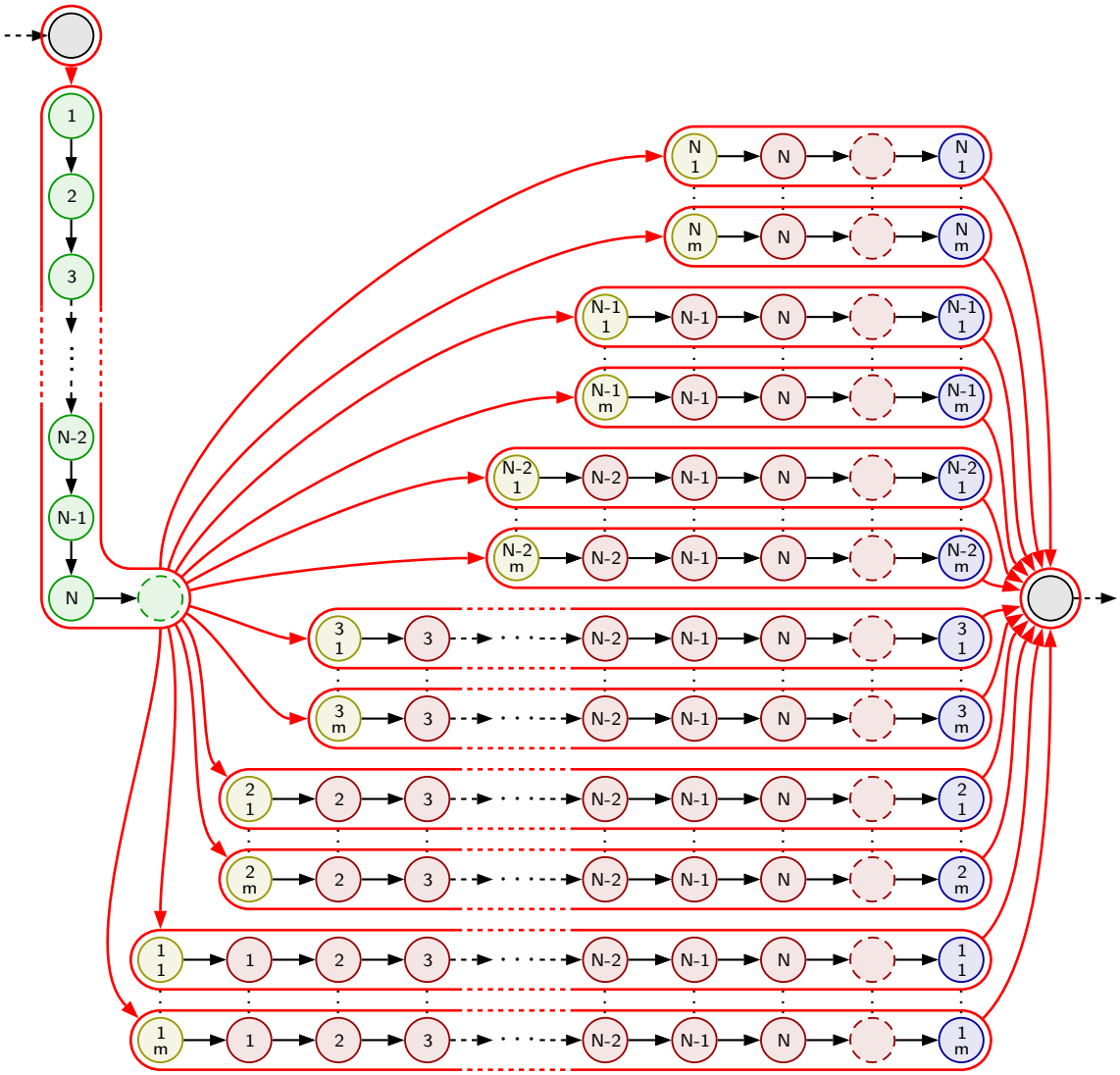


Abbildung 8.1: Separierter Abhängigkeitsgraph der Auswertung von Zielfunktion, Restriktion und deren Ableitungen während der Ausführung eines nichtlinearen Programms.

Die Ausführung des ersten Kernels geschieht als einzelner Block. In Prädiktionstask[1] i werden die folgenden drei Aufgaben erledigt ($j = 0, \ldots, M-1$):

1. Thread j berechnet

$$\bar{l}\left(\bar{x}_{u_N^{(j)}}\left(i-1, x^{(j)}(k)\right)\right)$$

und anschließend

$$\bar{x}_{u_N^{(j)}}\left(i, x^{(j)}(k)\right) \qquad \text{sowie} \qquad \bar{g}_2\left(\bar{x}_{u_N^{(j)}}\left(i, x^{(j)}(k)\right)\right)$$

2. Alle M Threads berechnen gemeinsam, wie in Abbildung 8.2 schematisiert,

$$\bar{g}_1\left(\bar{x}_{u_N^{(j_1)}}\left(i, x^{(j_1)}(k)\right), \bar{x}_{u_N^{(j_2)}}\left(i, x^{(j_2)}(k)\right)\right), \quad j_1, j_2 \in \{0, \ldots, M-1\}, j_1 < j_2 \tag{8.5}$$

3. Der prädizierte Schwarmzustand des aktuellen Horizontschritts

$$x_{u_N}\left(i, x(k)\right)$$

und die Zwischenergebnisse

$$\sum_{l=1}^{i} l\left(x_{u_N}\left(l, x(k)\right)\right) \qquad \text{und} \qquad \sum_{l=1}^{i} \tilde{g}\left(x_{u_N}\left(l, x(k)\right)\right)$$

werden zur Verwendung durch den zweiten Kernel abgespeichert.

Nachdem alle Prädiktionstasks abgearbeitet wurden, wird von Thread j der Endkostenterm

$$\bar{L}\left(\bar{x}_{u_N^{(j)}}\left(i, x^{(j)}(k)\right)\right)$$

berechnet und anschließend die ausgewerteten Summanden der Zielfunktion und der Restriktionsfunktion aller Threads addiert.

Die Arbeitsschritte im zweiten Kernel entsprechen im Wesentlichen denen des ersten Kernels. Der Unterschied besteht darin, dass die Prädiktionen mit einer entsprechend der numerischen Differenzierung modifizierten Kontrollsequenz u'_N berechnet werden. Die Modifikation der Kontrollsequenz zeigt erst während der Prädiktion ab einem bestimmten Horizontschritt Wirkung (siehe Kapitel 7.2). Es können also die Zwischenergebnisse der Zielfunktion und der Restriktionsfunktion verwendet werden, die im ersten Kernel berechnet wurden. Des Weiteren entfällt die Abspeicherung der Zwischenergebnisse für die Zielfunktion und Restriktionsfunktion in jedem Horizontschritt.

Die Anzahl der parallel ausgeführten Blöcke ist $NM\bar{m}$, wobei die laufende Nummer eines Threadblocks als Index für die partiell abzuleitende Komponente der Kontrollsequenz verwendet werden kann.

[1] In Abb. 8.1 grün dargestellt, mit durchgezogenem Rand.

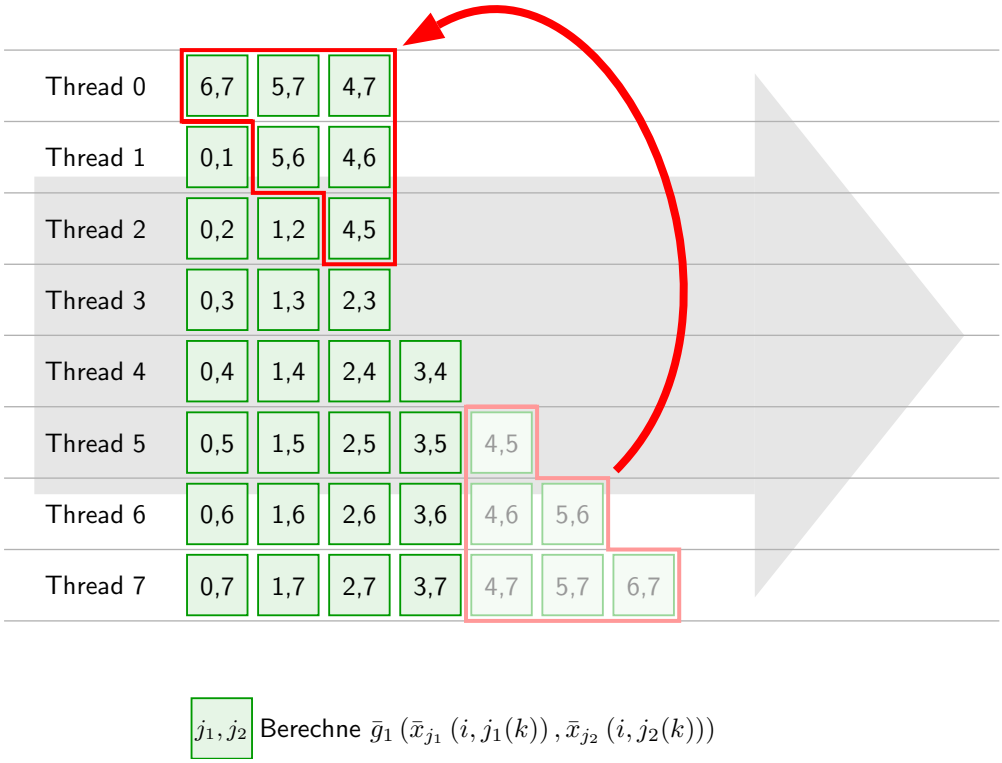


Abbildung 8.2: Exemplarische Darstellung der parallelen Auswertung von (8.5) mit $M = 8$.

8.4 Implementation der Device Kernels

Die Kernels werden wie bereits beim ROCK4–Algorithmus in Kapitel 6.3 samt aufrufendem Host Code in einer eigenen Datei implementiert. Auch hier ist es mangels Functionpointer wieder notwendig, alle Eigenschaften des Schwarmmodells fest zu implementieren. Die vollständige Umsetzung samt Host Code und kapselnden Klassen ist auf der beigelegten CD[2] zu finden.

8.4.1 Modelleigenschaften

Konstanten bzw. Längen statischer Arrays, werden als `#define`–Direktiven angegeben.

```
4  #define OBJECTSSHIFT 6
5  #define OBJECTS (1<<OBJECTSSHIFT)    // amount of objects
6  #define DIMX 4                       // object's state dimension
7  #define DIMU 2                       // object's control dimension
8
9  #define MAXHORIZ 10                  // maximum horizon length
10
11 #define ENDCOSTWEIGHT 20.0           // endcost multiplier
```

Die Minimaldistanz d zweier Objekte und $\bar{x}^{(ref)}$ sollen vom Anwender verändert werden dürfen. Die Speicherung dieser Werte erfolgt also im `__constant__` –Speicher.

[2]Siehe Dateiverzeichnis ab Seite 141.

```
13 __constant__ double cuobjdist;      // minimum distance between objects
14 __constant__ double curef[DIMX];    // array for x_ref
```

Die beiden Methoden `alpha()` und `beta()` berechnen $\alpha_{a,b}$ bzw. $\beta_{a,b}$. `restrpair()`, `restrobject()`, `objectivefunc()` und `phi()` dienen der Auswertung von $\bar{g}_1$, $\bar{g}_2$, $\bar{l}$ und $\bar{\Phi}$.

Methodenparameter von `alpha()`, `beta()`

`double xi`	Argument ξ zur Auswertung von $\alpha_{a,b}(\xi)$ und $\beta_{a,b}(\xi)$
`double a, b`	Parameter a und b der Funktionen $\alpha_{a,b}$ und $\beta_{a,b}$

```
16 __device__ double alpha(double xi, double a, double b)
17 {
18     double result = (xi-a)*(xi-b)/((b-a)*(b-a)/4.0);
19     result = -result*result*result;
20     if (result < 0.0)
21     {
22         return result;
23     }
24
25     return 0.0;
26 }
27
28 __device__ double beta(double xi, double a, double b)
29 {
30     double result = (xi-a)*(xi-b)/((b-a)*(b-a)/4.0);
31     result = result*result*result;
32     if (result < 0.0)
33     {
34         return result;
35     }
36
37     return 0.0;
38 }
```

Quellcode 8.1: Methoden zur Berechnung von $\alpha_{a,b}$ und $\beta_{a,b}$.

Methodenparameter von `restrpair()`

`double & x1, & y1`	Referenzen auf die Koordinaten $(x_1^{(i)}, x_3^{(i)})^T$ von Objekt i
`double & x2, & y2`	Referenzen auf die Koordinaten $(x_1^{(j)}, x_3^{(j)})^T$ von Objekt j

```
40 __device__ double restrpair(double & x1, double & y1, double & x2, double & y2)
41 {
42     return beta( sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)), -cuobjdist, cuobjdist);
43 }
```

Quellcode 8.2: Methode zur Berechnung einer Paarrestriktion $\bar{g}_1$ der Objekte i und j.

Methodenparameter von `restrobject()`

`double & x`	Referenz auf die Komponente $x_1^{(i)}$ des Objektzustands
`double & vx`	Referenz auf die Komponente $x_2^{(i)}$ des Objektzustands
`double & y`	Referenz auf die Komponente $x_3^{(i)}$ des Objektzustands
`double & vy`	Referenz auf die Komponente $x_4^{(i)}$ des Objektzustands

```
45 __device__ double restrobject(double & x, double & vx, double & y, double & vy)
46 {
47     double result = 0.0;
48
49     // The three circle restrictions
50     result += beta(sqrt((x-1.4)*(x-1.4) + (y-0.4)*(y-0.4)), -0.3, 0.3) +
51               beta(sqrt((x-1.4)*(x-1.4) + (y+0.4)*(y+0.4)), -0.3, 0.3) +
52               beta(sqrt((x-2.1)*(x-2.1) + y*y), -0.3, 0.3);
53
54     // Maximum velocity restriction
55     result += alpha(sqrt(vx*vx + vy*vy), -1.0, 1.0);
56
57     return result;
58 }
```

Quellcode 8.3: Methode zur Berechnung einer Objektrestriktion $\bar{g}_2$ von Objekt i.

