Effiziente Implementierung und Analyse eines Waveform-Relaxationsverfahrens mit NVIDIA CUDA

Alexander Fiebig

Bayreuth Reports on Parallel and Distributed Systems

No. 9, December 2016

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Bachelorarbeit

Effiziente Implementierung und Analyse eines Waveform-Relaxationsverfahrens mit NVIDIA CUDA

Efficient implementation and analysis of a Waveform-Relaxation method using NVIDIA CUDA

Alexander Fiebig

Bayreuth, am 16.12.2016

Universität Bayreuth
Angewandte Informatik 2

Betreuer: PD Dr. Matthias Korch, Tim Werner, M.Sc.
Prüfer: Prof. Dr. Thomas Rauber, PD Dr. Matthias Korch

Zusammenfassung

Das Waveform-Relaxationsverfahren (WR-Verfahren) ist ein numerisches Lösungsverfahren für Anfangswertprobleme (IVP) gewöhnlicher Differentialgleichungssysteme (ODE-Systeme). Es besitzt einen hohen Grad an Parallelität und eignet sich besonders für schwach gekoppelte ODE-Systeme.

In dieser Arbeit werden parallele Implementierungen des WR-Verfahrens auf GPUs betrachtet. Es werden zunächst allgemeine und problemspezifische Implementierungen für eine GPU vorgestellt. Anschließend werden darauf aufbauend Implementierungen für mehrere GPUs innerhalb eines Rechners und zum Schluss Implementierungen für verteilten Adressraum, welche alle GPUs eines Rechnernetzes nutzen, entwickelt. Die entstandenen Implementierungen werden hinsichtlich Rechenzeit, Speicherbedarf und Konvergenzverhalten untersucht und mit dem Eulerverfahren verglichen.

Abstract

The Waveform-Relaxation (WR) method is a numerical method for solving initial value problems (IVP) of systems of ordinary differential equations (ODE). It has a high degree of parallelism and is especially well suited for weakly coupled ODE-systems.

In this thesis, parallel implementations of the WR-method on GPUs are presented. First, general and problem specific implementations for one GPU are shown. Then, built on top of that, implementations for all GPUs in one computer and implementations for distributed memory that use all GPUs in a computer network are developed. The resulting implementations are analysed based on runtime, memory consumption and convergence behaviour and are compared with the Euler method.

Inhaltsverzeichnis

1 Einleitung

Viele physikalisch-technische Probleme lassen sich mit einem System von gewöhnlichen Differentialgleichungen (ODEs) beschreiben. Ein Beispiel ist die Simulation elektrischer Schaltungen im VLSI-Bereich. Dabei müssen ODE-Systeme gelöst werden, welche das zeitliche Verhalten mehrerer unbekannten Größen (z.B. verschiedene Spannungen) beschreiben. Zur Lösung dieser Differentialgleichungssysteme werden häufig numerische Verfahren eingesetzt, welche, vor allem bei großen Systemdimensionen, sehr rechenaufwändig sind. Deswegen sind möglichst effiziente Lösungsverfahren notwendig.

Ein solches numerisches Verfahren zur Lösung von ODE-Systemen ist das Waveform-Relaxationsverfahren (WR-Verfahren), dessen Anwendung sich aufgrund seiner Systemparallelität besonders zum Lösen schwach gekoppelter ODE-Systeme eignet. Um diese Parallelität bestmöglich auszunutzen bedarf es entsprechender Hardware. Moderne GPUs sind für parallele Berechnungen ausgelegt und deshalb dafür besonders gut geeignet.

In der vorliegenden Arbeit werden verschiedene Implementierungen des WR-Verfahrens für GPUs vorgestellt und untersucht. Zunächst wird die Funktionsweise des WR-Verfahrens erläutert und eine kurze Einführung in CUDA gegeben. Anschließend werden die verschiedenen Implementierungen erläutert und im letzten Abschnitt auf Rechenzeit, Speicherbedarf und Konvergenzverhalten hin untersucht und mit dem Eulerverfahren verglichen.