Methodenparameter von `objectivefunc()`

`double & x`	Referenz auf die Komponente $x_1^{(i)}$ des Objektzustands
`double & vx`	Referenz auf die Komponente $x_2^{(i)}$ des Objektzustands
`double & y`	Referenz auf die Komponente $x_3^{(i)}$ des Objektzustands
`double & vy`	Referenz auf die Komponente $x_4^{(i)}$ des Objektzustands

```
60 __device__ double objectivefunc(double & x, double & vx, double & y, double & vy)
61 {
62     double result = 0.0;
63     double sqr;
64
65     sqr = x - curef[0];
66     result += sqr*sqr;
67     sqr = 0.02*(vx - curef[1]);
68     result += sqr*sqr;
69     sqr = y - curef[2];
70     result += sqr*sqr;
71     sqr = 0.02*(vy - curef[3]);
72     result += sqr*sqr;
73
74     return result;
75 }
```

Quellcode 8.4: Methode zur Berechnung Der Zielfunktion $\bar{l}$ von Objekt i.

Methodenparameter von `phi()`	
`double T`	Prädiktionszeitraum T
`double & x`	Referenz auf die Komponente $x_1^{(i)}$ des Objektzustands
`double & vx`	Referenz auf die Komponente $x_2^{(i)}$ des Objektzustands
`double & y`	Referenz auf die Komponente $x_3^{(i)}$ des Objektzustands
`double & vy`	Referenz auf die Komponente $x_4^{(i)}$ des Objektzustands
`double * u`	Anzuwendende Kontrolle $u_N^{(i)}$ (Shared Memory)

```
77 __device__ void phi(double T, double & x, double & vx,
78                                double & y, double & vy, double u[DIMU])
79 {
80     double Tsqr=T*T;
81 
82     x += vx*T + 0.5*u[0]*Tsqr;
83     vx += u[0]*T;
84     y += vy*T + 0.5*u[1]*Tsqr;
85     vy += u[1]*T;
86 }
```

Quellcode 8.5: Methode zur Berechnung der Modellfunktion $\bar{\Phi}$ von Objekt i.

8.4.2 Kernel zur Berechnung von Ziel– und Restriktionsfunktion

Für den ersten Kernel werden zusätzlich zu den Konstanten der Modelleigenschaften die folgenden Konstanten und Variablen definiert:

```
6  // current swarm state x(k)
7  __constant__ double cucurrentx[DIMX<<OBJECTSSHIFT];
8 
9  // current control sequence u_N
10 __constant__ double cucontrol[DIMU*OBJECTS*MAXHORIZ];
11 
12 // length of control horizon N
13 __constant__ unsigned int cuhorizon;
14 
15 // size of time discretization T
16 __constant__ double cudiscr;
17 
18 // storage for value of J_N
19 __device__ double cuobjvalue;
20 
21 // storage for value of g
22 __device__ double curestrvalue;
```

Zur Berechnung der Restriktionen wird die `__device__` Methode `calcrestr()` implementiert, die von allen Threads parallel aufgerufen wird. In dieser Methode wird neben der Objektrestriktion $\bar{g}_2$ für das dem aktuellen Thread zugehörige Objekt eine Reihe von paarweisen Restriktionen $\bar{g}_1$ berechnet. Die Zuteilung der Paarberechnungen auf die einzelnen Threads wurde bereits in der Abbildung 8.2 erläutert.

Methodenparameter von `calcrestr()`

`double & restr`	Rückgabevariable für die berechneten Restriktionsterme
`double * x`	Array der Länge M mit der Komponente $x_1^{(i)}$ aller Objekte (Shared Memory)
`double * vx`	Array der Länge M mit der Komponente $x_2^{(i)}$ aller Objekte (Shared Memory)
`double * y`	Array der Länge M mit der Komponente $x_3^{(i)}$ aller Objekte (Shared Memory)
`double * vy`	Array der Länge M mit der Komponente $x_4^{(i)}$ aller Objekte (Shared Memory)

```
39 __device__ void calcrestr(double & restr, double * x, double * vx, double * y, double * vy)
40 {
41     // restriction for current thread's object
42     restr += restrobject(x[threadIdx.x], vx[threadIdx.x], y[threadIdx.x], vy[threadIdx.x]);
43 
44     // parallel computation of pairwise restrictions
45     for (int j=0; j<(1<<(OBJECTSSHIFT-1)); j++)
46     {
47         // compute indizes of both objects
48         int idx1 = -1;
49         int idx2;
50         if (j<threadIdx.x)
51         {
52             idx1 = j;
53             idx2 = threadIdx.x;
54         } else if (j< (1<<(OBJECTSSHIFT-1))-1)
55         {
56             idx1 = OBJECTS-2-j;
57             idx2 = OBJECTS-1-threadIdx.x;
58         }
59 
60         // if current thread has to compute a restriction, do it...
61         if (idx1 != -1)
62         {
63             restr += restrpair(x[idx1], y[idx1], x[idx2], y[idx2]);
64         }
65     }
66 }
```

Quellcode 8.6: Methode zur Berechnung aller Restriktionsterme, die dem aufrufenden Thread zugewiesen sind.

Mithilfe der oben genannten Modellmethoden, der Methode `calcrestr()` und der Hilfsmethode `makesum()`, die, wie bereits in Kapitel 4 erläutert und in der Methode `blockaction()` des ROCK4–Algorithmus auf Seite 73 implementiert, die Summe eines übergebenen Arrays im Shared Memory bildet und das Ergebnis im ersten Arrayelement ablegt, kann nun der erste Device Kernel `cudafunckernel()` zur Auswertung von J_N und g programmiert werden.

Methodenparameter von `cudafunckernel()`

`double * objval`	Array der Länge $(N-1)$ zum Zwischenspeichern der Zielfunktionsauswertungen für die Horizontschritte $1,\ldots,N-1$ (Device Memory)
`double * restrval`	Array der Länge $(N-1)$ zum Zwischenspeichern der Restriktionsauswertungen für die Horizontschritte $1,\ldots,N-1$ (Device Memory)
`double * traj`	Array der Länge $M\bar{n}(N-1)$ zum Zwischenspeichern der Schwarmzustände für die Horizontschritte $1,\ldots,N-1$ (Device Memory)

```
69 __global__ void cudafunckernel(double * objval, double * restrval,
```

```
                                double * traj)
{
    // object state
    __shared__ double x[OBJECTS];
    __shared__ double vx[OBJECTS];
    __shared__ double y[OBJECTS];
    __shared__ double vy[OBJECTS];

    // temporary summation array
    __shared__ double sum[OBJECTS];

    double objresult;       // accumulation variable of objectivefunction-evaluations
    double restrresult;     // accumulation variable of g_1 and g_2 evaluations
    double currentu[DIMU];  // register array with current control

    // read current state from constant memory
    x[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 0];
    vx[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 1];
    y[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 2];
    vy[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 3];

    objresult = 0.0;
    restrresult = 0.0;

    // every horizon step...
    for (int i=0; i<cuhorizon; i++)
    {
        // copy object's current control vector from control sequence
        for (int j=0; j<DIMU; j++)
        {
            currentu[j] = cucontrol[i*OBJECTS*DIMU + threadIdx.x*DIMU + j];
        }
        __syncthreads();

        // compute \bar{l} of current object state
        objresult += objectivefunc(x[threadIdx.x], vx[threadIdx.x],
                                   y[threadIdx.x], vy[threadIdx.x]);

        // compute prediction to next state
        phi(cudiscr, x[threadIdx.x], vx[threadIdx.x],
                     y[threadIdx.x], vy[threadIdx.x], currentu);

        __syncthreads();

        // compute this thread's restrictions of predicted state
        calcrestr(restrresult, x, vx, y, vy);

        // if this is not the last horizon step, store temporary trajectory,
        // objective function and restiction function values for next kernel
        if (i < cuhorizon-1)
        {
            // store state to device memory
            traj[i*(DIMX*OBJECTS) + threadIdx.x] = x[threadIdx.x];
            traj[i*(DIMX*OBJECTS) + OBJECTS + threadIdx.x] = vx[threadIdx.x];
            traj[i*(DIMX*OBJECTS) + 2*OBJECTS + threadIdx.x] = y[threadIdx.x];
            traj[i*(DIMX*OBJECTS) + 3*OBJECTS + threadIdx.x] = vy[threadIdx.x];

            // add objective values of all objects and store to device memory
            sum[threadIdx.x] = objresult;
            makesum(sum);
            if (threadIdx.x == 0)
            {
                objval[i] = sum[0];
            }

            // add restriction values of all objects and store to device memory
            sum[threadIdx.x] = restrresult;
            makesum(sum);
```

```
138             if (threadIdx.x == 0)
139             {
140                 restrval[i] = sum[0];
141             }
142         }
143     }
144
145     // accumulate restriction values of all objects and store to device memory
146     sum[threadIdx.x] = restrresult;
147     makesum(sum);
148     if (threadIdx.x==0)
149     {
150         curestrvalue = sum[0];
151     }
152
153     // add final costs of every object, ...
154     sum[threadIdx.x] = objresult + ENDCOSTWEIGHT*objectivefunc(x[threadIdx.x],
155                         vx[threadIdx.x], y[threadIdx.x], vy[threadIdx.x]);
156     // ... accumulate objective values of all objects and store result to device memory
157     makesum(sum);
158     if (threadIdx.x==0)
159     {
160         cuobjvalue = sum[0];
161     }
162
163 }
```

Quellcode 8.7: Device Kernel zur Berechnung von J_N und g eines Objektschwarms.

Es ist auffällig, dass der Zustandsvektor $x(i, x(k))$ des Objektschwarms im Kernel nicht als einzelnes Array im Speicher reserviert, sondern in den Zeilen 73 bis 76 nach Objektkomponenten aufgeteilt wird. Zwar wird durch diese Maßnahme die Flexibilität des Quellcodes enorm eingeschränkt, da das Modell nicht ohne weiteres in der Dimension variiert werden kann, die Speicherbankkonflikte[3] werden jedoch auf ein Minimum reduziert.

Das Objekt j ist Thread j zugeordnet. Wenn die Komponenten des Schwarmzustands in der Form

$$(x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, x_4^{(0)}, x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, x_4^{(1)}, \ldots)$$

im Shared Memory abgelegt wären, würden beispielsweise bei einer parallelen Leseaktion von $x_i^{(j)}$ (i ist dabei fest und beispielsweise $j = 0, \ldots, 15$, da alle Threads eines Half–Warps parallel zugreifen) bei einer `double`–Variablengröße von acht Bytes gleichzeitig auf die Speicherindizes $(\bar{n}j + (i-1)) \cdot 8$ (Indizes 0, 32, 64, 96,... , bei Zugriff auf die erste Komponente, Indizes 8, 40, 72, 104,... , bei Zugriff auf die zweite, usw...) zugegriffen werden. Nach Vergleich mit Abbildung 4.6 auf Seite 44 wird deutlich, dass aufgrund dieser Anordnung achtfache Speicherbankkonflikte auftreten würden, da sämtliche Daten, auf die gleichzeitig von den 16 Threads eines Half–Warps zugegriffen werden, auf insgesamt zwei Speicherbänken liegen.

Bei einer Aufteilung des Zustandsvektors auf vier separate Arrays, wie es im Device Kernel implementiert wurde, wird der Schwarmzustand in der Form

$$(x_1^{(0)}, \ldots, x_1^{(M-1)}, x_2^{(0)}, \ldots, x_2^{(M-1)}, x_3^{(0)}, \ldots, x_3^{(M-1)}, x_4^{(0)}, \ldots, x_4^{(M-1)})$$

im Shared Memory abgelegt. Da alle parallel bearbeiteten Daten nebeneinander im Speicher liegen, werden lediglich noch zweifache Speicherbankkonflikte[4] verursacht.

[3] Siehe Kapitel 4.1.3, ab Seite 43.

[4] Zugriffe auf `double`–Variablen verursachen grundsätzlich zweifache Konflikte, da Shared Memory mit vier–Byte–Anordnung auf die Verwendung von `float`–Variablen optimiert ist.

In genau dieser Anordnung werden ebenso die Schwarmzustände im Array `double traj[]` zwischengespeichert (Zeile 122-125). Da es sich hierbei um Device Memory handelt, ist die Wahl dieser Anordnung jedoch nicht durch Vermeiden von Speicherbankkonflikten begründet, sondern durch die Tatsache, dass hierdurch ein Coalesced Memory Access[5] stattfinden kann.

8.4.3 Kernel zur Berechnung der Gradienten

Für die Berechnung des rechtsseitigen Differenzenquotienten muss eine Differenzierschrittweite definiert werden:

```
4  __constant__ double cudiffh = 1E-7;
```

Die Implementation des Device Kernels entspricht im Wesentlichen der des Kernels zur Berechnung von J_N und g. Die Berechnungen beginnen jedoch erst an dem Horizontschritt, ab dem eine numerische Differenzierung einer Kontrollkomponente die Funktionen beeinträchtigen würde; die Zwischenspeicherung der Trajektorie und der Funktionswerte entfällt ebenfalls. Statt dessen wurden Instruktionen nur Berechnung des Differenzenquotienten hinzugefügt.