2 WR-Verfahren

2.1 Gewöhnliche Differentialgleichungssysteme

Eine gewöhnliche Differentialgleichung (ODE) ist eine Differentialgleichung, bei der zu einer gesuchten Funktion y nur Ableitungen nach genau einer Variablen t auftreten. Kann man die Differentialgleichung nach der höchsten vorkommenden Ableitung auflösen, so heißt sie explizit. Ist $f : \mathbb{R} \times \mathbb{R}^d \longmapsto \mathbb{R}^d$ eine stetige Funktion, so ist

$$\boldsymbol{y}'(t) = f(t, \boldsymbol{y}(t)) \tag{2.1}$$

ein System von expliziten ODEs erster Ordnung der Dimension d.

Ist zudem zu einem bestimmten Zeitpunkt t_0 ein Anfangswert bekannt, so handelt es sich um ein Anfangswertproblem (IVP). In dieser Arbeit werden nur IVPs der Form

$$\boldsymbol{y}'(t) = f(t, \boldsymbol{y}(t)), \quad \boldsymbol{y}(t_0) = \boldsymbol{y_0} \tag{2.2}$$

betrachtet.

Außerdem ist erwähnenswert, dass sich ein ODE-System beliebiger Ordnung n der Dimension m immer in ein ODE-System erster Ordnung der Dimension $d = n \cdot m$ überführen lässt. Dazu führt man n von einander abhängige Funktionen $\boldsymbol{y_1}, \boldsymbol{y_2}, \boldsymbol{y_3}, \ldots, \boldsymbol{y_n}$ mit $\boldsymbol{y_1} = \boldsymbol{y}, \boldsymbol{y_2} = \boldsymbol{y_1'}, \boldsymbol{y_3} = \boldsymbol{y_2'}, \ldots, \boldsymbol{y_n} = \boldsymbol{y_{n-1}'}$ ein und repräsentiert diese als zusätzliche Komponenten des Ergebnisvektors.

2.2 Explizites Euler-Verfahren

Eine Möglichkeit, das IVP aus Gleichung (2.2) numerisch zu lösen, ist das explizite Euler-Verfahren. Bei diesem Verfahren werden aus dem Anfangswert $\boldsymbol{y_0}$ iterativ mit einer gegebenen Schrittweite $h > 0$ neue Lösungen zu den Zeitpunkten

$$t_i = t_0 + i \cdot h, \quad i \in \mathbb{N} \tag{2.3}$$

berechnet.

Das Ganze wird solange gemacht, bis man einen gegebenen Zeitpunkt $t_s = t_0 + H$ erreicht, wobei $H = s \cdot h$ mit $s \in \mathbb{N}$ die betrachtete Integrallänge ist. Man führt also insgesamt $s = H/h$ Schritte aus. Diese Schritte werden auch Eulerschritte genannt. Für die Berechnung der Lösung zum Zeitpunkt t_i mit $i \in [1, s]$ ist die Lösung zum Zeitpunkt t_{i-1} nötig. Daraus ergibt sich folgende Iterationsvorschrift:

$$\boldsymbol{y}(t_i) = \boldsymbol{y}(t_{i-1}) + h \cdot f(t_{i-1}, \boldsymbol{y}(t_{i-1})) \tag{2.4}$$

Eine Lösung $\boldsymbol{y}(t_i)$ ist immer vektoriell zu deuten, da es sich um eine ODE der Dimension d handelt.

2.3 WR-Algorithmus

Der WR-Algorithmus basiert auf dem Konzept der Picard-Iteration. Die Picard-Iteration ist eine Methode zur globalen Approximation der Lösung von Gleichung (2.2). Sie basiert auf dem Lösen einer Sequenz von Differentialgleichungen folgender Form [1]:

$$\boldsymbol{y}'(t)^{(w+1)} = f(t, \boldsymbol{y}(t)^{(w)}), \quad \boldsymbol{y}(t_0)^{(w+1)} = \boldsymbol{y}_0, \quad t \in [t_0, t_s] \tag{2.5}$$

Dabei kann das Problem in jeder Iteration in d unabhängige Probleme aufgeteilt werden, welche parallel mit einem Zeitschrittverfahren gelöst werden können. In dieser Arbeit wurde als Zeitschrittverfahren das explizite Eulerverfahren gewählt (siehe Abschnitt 2.2). Eine solche Iteration wird auch als WR-Schritt bezeichnet.