Methodenparameter von `cudagradkernel()`

`double * objgrad`	Array der Länge $NM\bar{m}$, enthält anschließend ΔJ_N (Device Memory)
`double * restrgrad`	Array der Länge $NM\bar{m}$, enthält anschließend Δg (Device Memory)
`double * objval`	Berechnete Zielfunktionsauswertungen für die Horizontschritte $1, \ldots, N-1$ (Device Memory)
`double * restrval`	Berechnete Restriktionsauswertungen für die Horizontschritte $1, \ldots, N-1$ (Device Memory)
`double * traj`	Prädizierte Schwarmzustände für die Horizontschritte $1, \ldots, N-1$ (Device Memory)

```
165 __global__ void cudagradkernel(double * objgrad, double * restrgrad,
166                                double * objval, double * restrval, double * traj)
167 {
168     // object state
169     __shared__ double x[OBJECTS];
170     __shared__ double vx[OBJECTS];
171     __shared__ double y[OBJECTS];
172     __shared__ double vy[OBJECTS];
173
174     // temporary summation array
175     __shared__ double sum[OBJECTS];
176
177     double objresult;       // accumulation variable of objectivefunction-evaluations
178     double restrresult;     // accumulation variable of g_1 and g_2 evaluations
179     double currentu[DIMU];  // register array with current control
180
181     // compute the horizon index since that the numerical differentiation takes effekt
182     int horiz = blockIdx.x/(OBJECTS*DIMU);
183
184     if (horiz == 0)
185     {
186         // the first prediction: read x(k) from constant memory
```

[5] Siehe Kapitel 4.1.1, ab Seite 39.

```
187         x[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 0];
188         vx[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 1];
189         y[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 2];
190         vy[threadIdx.x] = cucurrentx[DIMX*threadIdx.x + 3];
191     }
192     else
193     {
194         // prediction starts later in horizon: read from trajectory in device memory
195         x[threadIdx.x] = traj[(horiz-1)*(DIMX*OBJECTS) + threadIdx.x];
196         vx[threadIdx.x] = traj[(horiz-1)*(DIMX*OBJECTS) + OBJECTS + threadIdx.x];
197         y[threadIdx.x] = traj[(horiz-1)*(DIMX*OBJECTS) + 2*OBJECTS + threadIdx.x];
198         vy[threadIdx.x] = traj[(horiz-1)*(DIMX*OBJECTS) + 3*OBJECTS + threadIdx.x];
199     }
200 
201     objresult = 0.0;
202     restrresult = 0.0;
203 
204     // every horizon step from "horiz" on...
205     for (int i=horiz; i<cuhorizon; i++)
206     {
207         for (int j=0; j<DIMU; j++)
208         {
209             // copy object's current control vector from control sequence
210             int diffindex = i*OBJECTS*DIMU + threadIdx.x*DIMU + j;
211             currentu[j] = cucontrol[diffindex];
212 
213             // if this control has to be disturbed due to numerical differentiation...
214             if (diffindex == blockIdx.x && gridDim.x > 1)
215             {
216                 //... add numerical differentiation stepsize to component
217                 currentu[j] += cudiffh;
218             }
219         }
220         __syncthreads();
221 
222         // compute \bar{l} of current object state
223         objresult += objectivefunc(x[threadIdx.x], vx[threadIdx.x],
224                                    y[threadIdx.x], vy[threadIdx.x]);
225 
226         // compute prediction to next state
227         phi(cudiscr, x[threadIdx.x], vx[threadIdx.x], y[threadIdx.x],
228                      vy[threadIdx.x], currentu);
229 
230         __syncthreads();
231 
232         // compute restrictions of predicted state
233         calcrestr(restrresult, x, vx, y, vy);
234     }
235 
236     // accumulate restriction values of all objects...
237     sum[threadIdx.x] = restrresult;
238     makesum(sum);
239     if (threadIdx.x==0)
240     {
241         if (horiz > 0)
242         {
243             // add restriction values of skipped predictions, computed by first kernel
244             sum[0] += restrval[horiz-1];
245         }
246         // store differential quotient to device memory
247         restrgrad[blockIdx.x] = (sum[0] - curestrvalue)/cudiffh;
248     }
249 
250     // add every object's final costs and accumulate restriction values of all objects...
251     sum[threadIdx.x] = objresult + ENDCOSTWEIGHT*objectivefunc(x[threadIdx.x],
252                                vx[threadIdx.x], y[threadIdx.x], vy[threadIdx.x]);
253     makesum(sum);
254     if (threadIdx.x==0)
```

```
255     {
256         if (horiz > 0)
257         {
258             // add objective values of skipped predictions, computed by first kernel
259             sum[0] += objval[horiz-1];
260         }
261         // store differential quotient to device memory
262         objgrad[blockIdx.x] = (sum[0] - cuobjvalue)/cudiffh;
263     }
264
265 }
```

Quellcode 8.8: Device Kernel zur Berechnung von ΔJ_N und Δg eines Objektschwarms.

8.4.4 Feinabstimmung der Kernelgrößen

Wie bereits beim ROCK4–Algorithmus auf Seite 83 sollen auch an dieser Stelle die Kernels bzgl. maximaler GPU–Ausnutzung analysiert werden. Im Gegensatz zu den Kernels des ROCK4–Algorithmus ist hier jedoch auch die Blockgröße festgelegt (durch die Objektanzahl). Eine Optimierung der GPU–Ausnutzung kann somit nur mithilfe einer Registerbeschränkung stattfinden.

Die beiden Device Kernels der Schwarmsteuerung benötigen die in Tabelle 8.1 aufgelisteten Ressourcen.

		Memory [Bytes]			
Device Kernel	Register	*Shared*	*Local*	Blockgröße	G
32 Objekte					
`cudafunckernel()`	54	1304	0	32	0.25
`cudagradkernel()`	53	1320	0	32	0.25
64 Objekte					
`cudafunckernel()`	55	2584	0	64	0.25
`cudagradkernel()`	53	2600	0	64	0.25
128 Objekte					
`cudafunckernel()`	55	5144	0	128	0.25
`cudagradkernel()`	53	5160	0	128	0.25
256 Objekte					
`cudafunckernel()`	56	10264	0	256	0.25
`cudagradkernel()`	53	10280	0	256	0.25

Tabelle 8.1: Benötigte Ressourcen der einzelnen Kernels der Schwarm–Implementation.

Um mögliche Verbesserungen der GPU–Ausnutzung G zu ermitteln, wird G für einige Registerbeschränkungen berechnet. Es zeigt sich, dass eine Beschränkung auf 40 Register die GPU–Ausnutzung bei 64 und 128 Objekten um 50% steigern würde (siehe Tabelle 8.2).

Register \ Objekte	32	64	128	256
32	0.25	0.375	0.375	0.25
34	0.25	0.375	0.375	0.25
36	0.25	0.375	0.375	0.25
38	0.25	0.375	0.375	0.25
40	0.25	0.375	0.375	0.25
42	0.25	0.3125	0.25	0.25
44	0.25	0.3125	0.25	0.25
46	0.25	0.3125	0.25	0.25
48	0.25	0.3125	0.25	0.25
50	0.25	0.25	0.25	0.25
52	0.25	0.25	0.25	0.25
54	0.25	0.25	0.25	0.25
56	0.25	0.25	0.25	0.25

Tabelle 8.2: GPU–Ausnutzung der Kernels der Schwarm–Implementation bei unterschiedlichen Register– und Blockgrößenkonfigurationen.

Eine weitere Beschränkung der Registeranzahl ist wirkungslos, da für kleinere Registeranzahlen die Menge des Shared Memory die aktive Restriktion wird[6]. Glücklicherweise wird trotz der Registerbeschränkung kein Local Memory für die Kernelausführung benötigt. Somit kann durch die Beschränkung der Registeranzahl auf 40 die GPU–Ausnutzung wirkungsvoll verbessert und die Laufzeit verkürzt werden.

8.5 Resultate und Geschwindigkeitsvergleiche

Es sollen nun die folgenden Fragen beantwortet werden:

1. Wie verhalten sich die Laufzeiten der Kernels zu äquivalenten CPU–Implementationen?

2. Wie verhalten sich die Laufzeiten eines MPC–Algorithmus während der Verwendung der Kernels zu äquivalenten CPU–Implementationen?

3. Erfüllen die Schwarmkontrollen die Erwartungen hinsichtlich der Ansteuerung von $\bar{x}^{(ref)}$ und der Einhaltung aller Restriktionen?

Zur Beantwortung dieser Fragen werden Testläufe mit $M \in \{32, 64, 128, 256\}$ und $N \in \{5, 6, 7, 8, 9, 10\}$ durchgeführt. Als Zeitdiskretisierungsintervall hat sich für dieses Schwarmproblem $T = 0.02$ als sinnvoll erwiesen und wird daher für alle Testläufe verwendet.

[6] Aktive Restriktion im Sinne des Maximierungsproblems (4.2) auf Seite 49.

8.5.1 Nettolaufzeiten der einzelnen Kernels

Der Optimierer *Ipopt* veranlasst nicht bei jeder Iteration eine Auswertung von J_N, g und deren Gradienten. Im Allgemeinen werden die Funktionen getrennt von den Gradienten berechnet. Um diesen Umstand bei den Laufzeitmessungen der Device Kernels zu berücksichtigen, werden zwei Messreihen durchgeführt:

1. Messung der Nettolaufzeiten während der Ausführung des Kernels `cudafunckernel()` zur Berechnung von J_N und g.

2. Messung der Nettolaufzeiten während einer Ausführung von `cudafunckernel()` gefolgt von einer Ausführung von `cudagradkernel()` zur Berechnung von ΔJ_N und Δg.

Die erste Messreihe, deren Ergebnisse in Abbildung 8.3 grafisch dargestellt werden, zeigt für GPU und CPU im Mittel etwa gleiche Laufzeiten. Dieses schlechte Resultat folgt aus der Tatsache, dass der Kernel in nur *einem* Threadblock ausgeführt wird und daher auch nur einer von 30 Multiprozessoren der GTX 285 Grafikkarte genutzt werden kann. Zudem werden verhältnismäßig viele Speicheroperationen durchgeführt, um Zwischenergebnisse für `cudagradkernel()` zu sichern.

Bei der Betrachtung des Speedup–Faktors[7] ist auffällig, dass dieser "quasi unabhängig"[8] von der Horizontlänge N ist, da während der Ausführung in einem einzelnen Threadblock die Berechnung bzgl. der Horizontlänge nicht parallelisiert wird (Umsetzung lediglich als serielle `for`–Schleife im Device Kernel).

Bei steigender Objektanzahl M wird jedoch die Anzahl der parallelen Threads des Threadblocks vergrößert, weshalb die Grafikkarte letztendlich etwas kürzere Laufzeiten Werte als die CPU erzielen kann.

Bessere Ergebnisse für die GPU liefert die zweite Messreihe, dargestellt in Abbildung 8.4. Hier zeigt sich, dass der relativ schlechte Speedup–Faktor des ersten Kernels nur wenig Auswirkung hat, da dessen Laufzeiten nur verhältnismäßig kurz[9] im Vergleich zu den Laufzeiten des zweiten Kernels sind.

Der zweite Kernel wird in $NM\bar{m}$ Threadblöcken ausgeführt, daher beeinflusst hier auch die Horizontlänge N den Speedup–Faktor. Insbesondere bei geringer Objektanzahl M, bei der die Grafikkarte noch nicht vollständig mit aktiven Threadblöcken ausgelastet ist, ist der Einfluss der Horizontlänge auf den Speedup–Faktor am größten.

Die einzelnen Messergebnisse sind in Tabelle 8.3 aufgelistet.

[7] Vgl. Kapitel 6.4.1 ab Seite 85.

[8] Das sehr schwache Wachstum des Speedup–Faktors bei steigender Horizontlänge wird durch die Dauer des Treiberaufrufs verursacht, die bei längerem Horizont (= längere Kernellaufzeit) weniger Auswirkung auf die Messergebnisse hat.

[9] Vgl. Tabelle 8.3.

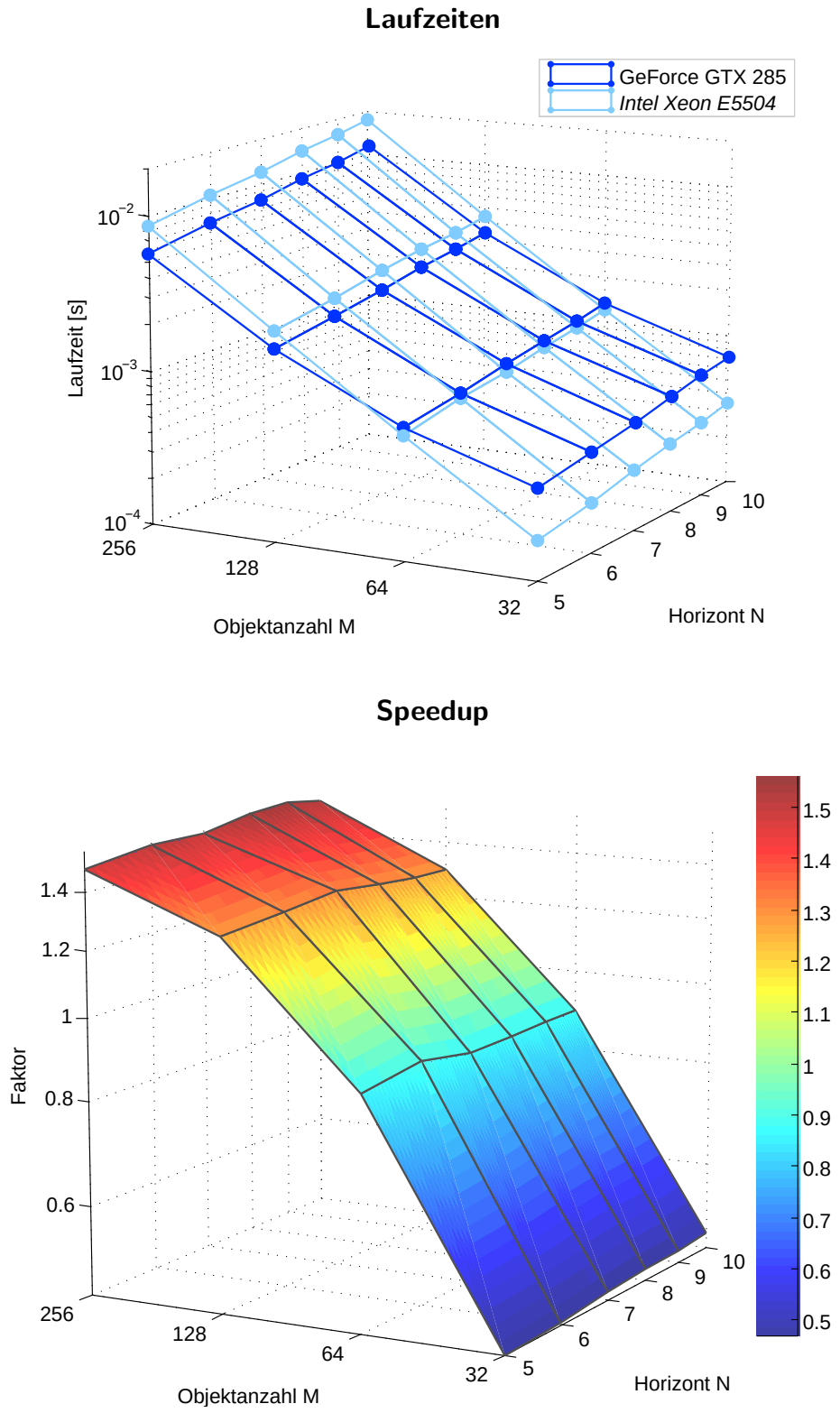


Abbildung 8.3: Nettolaufzeit– und Speedupmessungen während der Berechnung von J_N und g.

		— 1. Messreihe —			— 2. Messreihe —		
M	N	CPU [s]	GPU [s]	Speedup	CPU [s]	GPU [s]	Speedup
32	5	$1.796 \cdot 10^{-4}$	$3.815 \cdot 10^{-4}$	$\times 0.471$	$3.412 \cdot 10^{-2}$	$1.348 \cdot 10^{-3}$	$\times 25.318$
32	6	$2.106 \cdot 10^{-4}$	$4.458 \cdot 10^{-4}$	$\times 0.472$	$4.767 \cdot 10^{-2}$	$1.728 \cdot 10^{-3}$	$\times 27.587$
32	7	$2.454 \cdot 10^{-4}$	$5.102 \cdot 10^{-4}$	$\times 0.481$	$6.235 \cdot 10^{-2}$	$2.144 \cdot 10^{-3}$	$\times 29.086$
32	8	$2.790 \cdot 10^{-4}$	$5.748 \cdot 10^{-4}$	$\times 0.485$	$8.048 \cdot 10^{-2}$	$2.648 \cdot 10^{-3}$	$\times 30.391$
32	9	$3.121 \cdot 10^{-4}$	$6.394 \cdot 10^{-4}$	$\times 0.488$	$9.969 \cdot 10^{-2}$	$3.179 \cdot 10^{-3}$	$\times 31.361$
32	10	$3.456 \cdot 10^{-4}$	$7.040 \cdot 10^{-4}$	$\times 0.491$	$1.210 \cdot 10^{-1}$	$3.786 \cdot 10^{-3}$	$\times 31.973$
64	5	$6.268 \cdot 10^{-4}$	$7.054 \cdot 10^{-4}$	$\times 0.889$	$2.322 \cdot 10^{-1}$	$5.881 \cdot 10^{-3}$	$\times 39.485$
64	6	$7.585 \cdot 10^{-4}$	$8.314 \cdot 10^{-4}$	$\times 0.912$	$3.248 \cdot 10^{-1}$	$8.015 \cdot 10^{-3}$	$\times 40.524$
64	7	$8.518 \cdot 10^{-4}$	$9.599 \cdot 10^{-4}$	$\times 0.887$	$4.327 \cdot 10^{-1}$	$1.040 \cdot 10^{-2}$	$\times 41.618$
64	8	$9.685 \cdot 10^{-4}$	$1.089 \cdot 10^{-3}$	$\times 0.889$	$5.538 \cdot 10^{-1}$	$1.316 \cdot 10^{-2}$	$\times 42.081$
64	9	$1.088 \cdot 10^{-3}$	$1.217 \cdot 10^{-3}$	$\times 0.894$	$6.900 \cdot 10^{-1}$	$1.623 \cdot 10^{-2}$	$\times 42.529$
64	10	$1.204 \cdot 10^{-3}$	$1.345 \cdot 10^{-3}$	$\times 0.895$	$8.414 \cdot 10^{-1}$	$1.960 \cdot 10^{-2}$	$\times 42.934$
128	5	$2.245 \cdot 10^{-3}$	$1.737 \cdot 10^{-3}$	$\times 1.293$	1.723	$3.966 \cdot 10^{-2}$	$\times 43.444$
128	6	$2.707 \cdot 10^{-3}$	$2.067 \cdot 10^{-3}$	$\times 1.309$	2.410	$5.490 \cdot 10^{-2}$	$\times 43.897$
128	7	$3.201 \cdot 10^{-3}$	$2.401 \cdot 10^{-3}$	$\times 1.333$	3.212	$7.243 \cdot 10^{-2}$	$\times 44.352$
128	8	$3.600 \cdot 10^{-3}$	$2.735 \cdot 10^{-3}$	$\times 1.316$	4.118	$9.254 \cdot 10^{-2}$	$\times 44.505$
128	9	$3.995 \cdot 10^{-3}$	$3.071 \cdot 10^{-3}$	$\times 1.301$	5.138	$1.149 \cdot 10^{-1}$	$\times 44.694$
128	10	$4.403 \cdot 10^{-3}$	$3.403 \cdot 10^{-3}$	$\times 1.294$	6.270	$1.399 \cdot 10^{-1}$	$\times 44.814$
256	5	$8.523 \cdot 10^{-3}$	$5.719 \cdot 10^{-3}$	$\times 1.490$	$1.358 \cdot 10^{1}$	$2.974 \cdot 10^{-1}$	$\times 45.675$
256	6	$1.045 \cdot 10^{-2}$	$6.844 \cdot 10^{-3}$	$\times 1.527$	$1.839 \cdot 10^{1}$	$4.135 \cdot 10^{-1}$	$\times 44.467$
256	7	$1.208 \cdot 10^{-2}$	$7.970 \cdot 10^{-3}$	$\times 1.516$	$2.453 \cdot 10^{1}$	$5.485 \cdot 10^{-1}$	$\times 44.711$
256	8	$1.416 \cdot 10^{-2}$	$9.097 \cdot 10^{-3}$	$\times 1.557$	$3.148 \cdot 10^{1}$	$7.025 \cdot 10^{-1}$	$\times 44.804$
256	9	$1.597 \cdot 10^{-2}$	$1.022 \cdot 10^{-2}$	$\times 1.562$	$3.932 \cdot 10^{1}$	$8.754 \cdot 10^{-1}$	$\times 44.915$
256	10	$1.747 \cdot 10^{-2}$	$1.135 \cdot 10^{-2}$	$\times 1.538$	$4.804 \cdot 10^{1}$	1.067	$\times 45.013$

Tabelle 8.3: Nettolaufzeiten der Device Kernels zur Berechnung von J_N, g und deren Gradienten bei unterschiedlichen Objektanzahlen und Horizontlängen.

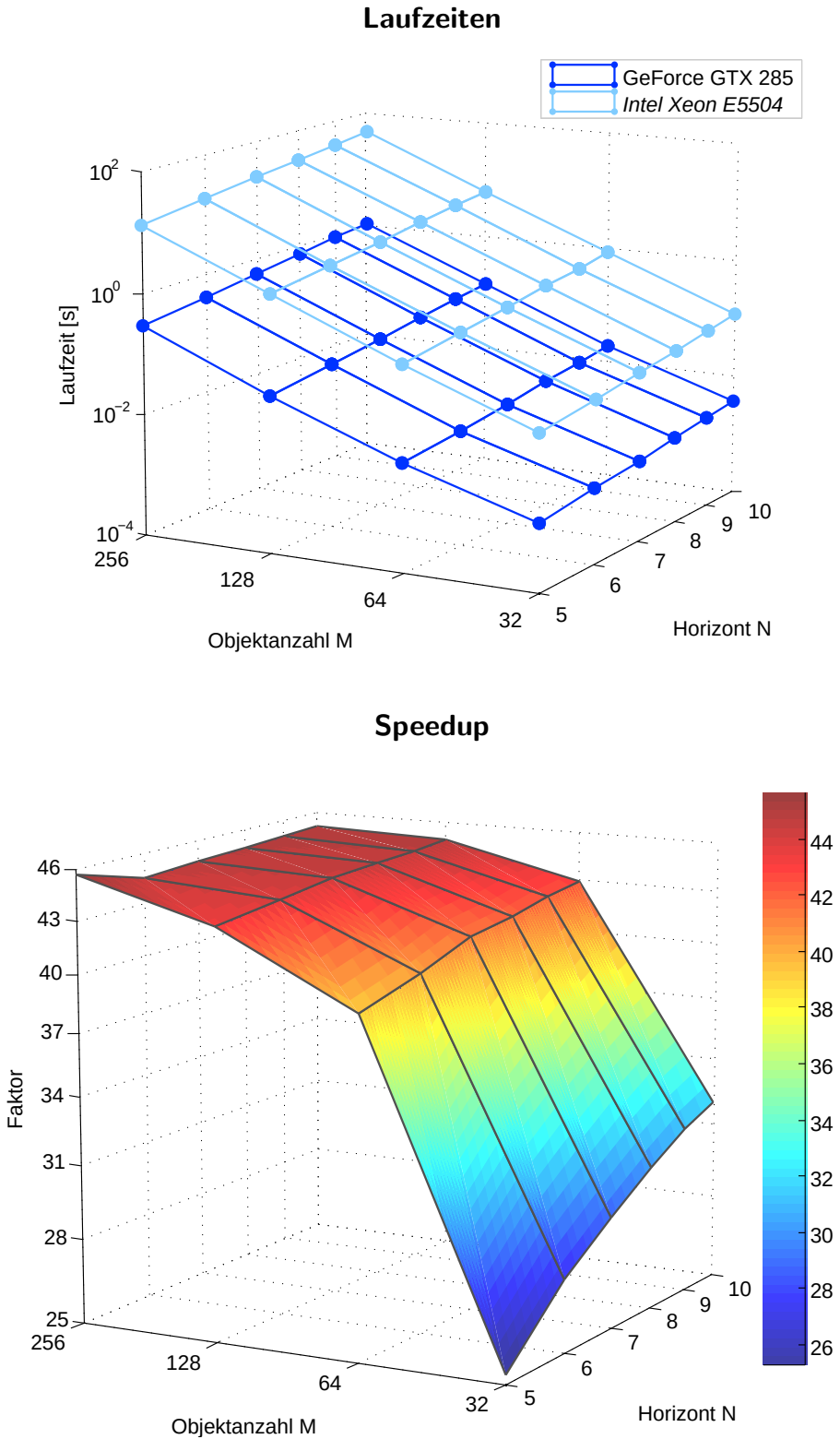


Abbildung 8.4: Nettolaufzeit– und Speedupmessungen während der Berechnung von ΔJ_N und Δg.

8.5.2 Laufzeiten des MPC–Algorithmus

Die Speedup–Faktoren der Funktionsauswertungen unterscheiden sich offensichtlich erheblich von den Speedup–Faktoren der Gradientenauswertungen. Unter Berücksichtigung der Tatsache, dass der *Ipopt*–Optimierer selbst auch Rechenzeit benötigt, und Funktionen und Gradienten während der Iterationen des Optimierers unterschiedlich oft berechnet werden, können aus den Speedup–Faktoren in Tabelle 8.3 keine Voraussagen über die Geschwindigkeitsverhältnisse beim Einsatz der Device Kernels im MPC–Algorithmus getroffen werden. Dieser Umstand motiviert eine weitere Messreihe.

Es sollen die Rechenzeiten des MPC–Algorithmus während der ersten zehn MPC–Schritte gemessen werden. Die Rahmenbedingungen seien definiert durch

$$\begin{aligned} \bar{x}^{(ref)} &:= 0_{\mathbb{R}^{\bar{n}}} \\ x^{(j)}(0) &:= \begin{pmatrix} j \cdot 0.12 \\ 0 \\ 1 \\ 0 \end{pmatrix} \end{aligned} \tag{8.6}$$

Die Toleranz des Optimierers sei 10^{-3}.

Es ergeben sich die Messdaten aus Tabelle 8.4. Die Ergebnisse sind in Abbildung 8.5 grafisch dargestellt. Hierzu muss erwähnt sein, dass ausschließlich die Laufzeiten der Grafikkarte real gemessen wurden und die Anzahl der Auswertungen und der *Ipopt*–Anteil sich eben auf diese Messungen beziehen. Die CPU–Laufzeiten wurden aus Zeitgründen anhand der Laufzeiten der einzelnen Kernels aus Tabelle 8.3 und der Anzahl deren Aufrufe hochgerechnet. Da sich die Ergebnisse der GPU–Implementierung aus numerischen Gründen von denen der CPU–Varianten leicht unterscheiden können – in Kapitel 6.4.2 wurde in Verbindung mit dem ROCK4–Algorithmus bereits genauer auf dieses Phänomen eingegangen – ist anzunehmen, dass sich auch die Iterationen des Optimierers bei der CPU–Variante etwas von der GPU–Variante unterscheiden würden und die hochgerechneten CPU–Laufzeiten somit nicht exakt der Realität entsprechen. Eine näherungsweise Bestimmung des Speedup–Faktors ist jedoch durchaus trotz dieser Vereinfachung gerechtfertigt.

Der hohe Speedup–Faktor der Gradientenauswertung scheint gerade bei großen Problemen besonders gewichtig in die Rechenzeit der MPC–Schritte einzugehen, weshalb die GPU–Variante des MPC–Algorithmus hohe Geschwindigkeitsvorteile erzielt.

Ein Grund hierfür ist, dass während der Iterationen des *Ipopt*–Optimierers alle Gradientenauswertungen immer in Einheit mit Funktionenauswertungen einhergehen. Es werden also beide Device Kernels nacheinander ausgeführt, was gerade die hohen Speedup–Faktoren in der zweiten Messreihe in Tabelle 8.3 zur Folge hatte. Einzelne Funktionsauswertungen ohne Gradienten kommen nur sehr selten vor und haben somit angesichts der verhältnismäßig geringen Kernellaufzeit quasi keinen Einfluss auf den Speedup–Faktor.

Verringert wird der Speedup–Faktor lediglich noch durch die Dauer der internen Berechnungen des *Ipopt*–Optimierers, die während der Messungen im schlechtesten Fall sogar über 50% der gesamten GPU–Laufzeit ausgemacht hat. Diese Laufzeiten sind für CPU– und GPU–Variante des MPC–Algorithmus identisch und reduzieren somit den Geschwindigkeitsvorteil der GPU–Implementation.

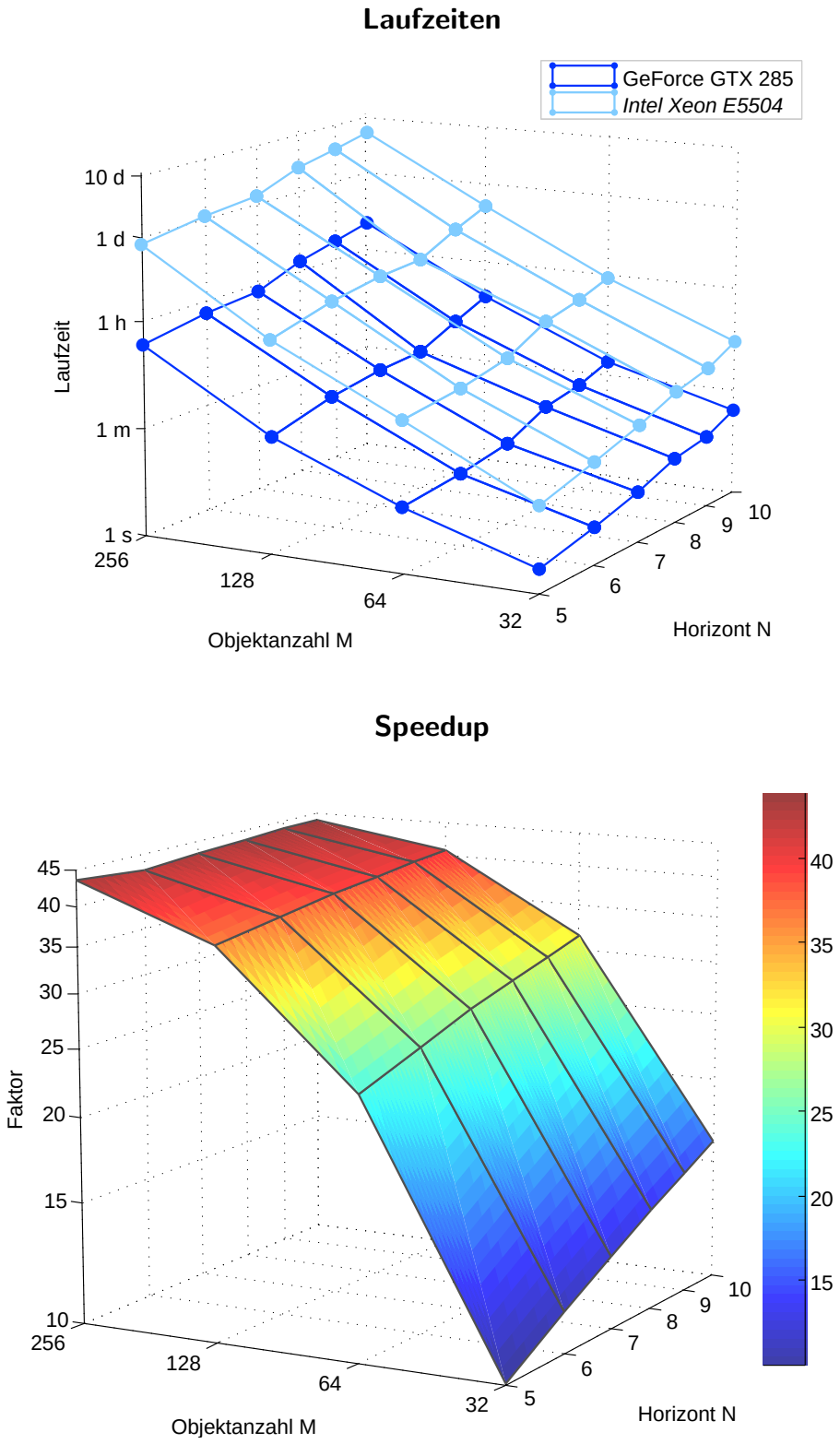


Abbildung 8.5: Laufzeit– und Speedupmessungen während der Berechung von zehn MPC–Schritten.

		— Rechenzeiten —				Auswertungen	
M	N	CPU [s]	GPU [s]	*Ipopt*–Anteil [s]	Speedup	J_N, g	ΔJ_N, Δg
32	5	$2.579 \cdot 10^1$	2.572	1.328	×10.029	741	713
32	6	$4.723 \cdot 10^1$	4.100	2.027	×11.519	992	944
32	7	$8.778 \cdot 10^1$	6.869	3.256	×12.780	1408	1350
32	8	$1.664 \cdot 10^2$	$1.183 \cdot 10^1$	5.356	×14.067	2076	1994
32	9	$2.421 \cdot 10^2$	$1.604 \cdot 10^1$	6.987	×15.093	2475	2351
32	10	$4.214 \cdot 10^2$	$2.617 \cdot 10^1$	$1.088 \cdot 10^1$	×16.099	3545	3381
64	5	$2.950 \cdot 10^2$	$1.236 \cdot 10^1$	4.111	×23.866	1279	1249
64	6	$4.049 \cdot 10^2$	$1.564 \cdot 10^1$	4.730	×25.893	1270	1229
64	7	$6.228 \cdot 10^2$	$2.247 \cdot 10^1$	6.279	×27.712	1470	1422
64	8	$1.431 \cdot 10^3$	$4.924 \cdot 10^1$	$1.273 \cdot 10^1$	×29.065	2635	2557
64	9	$1.988 \cdot 10^3$	$6.571 \cdot 10^1$	$1.579 \cdot 10^1$	×30.261	2966	2854
64	10	$3.211 \cdot 10^3$	$1.023 \cdot 10^2$	$2.290 \cdot 10^1$	×31.371	3949	3783
128	5	$3.155 \cdot 10^3$	$8.565 \cdot 10^1$	$1.010 \cdot 10^1$	×36.840	1871	1823
128	6	$6.080 \cdot 10^3$	$1.598 \cdot 10^2$	$1.650 \cdot 10^1$	×38.052	2569	2513
128	7	$8.757 \cdot 10^3$	$2.241 \cdot 10^2$	$2.060 \cdot 10^1$	×39.079	2792	2717
128	8	$9.430 \cdot 10^3$	$2.374 \cdot 10^2$	$1.974 \cdot 10^1$	×39.718	2353	2283
128	9	$2.004 \cdot 10^4$	$4.969 \cdot 10^2$	$3.727 \cdot 10^1$	×40.339	4016	3891
128	10	$3.627 \cdot 10^4$	$8.887 \cdot 10^2$	$6.111 \cdot 10^1$	×40.813	5933	5771
256	5	$6.683 \cdot 10^4$	$1.532 \cdot 10^3$	$4.209 \cdot 10^1$	×43.613	5065	4914
256	6	$9.917 \cdot 10^4$	$2.321 \cdot 10^3$	$5.530 \cdot 10^1$	×42.735	5492	5387
256	7	$1.236 \cdot 10^5$	$2.862 \cdot 10^3$	$5.999 \cdot 10^1$	×43.175	5161	5034
256	8	$2.408 \cdot 10^5$	$5.546 \cdot 10^3$	$1.037 \cdot 10^2$	×43.425	7885	7644
256	9	$3.332 \cdot 10^5$	$7.631 \cdot 10^3$	$1.288 \cdot 10^2$	×43.667	8726	8468
256	10	$4.841 \cdot 10^5$	$1.103 \cdot 10^4$	$1.701 \cdot 10^2$	×43.871	10351	10069

Tabelle 8.4: Benötigte Rechenzeit für zehn MPC–Schritte bei unterschiedlichen Objektanzahlen und Horizontlängen.

8.5.3 Das Verhalten der Objekte

Nachdem mithilfe erster kurzer Testläufe der Speedup–Faktor festgestellt wurde, sollen nun 2000 MPC–Schritte berechnet werden. Bei den Berechnungen wird sich aus Zeitgründen auf die Horizontlänge $N = 5$ und die GPU–Variante beschränkt. Es gilt nun empirisch zu überprüfen, ob die Schwarmsteuerung die vorgegebenen Modellbedingungen aus Kapitel 8.3 erfüllt, also die Objekte den Referenzpunkt $\bar{x}_{ref}$ ansteuern und alle Restriktionen eingehalten werden.

Hierfür werden zwei Szenarien erzeugt. In den MPC–Schritten $k = 0, \ldots, 999$ ist der Referenzpunkt als $\bar{x}^{(ref)} = (0, 0, 0, 0)^T$ definiert. Die Startbedingungen des Schwarms sind gemäß (8.6) gegeben. In diesem Szenario sollte die Zielstrebigkeit der einzelnen Objekte bei der Ansteuerung des Referenzpunktes gut erkennbar sein. Ebenso wird sich zeigen, ob die Abstandsrestriktionen während der Ansammlung bei $\bar{x}_{ref}$ sowie die maximale Geschwindigkeit eingehalten werden.

Das zweite Szenario tritt ab $k = 1000$ ein. Der Referenzpunkt wird sprungartig nach

$\bar{x}^{(ref)} = (3, 0, 0, 0)^T$ verschoben und der Schwarm muss nun an den drei "verbotenen" Zonen[10] vorbeigesteuert werden. Interessant ist in diesem Szenario die Art und Weise, wie sich der Schwarm in Bewegung setzt, wenn die Ausgangskonstellation bereits aus numerischer Sicht optimal ist. Zudem soll überprüft werden, ob der Optimierer auch in Situationen, in denen es "sehr eng" für die Objekte wird, zulässige Lösungen findet und keine Objekte von der Masse "eingequetscht" werden.

Die Abbildungen 8.6 und 8.7 illustrieren die berechneten Resultate der beiden Szenarien anhand eines Schwarms mit 64 Objekten[11]. Um die paarweisen Abstandsrestriktionen zu verdeutlichen, werden die Objekte als blaue Kreisscheiben mit Radius $r = 0.05$ dargestellt. Die drei verbotenen Mengen sind als rote Kreise mit Radius $r = 0.25$ eingezeichnet und der Referenzpunkt $(\bar{x}_1^{(ref)}, \bar{x}_3^{(ref)})$ ist mit einem kleinen roten Kreis markiert.

Es zeigt sich, dass alle Objekte zu den aktuellen Referenzpunkten gesteuert werden, ohne Restriktionen zu verletzen. Zwar ist die Einhaltung der Geschwindigkeitsbeschränkung nicht den Abbildungen zu entnehmen, die Ausgabe der Schwarmzustände zwischen den MPC–Schritten hat jedoch während der Berechnung bestätigt, dass die Geschwindigkeitsvektoren der einzelnen Objekte immer betragsmäßig kleiner als eins sind. Beim Eintritt in das zweite Szenario $k = 1000, \ldots, 1999$ setzt sich der Schwarm geschlossen in Bewegung und die drei Zonen werden ohne Probleme passiert.

Zu beobachten ist jedoch auch, dass mit steigender Objektanzahl die Kontrolle eines einzelnen Objekts an Präzision verliert, sobald es sich ortsmäßig bereits relativ nahe am anzusteuernden Referenzpunkt $\bar{x}^{(ref)}$ befindet. Die Abbildungen 8.8 und 8.9 demonstrieren diesen Effekt für die verschiedenen Objektanzahlen[12]. Während bei 32 Objekten noch eine relativ präzise Steuerung zum Referenzpunkt stattfindet, scheinen sich bei größeren Anzahlen die einzelnen Objekte dem Referenzpunkt nur noch "ungenau" zu nähern und sich erst einige MPC–Schritte später zu sammeln. Mit steigender Objektanzahl umgehen die Objekte die roten Bereiche offenbar weiträumiger, anstatt nach dem Passieren wieder direkt Kurs auf den Referenzpunkt zu nehmen.

Dieses Verhalten ist in der numerischen Ungenauigkeit des Optimierers begründet und wird sowohl durch die quadratische Zielfunktion als auch durch die schwächere Gewichtung der Geschwindigkeitskomponenten provoziert. Mit steigender Objektanzahl liefern die einzelnen Objekte einen immer kleineren Beitrag zur Zielfunktion. Insbesondere werden die potentiellen Abstiege in der Zielfunktion eines Objekts wegen der Toleranz des Optimierers nicht mehr oder nur noch schwach wahrgenommen, wenn es sich bereits nahe am Referenzpunkt befindet, während einige Objekte noch weit entfernt sind.

Insgesamt betrachtet konnten mit der GPU deutliche Geschwindigkeitsvorteile gegenüber der CPU erzielt werden. Generell scheint jedoch die begrenzte Genauigkeit der Gleitkommaarithmetik die Anzahl der Schwarmobjekte auf natürliche Weise zu beschränkten. Eine Schwarmgröße von 64 Objekten hat sich in den Experimenten als ein guter Kompromiss zwischen dem Speedupfaktor und der Kontrollpräzision erwiesen.

[10] Vgl. hierzu Kapitel 8.3.

[11] Man beachte, dass manche Objekte ggf. nicht dargestellt werden, da sie sich möglicherweise außerhalb des dargestellten Bereichs befinden. Animationen der berechneten Schwarmsteuerungen für 32, 64, 128 und 256 Objekte sind auf der beigelegten CD zu finden.

[12] Bei jeder Objektanzahl wurde genau der MPC–Schritt abgebildet, bei dem sich die Objekte am weitesten über das Ziel hinaus bewegt haben.

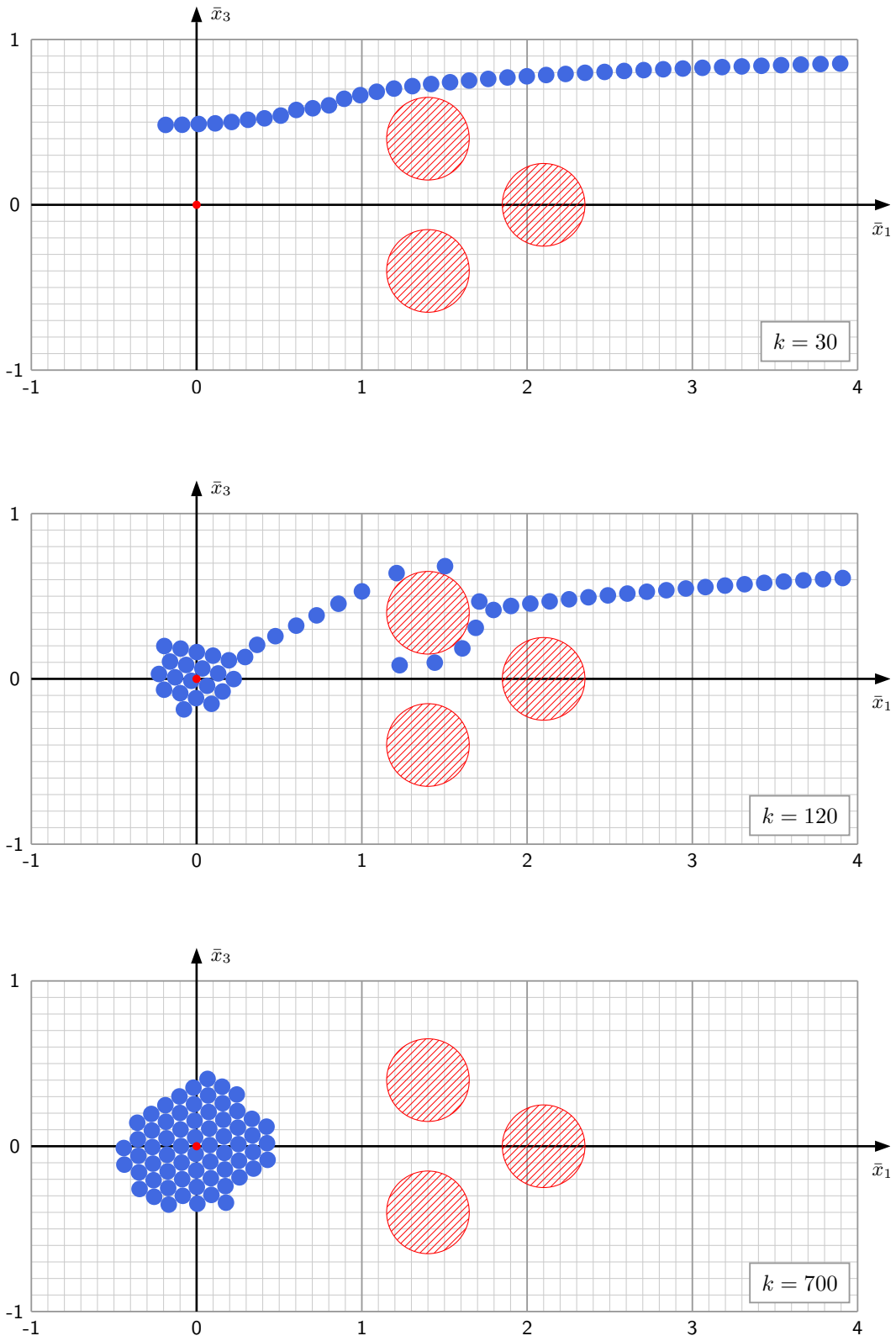


Abbildung 8.6: Das Verhalten eines Schwarms mit 64 Objekten bei der Ansteuerung des Referenzpunkts $\bar{x}^{(ref)} = (0, 0, 0, 0)^T$.

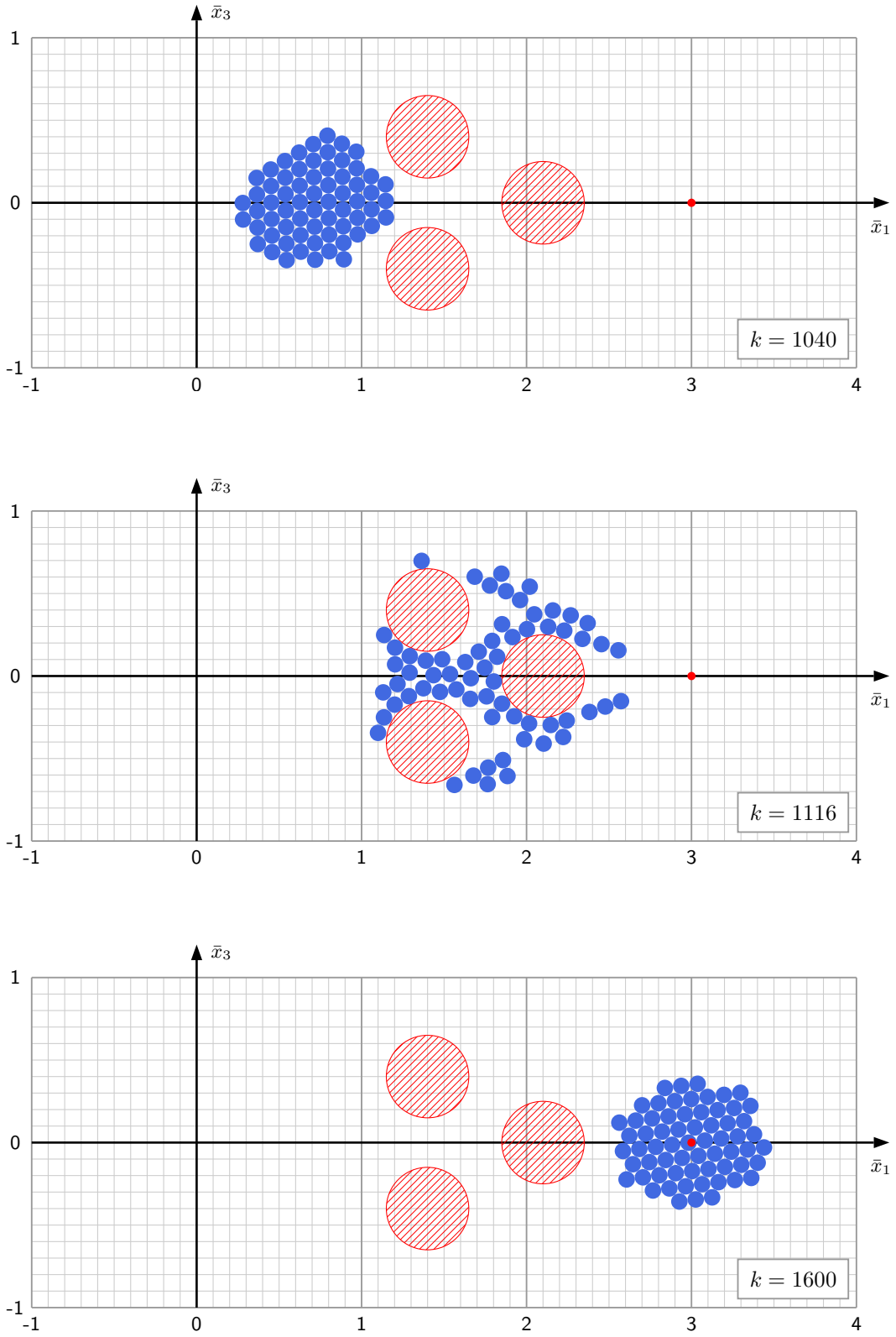


Abbildung 8.7: Das Verhalten eines Schwarms mit 64 Objekten nach Änderung des Referenzpunkts auf $\bar{x}^{(ref)} = (3, 0, 0, 0)^T$.

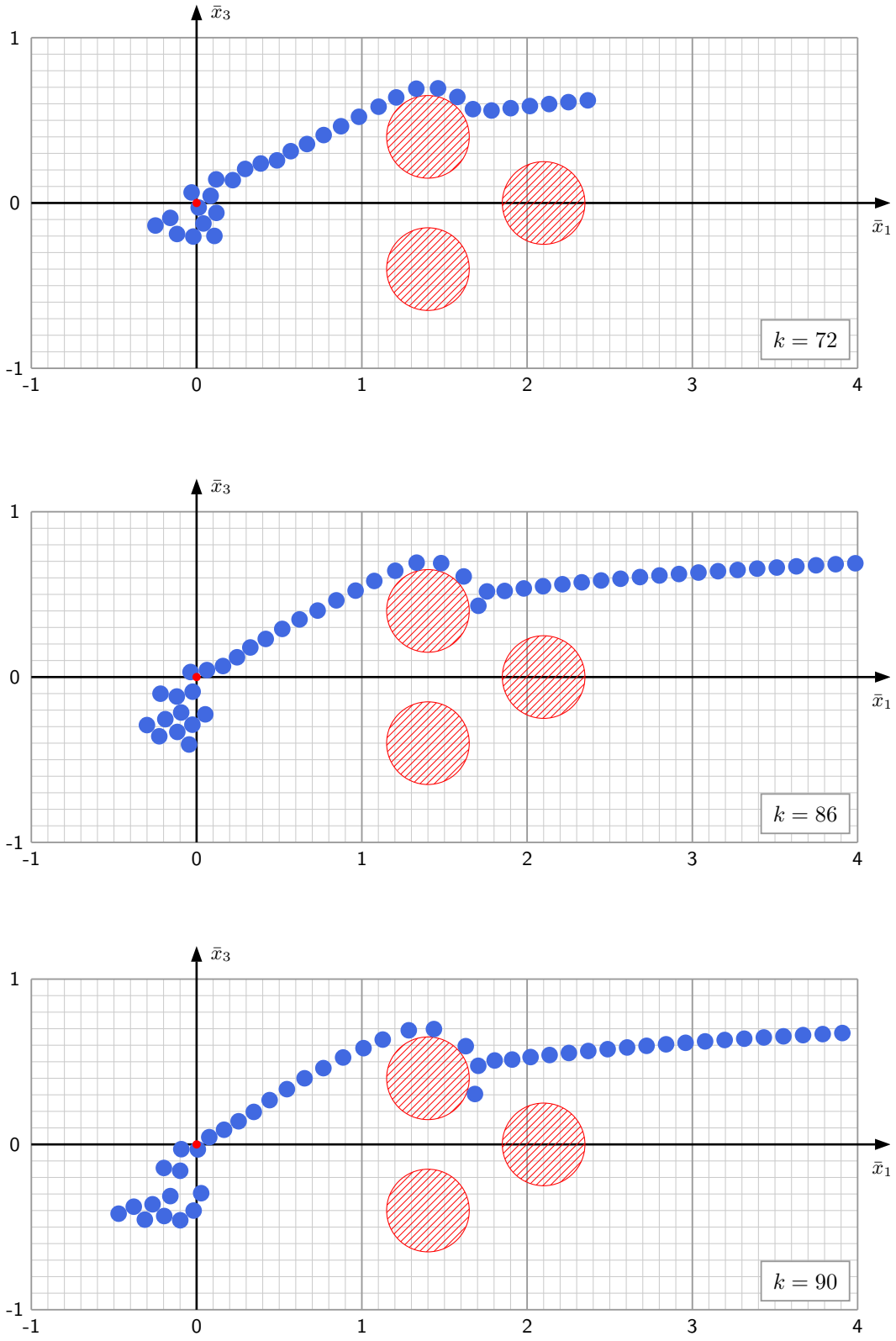


Abbildung 8.8: Zustand unterschiedlicher Schwärme (von oben nach unten: 32, 64 und 128 Objekte) kurz nach dem Eintreffen der ersten Objekte bei $(\bar{x}_1^{(ref)}, \bar{x}_3^{(ref)}) = (0, 0)$.

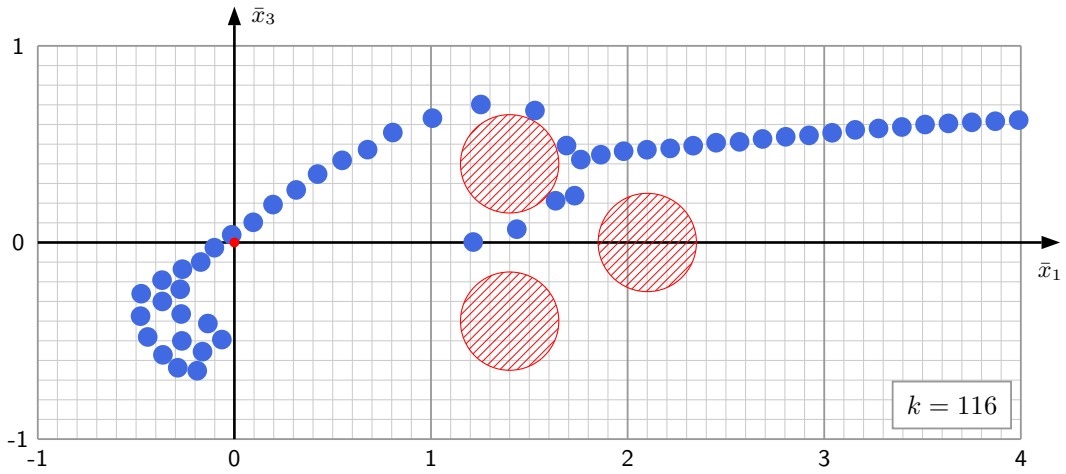


Abbildung 8.9: Zustand eines Schwarms mit 256 Objekten kurz nach dem Eintreffen der ersten Objekte bei $(\bar{x}_1^{(ref)}, \bar{x}_3^{(ref)}) = (0, 0)$.

Teil IV

Abschließende Betrachtung

Kapitel 9

Fazit

Zum Abschluss dieser Arbeit soll die Verwendung der CUDA–Architektur zur Implementierung numerischer Algorithmen kritisch betrachtet werden. Ebenso wird ein Ausblick auf die zukünftige Entwicklung der GPU–Programmierung angestellt.

9.1 Vorteile und Nachteile

Der Einstieg in die CUDA–Programmierung kann dem Programmierer eine hohe Frustrationstoleranz abverlangen. Ein Grund hierfür dürfte sein, dass sich möglicherweise der Abhängigkeitsgraph eines gewünschten Algorithmus trotz größter Bemühungen nicht in genügend unabhängige Teilgraphen zerlegen lässt und die Rechenleistung der Hardware daher nur zu einem Bruchteil genutzt werden kann. Die Implementierung des ROCK4–Algorithmus in dieser Arbeit hat auch gezeigt, dass, obwohl ein Algorithmus breit genug parallelisierbar ist, die Notwendigkeit einer Synchronisation aller Tasks zu Programmen mit geringerer Effizienz führen kann, da in diesem Fall oft Zwischenergebnisse im Device Memory abgelegt und Device Kernels neu gestartet werden müssen. Aufgrund dieser Tatsache ist in der Praxis bei den meisten Algorithmen eine parallelisierte Implementation auf der CPU rentabler.

Das Schreiben eines Programms für die GPU ist – und dies ist der zweite Aspekt, der gegen die GPU spricht – sehr zeitraubend. Zur Erinnerung: Die extrem aufwändige GPU–Implementierung der simplen Bestimmung eines Maximums oder einer Summe in Kapitel 6.3.2 könnte auf der CPU in sehr wenigen Quellcode–Zeilen umgesetzt werden. Diese vielfach höhere Entwicklungszeit von GPU–Programmen muss daher durch einen guten Speedupfaktor gerechtfertigt werden.

Der *nvcc*–Compiler ist, wie auch die Runtime Library und der Grafiktreiber, Beta–Software. Es kann also davon ausgegangen werden, dass diese Software Bugs enthält, die dem CUDA–Programmierer das Leben erschweren, denn sie hindern ggf. Device Kernels sehr effektiv an ihrer fehlerfreien Ausführung. Compilerbugs sind für Programmierer sehr problematisch, da falsche Resultate eines Programms im Allgemeinen auf Fehler im Quellcode des Programmierers zurückzuführen sind. Eine Fehlersuche kann daher in diesem Fall vielfach mehr Zeit in Anspruch nehmen und erfordert ggf. Recherche in den von *n*Vidia betriebenen Internetforen für CUDA–Entwickler. Im seltenen Ernstfall können die Auswirkungen eines Compilerfehlers nur vermieden werden, indem der Algorithmus anders implementiert wird.

Als weiterer Nachteil sind natürlich auch die Einschränkungen aufzuführen, die bereits in Kapitel 3.4 beschrieben wurden. Das Fehlen von Functionpointern kann die Programmierung allgemeiner Bibliotheken erschweren. Eine CPU–Bibliothek mit einem Differentialgleichungslöser kann beispielsweise bereits mit einem einzigen Functionpointer an ein Problem angepasst werden. Der Zeiger würde auf eine Methode verweisen, die die rechte Seite f der DGL auswertet. Diese Vorgehensweise ist im Device Code nicht möglich, da alle Methoden *inline* kompiliert werden. Der Anwender müsste der Bibliothek einen vollständigen Device Kernel übergeben, der das numerische Verfahren beinhaltet und die rechte Seite f bereits fest integriert hat, was den Zweck einer solchen Bibliothek fragwürdig erscheinen lässt.

All diesen Nachteilen steht ein einziger essentieller Vorteil gegenüber: Bei einem geeigneten Problem und einer effizienten Implementierung ist die Geschwindigkeitssteigerung enorm. Der Speedupfaktor $\times 40$ der MPC–Implementierung auf der GPU lässt die Größenordnung der Laufzeit von Wochen oder einem Monat auf Stunden schrumpfen. Diese Tatsache kann Grund genug sein, etwa längere Entwicklungszeiten, oder sehr spezielle Programmierung mit wenig Verwendungsmöglichkeiten des Programms für andere Problemstellungen, in Kauf zu nehmen.

9.2 Vorausblick

Obwohl die CUDA–Architektur beeindruckende Rechenleistung für wissenschaftliche Anwendungen bietet, so bleibt die Hardware immer noch eine *Grafik*karte, die ursprünglich darauf optimiert wurde, realistisch anmutende Bilder mit `float`–Genauigkeit zu rendern[1]. Um diese Optimierung deutlich zu machen, wurden die Kernels aus Kapitel 8.4 und deren äquivalente CPU–Implementierungen in einem weiteren Test ausschließlich unter Verwendung von `float`–Variablen kompiliert und die Messreihen aus Kapitel 8.5.1 erneut durchgeführt. Als Resultat wurden Speedupfaktoren zwischen $\times 1.27$ – $\times 12.86$ in der ersten und $\times 104.3$ – $\times 504.4$ in der zweiten Messreihe festgestellt. Dieser Geschwindigkeitsvorteil ist überragend, jedoch in diesem Fall in der Praxis leider irrelevant, da die auf diese Weise berechneten Ergebnisse für numerisches Differenzieren in einem nichtlinearen Programm benötigt werden, das eine gewisse Präzision in den Gradientenberechnungen erwartet.

Die nächste Generation der CUDA–Architektur, genannt *Fermi*[2], soll diese Lücke zwischen einer Grafikkarte für 3D–Anwendungen und einer Hardware für präzise algorithmische Berechnungen schließen und durch zusätzliche Fähigkeiten der GPU die Menge der implementierbaren Algorithmen erweitern.

Auf den Multiprozessoren der Fermi–Architektur arbeiten 32 Prozessoren. Bisher wurde jeder Multiprozessor mit acht Prozessoren bestückt, sodass allein basierend auf dieser Verbesserung die Threadblöcke viermal schneller bearbeitet werden könnten, als es bisher der Fall war.

Den Angaben des Herstellers zufolge sind `double`–Berechnungen mit der neuen Architektur achtmal schneller möglich, als mit der ersten CUDA–Generation. Eine zusätzliche Erweiterung des Anwendungsspektrums bietet die Vergrößerung des Shared Memory von 16

[1] Intrinsic Functions arbeiten höchstens mit `float`–Genauigkeit, Speicherbänke des Shared Memory sind für 4–Byte–Variablen optimiert, nur 32–Bit–Register, usw...

[2] Siehe [11, 12].

KB auf 64 KB. Das neue Shared Memory kann zusätzlich teilweise als L1–Cache verwendet werden. Dies erlaubt die Auslagerung von Registerinhalten, sodass eine geringere Anzahl an Registern benötigt wird und somit wesentlich komplexere Device Kernels programmiert werden können bzw. eine höhere GPU–Auslastung erzielt werden kann.

Eine besondere Errungenschaft ist die Fähigkeit der nächsten CUDA–Generation, bis zu 16 inhaltlich verschiedene Kernels gleichzeitig auszuführen. Dies bringt u.a. zwei wesentliche Vorteile für die Algorithmenprogrammierung mit sich:

1. Es können auch diejenigen Algorithmen effizient implementiert werden, bei denen sich der Abhängigkeitsgraph zwar in unabhängige Teilgraphen zerlegen lässt, diese jedoch teilweise unterschiedlich strukturiert sind. Bei der bisherigen CUDA–Architektur hätte dies eine serielle Bearbeitung der Teilgraphen zur Folge gehabt, die bei relativ kleinen Blockgittern sogar die Grafikkarte nur unvollständig ausnutzen würden.

2. Bei einem Multiuser–System können mehrere Anwender gleichzeitig die CUDA–Hardware nutzen. Bei der bisherigen Architektur wäre die Grafikkarte während der Verwendung durch einen Benutzer für andere Anwender gesperrt. Aufgrund dieser neuen Fähigkeit lässt sich die Hardware in Compute Servern effizienter einsetzen.

Allem Anschein nach wurde mit der CUDA–Architektur der ersten Generation der Startschuss für das GPU–Computing im wissenschaftlichen Anwendungsbereich gegeben. Bereits die im ersten Quartal 2010 veröffentlichte Fermi–Architektur verspricht mehr Flexibilität und vielfach höhere Speedupfaktoren, als im Rahmen der Untersuchungen zu dieser Arbeit erreicht wurden. Dank derartiger Entwicklungen könnten besonders rechenlastige Algorithmen in Zukunft für zeitkritische Bereiche attraktiv werden, in denen bisher eine Echtzeitanwendung unvorstellbar gewesen ist.

Teil V

Anhang

Dateiverzeichnis

/doc

PDF–Dokumentationen und Handbücher von n*Vidia* .

`CUDA_Reference_Manual_2.3.pdf` Referenzhandbuch [10]

`NVIDIA_CUDA_BestPracticesGuide_2.3.pdf` Tipps und Tricks zur Programmierung von CUDA–Anwendungen [8]

`NVIDIA_CUDA_Programming_Guide_2.3.pdf` Programmierhandbuch [9]

`NVIDIAFermiComputeArchitectureWhitepaper.pdf` Präsentation der neuen Fermi–Architektur [11]

`NVIDIA_FermiTuningGuide.pdf` Tipps und Tricks zur effizienten Programmierung der Fermi–Architektur [12]

/implementations/gputest

Quellcodes für ein Testprogramm, das alle verfügbaren CUDA–fähigen Grafikkarten und deren Eigenschaften auflistet.

`compile` Kompiliert den Quellcode (Shell–Skript)

`gputest.cu` Quellcode des Testprogramms

/implementations/helloworld

Quellcodes für das "Hello World"–Testprogramm aus Kapitel 3.1.

`compile` Kompiliert den Quellcode (Shell–Skript)

`helloworld.cu` Quellcode des "Hello World"–Programms

/implementations/kernel_optimization

Quellcodes für ein Benchmarkprogramm, verwendet für die Laufzeitmessungen aus Kapitel 4.

`compile` Kompiliert den Quellcode (Shell–Skript)

`optimization.cu` Quellcode des Benchmarkprogramms

/implementations/mpc_swarm

Quellcodes der Implementation der Schwarmsteuerung aus Kapitel 8.4.

`compile` Kompiliert den Quellcode (Shell–Skript)

`cpuswarmkernel.cpp` Zur GPU äquivalente Implementation der Kernels für die CPU

`cpuswarmkernel.h` Headerfile der Aufrufmethoden in `cpuswarmkernel.cpp`

`cudaswarmkernel.cu` Implementation der Device Kernels für die GPU inkl. Aufrufmethoden als Host Code

`cudaswarmkernel.h` Headerfile der Aufrufmethoden in `cudaswarmkernel.cu`

`main.cpp` Hauptprogramm zur Berechnung der Schwarmanimationen und Bestimmung der Rechenzeit von zehn MPC Schritten

`model.h` Headerfile mit der Definition des Modells

`mpc.cpp` Klasse des MPC–Algorithmus

`mpc.h` Headerfile der Klassendefinition in `mpc.cpp`

`swarm.cpp` Klasse zur Kapselung der Kernel–Aufrufmethoden

`swarm.h` Headerfile der Klassendefinition in `swarm.cpp`

`swarmtest.cpp` Hauptprogramm zur Speedupmessung der Kernels

`types.h` Headerfile mit der Definition des Gleitkommatyps

/implementations/pde_rock4

Quellcodes der Implementation des ROCK4–Algorithmus aus Kapitel 6.3.

`compile` Kompiliert den Quellcode (Shell–Skript)

`cpurock4kernel_constants.h` Headerfile mit der Definition der Koeffizienten des ROCK4–Algorithmus für die CPU

`cpurock4kernel.cpp` Zur GPU äquivalente Implementation der Kernels für die CPU

`cpurock4kernel.h` Headerfile der Aufrufmethoden in `cpurock4kernel.cpp`

`cudarock4kernel_constants.h` Headerfile mit der Definition der Koeffizienten des ROCK4–Algorithmus für die GPU

`cudarock4kernel.cu` Implementation der Device Kernels für die GPU inkl. Aufrufmethoden als Host Code

`cudarock4kernel.h` Headerfile der Aufrufmethoden in `cudarock4kernel.cu`

`main.cpp` Hauptprogramm für die Berechnung einer Lösung mit Speedupmessung

`rock4.cpp` Klasse zur Kapselung der Kernel–Aufrufmethoden

`rock4.h` Headerfile der Klassendefinition in `rock4.cpp`

`types.h` Headerfile mit Typdefinitionen

/implementations/thirdparty

Quellcodes für Ipopt–Interface und Laufzeitmessung, entnommen aus dem NMPC–Projekt[3].

`btipopt.cpp` Wrapperklasse für den Ipopt–Optimierer

`btipopt.h` Headerfile der Klassendefinition in `btipopt.cpp`

`minprog.cpp` Klasseninterface für Optimierer–Klassen

`minprog.h` Headerfile der Klassendefinition in `minprog.cpp`

`minprogtypedefs.h` Typdefinitionen für die Klassen in `minprog.cpp` und `btipopt.cpp`

`nlpproblem.cpp` Problemklasse für den Ipopt–Optimierer

`nlpproblem.h` Headerfile der Klassendefinition in `nlpproblem.cpp`

`rtclock.cpp` Klasse für präzise Laufzeitmessungen

`rtclock.h` Headerfile der Klassendefinition in `rtclock.cpp`

/ipopt

Linux–Quellcodepaket des Ipopt–Optimierers[4].

`Ipopt-3.8.1.tgz` Quellcodearchiv

/nvidia

Linux–Installationspakete (SUSE 11.1, 64 Bit) des CUDA–Treibers und der Runtime Library (siehe Kapitel 1.2).

`cudadriver_2.3_linux_64_190.18.run` Ausführbare Datei zur Installation des CUDA–Treibers

`cudatoolkit_2.3_linux_64_suse11.1.run` Ausführbare Datei zur Installation der Runtime Library

[3] http://www.nonlinearmpc.com (12.05.2010).
[4] https://projects.coin-or.org/Ipopt (12.05.2010).

/swarmanimations

Animierte Sequenzen der Schwarmkontrolle mit unterschiedlichen Objektanzahlen aus Kapitel 8.5.3. Zum Abspielen wird ein H.264–Videocodec benötigt.

`swarm_obj128_horiz5_discr002.avi` Animation einer Schwarmkontrolle mit $N = 5$, $T = 0.02$ und $M = 128$

`swarm_obj256_horiz5_discr002.avi` Animation einer Schwarmkontrolle mit $N = 5$, $T = 0.02$ und $M = 256$

`swarm_obj32_horiz5_discr002.avi` Animation einer Schwarmkontrolle mit $N = 5$, $T = 0.02$ und $M = 32$

`swarm_obj64_horiz5_discr002.avi` Animation einer Schwarmkontrolle mit $N = 5$, $T = 0.02$ und $M = 64$

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Glossar

Programmieren mit CUDA

Aktiver Threadblock: Ein Threadblock, der im Moment zusammen mit anderen aktiven Threadblöcken auf einem Multiprozessor bearbeitet wird

Aktiver Warp/Thread: Ein Warp bzw. Thread, der zu einem aktiven Threadblock gehört

Atomic Functions: Funktionen, die während der Ausführung den zu bearbeitenden Speicherbereich vor Zugriffen anderer Threads schützen

Block: Synonym für *Threadblock*

Blockgitter: Zweidimensionale Anordnung von Threadblöcken

Broadcasting: Gleichzeitiger Lesezugriff von mehreren Threads auf die gleiche Adresse im Shared Memory wird zu einem Zugriff zusammmengefasst

Bruttolaufzeit: Reine Laufzeit eines Device Kernels ohne zusätzliche Verzögerungen wie z.B. Treiberaufruf

Coalesced Memory Access: 16 Speicherzugriffe eines Half–Warps werden zu einem einzelnen "verschmolzen"

Compute Capabilities: Fähigkeiten der Grafikkarten, siehe Kapitel 2.6

Constant Memory: Kleiner Speicher mit Cache für konstante Werte, während der Laufzeit des gesamten Programms verfügbar

CUDA–Toolkit: Installierbares Paket, beinhaltet *Runtime Library* und den Compiler *nvcc*

Device Code: Quellcode, der ausschließlich für die GPU als Device Kernel kompiliert wird

Device Kernel: Für die GPU kompiliertes Programm, im Quellcode mit `__global__` markierte Methode

Device Memory: Hauptspeicher der Grafikkarte, wird auch als "Grafikspeicher" bezeichnet, ist während der gesamten Programmlaufzeit verfügbar

Grid: In der Literatur gebräuchlicher Begriff für *Blockgitter*

Half–Warp: Die ersten oder letzten 16 Threads eines Warps

Host Code: Quellcode, der im herkömmlichen Sinne für die CPU kompiliert wird

Intrinsic Functions: Mathematische Funktionen, die direkt per Prozessorbefehl von der GPU berechnet werden können

Kernelgröße: Begriff zur Umschreibung der Menge an Ressourcen (Register, Shared Memory), die ein Device Kernel zur Ausführung benötigt

Local Memory: Auslagerungsspeicher im *Device Memory* für Threads, falls Register nicht ausreichen

Multiprozessor: Ein physischer Prozessorkern einer GPU, der aus acht SIMD–Prozessoren besteht

Nettolaufzeit: Laufzeit eines Device Kernels inkl. Treiberaufrufdauer und Hintergrundprozessen

Pinned Memory: Speziell gekennzeichneter Bereich im CPU–Speicher, auf den die GPU zugreifen kann

Register: Kleine 4–Byte–Speichereinheit eines Multiprozessors, während der Laufzeit eines einzelnen Threads verfügbar

Runtime Library: Bibliothek von *n*Vidia , die Methoden der Treiber–API kapselt, Teil des CUDA–Toolkits

Shared Memory: Schneller, kleiner, lokaler Speicher eines Multiprozessors, ist während der Ausführung eines Threadblocks verfügbar

Threadblock: Gruppe von maximal 512 Threads, die auf einem Multiprozessor gleichzeitig mit Möglichkeit der Synchronisierung ausgeführt werden

Treiber–API: Methoden des Grafikkartentreibers zur GPU–Steuerung

Warp: 32 Threads eines Threadblocks, die nach dem SIMD–Prinzip absolut synchron abgearbeitet werden

Modellprädiktive Regelung

$\bar{\Phi}$ Modellfunktion eines Schwarmobjekts

Φ Modellfunktion des zu kontrollierenden Systems

d Minimale Ortsdistanz zwischen zwei Objekten des Beispielschwarms

F Feedbackfunktion zur Berechnung einer Kontrolle ausgehend von einem Systemzustand

g	Funktion zur Beschreibung expliziter Beschränkungen der Kontrollsequenzen
$\bar{g}_1$	Funktion zur Beschreibung nichtlinearer Restriktionen von Objektpaaren
$\bar{g}_2$	Funktion zur Beschreibung nichtlinearer Restriktionen eines einzelnen Objekts
Dg, Δg	Jacobimatrix von g, bzw. Gradient von g bei eindimensionalem Bildraum
$\tilde{g}$	Funktion zur Beschreibung nichtlinearer Beschränkungen der Systemzustände
$\bar{J}_N$	Zielfunktion eines einzelnen Objekts, Summe der Kostenfunktionen $\bar{l}$ und der Endkostenfunktion $\bar{L}$
ΔJ	Gradient der Zielfunktion J_N
J_N	Zielfunktion, Summe der Kostenfunktionen l und der Endkostenfunktion L
k	Nummer der aktuellen Abtastung
L	Endkostenfunktion, wird im Horizontschritt N ausgewertet
l	Kostenfunktion, wird in den Horizontschritten $0, \ldots, N-1$ ausgewertet
$\bar{L}$	Endkostenfunktion eines einzelnen Objekts, wird im Horizontschritt N ausgewertet
$\bar{l}$	Kostenfunktion eines einzelnen Objekts, wird in den Horizontschritten $0, \ldots, N-1$ ausgewertet
M	Anzahl der Objekte eines Schwarms
m	Dimension der Systemkontrolle
$\bar{m}$	Dimension der Systemkontrolle eines Schwarmobjekts
N	Länge des Kontrollhorizonts
n	Dimension des Systemzustands
$\bar{n}$	Dimension des Systemzustands eines Schwarmobjekts
p	Anzahl der Komponenten des Systemzustands, die einem Thread zugeteilt sind
r	Dimension des Bildraums von g
r_1	Dimension des Bildraums von $\bar{g}_1$
r_2	Dimension des Bildraums von $\bar{g}_2$
r'	Dimension des Bildraums von $\tilde{g}$
T	Abstand zwischen zwei Abtastzeitpunkten

$\mathbb{U}$	Menge zulässiger Kontrollen
$\mathcal{U}_N$	Menge aller zulässigen Kontrollsequenzen mit Horizont N
$\bar{u}^u, \bar{u}^o$	Untere und obere Schranken der Kontrollbeschränkungen eines einzelnen Objekts
$u(k)$	Feedbackkontrolle zum Abtastzeitpunkt k
u_N	Kontrollsequenz mit Horizont N
$u_N(i)$	i–te Kontrolle der Kontrollsequenz u_N, $i = 0, \ldots, N-1$
$u_N^{(j)}$	Kontrollsequenz des Objekts j mit Horizont N
$u_N^{(j)}(i)$	i–te Kontrolle der Kontrollsequenz $u_N^{(j)}$, $i = 0, \ldots, N-1$ des Objekts j mit Horizont N
$\hat{u}_N$	Optimale Kontrollsequenz
u^u, u^o	Untere und obere Schranken der Kontrollbeschränkungen
v_{max}	Maximaler Geschwindigkeitsbetrag eines Objekts des Beispielschwarms
$\mathbb{X}$	Menge zulässiger Systemzustände
$\bar{x}^u, \bar{x}^o$	Untere und obere Schranken der Systemzustände der einzelnen Objekte
$x^{(j)}$	Systemzustand des j–ten Objekts, im Schwarmzustand x angeordnet
$x(k)$	Systemzustand zum Abtastzeitpunkt k
$\bar{x}^{(ref)}$	Referenzpunkt zu dem ein Beispielschwarm gesteuert werden soll
x^u, x^o	Untere und obere Schranken der Systemzustände
$x_{u_N}(i, x)$	Prädiktion des Zustands x nach i Zeitschritten, basierend auf der Kontrollsequenz u_N
$\bar{x}_{u_N^{(j)}}\left(i, x^{(j)}\right)$	Prädiktion des Zustands von Objekt j nach i Zeitschritten, basierend auf der entsprechenden Objektkontrollsequenz $u_N^{(j)}$

Parabolische partielle Differentialgleichungen

$\hat{a}_{i,j}$	Koeffizienten des Runge–Kutta–Tableaus der Runge–Kutta–Abbildung W
$\tilde{a}_{i,j}$	Koeffizienten des Runge–Kutta–Tableaus der Runge–Kutta–Abbildung P
$a_{i,j}$	Koeffizienten eines Runge–Kutta–Tableaus

$\alpha^{ij}(x,t)$	Koeffizient des Differentialoperators für die zweiten Ableitungen zur Zeit t am Ort x
$\beta^{i}(x,t)$	Koeffizient des Differentialoperators für die ersten Ableitungen zur Zeit t am Ort x
$\bar{b}_i$	Koeffizienten des Runge–Kutta–Tableaus der eingebetteten Methode $\bar{W}$
$\hat{b}_i$	Koeffizienten des Runge–Kutta–Tableaus der Runge–Kutta–Abbildung W
$\tilde{b}_i$	Koeffizienten des Runge–Kutta–Tableaus der Runge–Kutta–Abbildung P
b_i	Koeffizienten eines Runge–Kutta–Tableaus
ε	Fehlerkorrekturfaktor der Schrittweitensteuerung
$f(\hat{y})$	Autonome rechte Seite eines gewöhnlichen Differentialgleichungssystem mit Zustand $\hat{y}$.
$\tilde{f}(x,t,y)$	Rechte Seite einer partiellen Differentialgleichung zur Zeit t am Ort x mit Zustand y
G	Ortsdiskretisierung eines Ortsgebiets Ω
Γ	Rand von Ω
$\gamma^{i}(x,t)$	Koeffizient des Differentialoperators für den linearen Anteil zur Zeit t am Ort x
g_i	Ergebnis der Rekursionsformel zur Auswertung von $P(\hat{h},\hat{y})$ nach dem i–ten Rekursionsschritt
$\hat{h}$	Schrittweite eines Runge–Kutta–Verfahrens
h_i	Abstand zweier benachbarteter Diskretisierungspunkte von G in der i–ten Komponente
κ_i	Koeffizienten für die Auswertung der Runge–Kutta–Abbildung $P(\hat{h},\hat{y})$ mittels Rekursionsformel
$\hat{k}_i$	Hilfsgrößen bei der Auswertung eines Runge–Kutta–Tableaus
k_i	Index eines Diskretisierungspunkts von G in der i–ten Komponente
L	Differentialoperator zweiter Ordnung
μ_i	Koeffizienten für die Auswertung der Runge–Kutta–Abbildung $P(\hat{h},\hat{y})$ mittels Rekursionsformel
n	Dimension des gewöhnlichen Differentialgleichungssystems erster Ordnung nach der Diskretisierung mit der Methode der finiten Differenzen

N_i	Anzahl der äquidistanten Diskretisierungspunkte einer Ortsdiskretisierung G in der i–ten Komponente
ν_i	Koeffizienten für die Auswertung der Runge–Kutta–Abbildung $P(\hat{h}, \hat{y})$ mittels Rekursionsformel
Ω	Offenes Beschränktes Ortsgebiet, Teilmenge im $\mathbb{R}^p$
p	Dimension des Ortsgebiets
$P(\hat{h}, \hat{y})$	$(s-4)$–stufige Runge–Kutta–Abbildung mit Startzustand $\hat{y}$ und Schrittweite $\hat{h}$
$r(x)$	Startzustand eines Anfangs–/Randwert–Problems am Ort x
s	Stufen eines Runge–Kutta–Tableaus
T	Zeitliche Beschränkung bei der Betrachtung einer Lösung y
t	Zeitpunkt einer Lösung
$W(\hat{h}, \hat{y})$	Vierstufige Runge–Kutta–Abbildung mit Startzustand $\hat{y}$ und Schrittweite $\hat{h}$
w_i	Seitenlänge eines rechteckigen Ortsgebietes in der i–ten Komponente
$WP(\hat{h}, \hat{y})$	s–stufige kaskadierte Runge–Kutta–Abbildung $WP(\hat{h}, \hat{y}) = W(\hat{h}, P(\hat{h}, \hat{y}))$
x	Punkt im Ortsgebiet Ω
$y(x,t)$	Lösung einer partiellen Differentialgleichung zur Zeit t am Ort x
$y_{k_1,\ldots,k_p}(t)$	Wert der Lösung y einer partiellen Differentialgleichung am Punkt $x = \sum_{i=1}^{p}(k_i+1)h_i \cdot e_i$
$\hat{y}(t)$	Lösung des gewöhnlichen Differentialgleichungssystems erster Ordnung nach der Diskretisierung mit der Methode der finiten Differenzen zum Zeitpunkt t

Stichwortverzeichnis

Literaturverzeichnis

[1] Abdulle, A.: Fourth Order Chebyshev Methods With Recurrence Relation. In: *SIAM Journal on Scientific Computing* 23 (2001-2002), Nr. 6, S. 2041–2054

[2] Grama, A. ; Gupta, A. ; Karypis, G. ; Kumar, V.: *Introduction to Parallel Computing*. Pearson Education Ltd., 2003

[3] Grüne, L.: Analysis and design of unconstrained nonlinear MPC schemes for finite and infinite dimensional systems. In: *SIAM Journal on Control and Optimization* 48 (2009), S. 1206–1228

[4] Hairer, E. ; Wanner, G.: *Solving Ordinary Differential Equations II*. Springer-Verlag Berlin Heidelberg, 2002 (Springer Series in Computational Mathematics 14)

[5] Knabner, P. ; Angermann, L.: *Numerik partieller Differentialgleichungen*. Springer-Verlag Berlin Heidelberg, 2000

[6] Mayne, D. Q. ; Rawlings, J. B. ; Rao, C. V. ; Scokaert, P. O. M.: Constrained model predictive control: Stability and optimality. In: *Automatica* 36 (2000), Nr. 6, S. 789–814

[7] Nocedal, J. ; Wright, S. J.: *Numerical Optimization*. Springer Science+Business Media,LLC, 2006

[8] NVIDIA: *NVIDIA C Programming Best Practices Guide*. Juli 2009. – Version 2.3 – PDF–Datei auf CD: /doc/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[9] NVIDIA: *NVIDIA CUDA Programming Guide*. August 2009. – Version 2.3.1 – PDF–Datei auf CD: /doc/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[10] NVIDIA: *NVIDIA CUDA Reference Manual*. Juli 2009. – Version 2.3 – PDF–Datei auf CD: /doc/CUDA_Reference_Manual_2.3.pdf

[11] NVIDIA: *Whitepaper, NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009. – PDF–Datei auf CD: /doc/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[12] NVIDIA: *Tuning CUDA Applications for Fermi*. Februar 2010. – Version 1.0 – PDF–Datei auf CD: /doc/NVIDIA_FermiTuningGuide.pdf

[13] Pannek, J.: *Receding Horizon Control: A Suboptimality–based Approach*, University of Bayreuth, Diss., 2009

[14] Schwandt, H.: *Parallele Numerik*. Teubner Verlag, 2003

[15] Törnig, W. ; Gipser, M. ; Kaspar, B.: *Mathematische Methoden in der Technik*. Bd. 1: *Numerische Lösung von partiellen Differentialgleichungen der Technik*. Teubner, 1985

[16] Tröltzsch, F.: *Optimale Steuerung partieller Differentialgleichungen. Theorie, Verfahren und Anwendungen*. Wiesbaden: Vieweg, 2005. – x+297 S.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Bayreuth, den 14. Mai 2010

(Thomas Jahn)