Allerdings ist die Konvergenz der WR-Schritte $\boldsymbol{y}(t)^{(1)}, \boldsymbol{y}(t)^{(2)}, \ldots$ hin zur Lösung $\boldsymbol{y}(t)$ sehr langsam, da zur Berechnung von $\boldsymbol{y}'(t)^{(w+1)}$ ausschließlich Werte der Systemkomponenten aus dem letzten WR-Schritt verwendet werden.

2.3.1 Gauß-Seidel- und Jacobi-WR-Methoden

Eine Möglichkeit, die Konvergenz zu verbessern, ist, die klassische Picard-Iteration, definiert in Gleichung (2.5), leicht abzuwandeln und die zum Lösen linearer Gleichungssysteme bekannten Gauß-Seidel- und Jacobi-Iterationsschemata zu verwenden. So ist die Gauß-Seidel-WR-Methode als

$$y_j'(t)^{(w+1)} = f_j(t, y_1(t)^{(w+1)}, \ldots, y_j(t)^{(w+1)}, y_{j+1}(t)^{(w)}, \ldots, y_d(t)^{(w)}), \quad j \in [1, d] \tag{2.6}$$

und die Jacobi-WR-Methode als

$$y_j'(t)^{(w+1)} = f_j(t, y_1(t)^{(w)}, \ldots, y_j(t)^{(w+1)}, y_{j+1}(t)^{(w)}, \ldots, y_d(t)^{(w)}), \quad j \in [1, d] \tag{2.7}$$

definiert [1].

Die Gauß-Seidel WR-Methode hat die bessere Konvergenz, ist aber sequentieller Natur, da hier zur Berechnung von $y_j'(t)^{(w+1)}$ die Werte der Systemkomponenten $y_1(t)^{(w+1)}$, $\ldots, y_j(t)^{(w+1)}$ bekannt sein müssen. Dagegen ist die Jacobi WR-Methode paralleler Natur, denn es wird nur der Wert von $y_j(t)^{(w+1)}$ aus dem aktuellen WR-Schritt benötigt. Für alle anderen Systemkomponenten werden bereits bekannte Werte aus dem letzten WR-Schritt verwendet. Dafür ist bei der Jacobi WR-Methode die Konvergenz wieder deutlich schlechter, d.h. es werden mehr WR-Schritte benötigt, um die gleiche Genauigkeit zu erreichen.

2.3.2 Block-Jacobi-WR-Methode

Bei der Block-Jacobi-WR-Methode handelt es sich um eine Verallgemeinerung der Jacobi-WR-Methode. Eine Block-Jacobi-WR-Methode mit Blockgröße $1 \leq b \leq d$ ist als

$$\begin{aligned} &y_j'(t)^{(w+1)} = f_j(t, y_1(t)^{(w)}, \ldots, y_m(t)^{(w+1)}, \ldots, y_{m+b-1}(t)^{(w+1)}, y_{m+b}(t)^{(w)}, \ldots, y_d(t)^{(w)}) \\ &m \in [1, d-b+1], \quad j \in [m, m+b-1] \end{aligned} \tag{2.8}$$

definiert.

Im Fall $b = 1$ ist das die normale Jacobi WR-Methode aus Gleichung (2.7). Damit kann durch Variation der Blockgröße ein Kompromiss zwischen Parallelität und Konvergenz gefunden werden [8].

2.3.3 Windowing

Eine weitere Möglichkeit, die Konvergenz zu verbessern, ist die Aufteilung des Integrationsintervalls in mehrere sogenannte Windows. Auf jedem dieser Windows wird dann der WR-Algorithmus angewandt. Dabei verläuft die Berechnung der Windows sequentiell, da zur Berechnung von Window $i+1$ die Ergebnisse aus Window i verwendet werden [1].

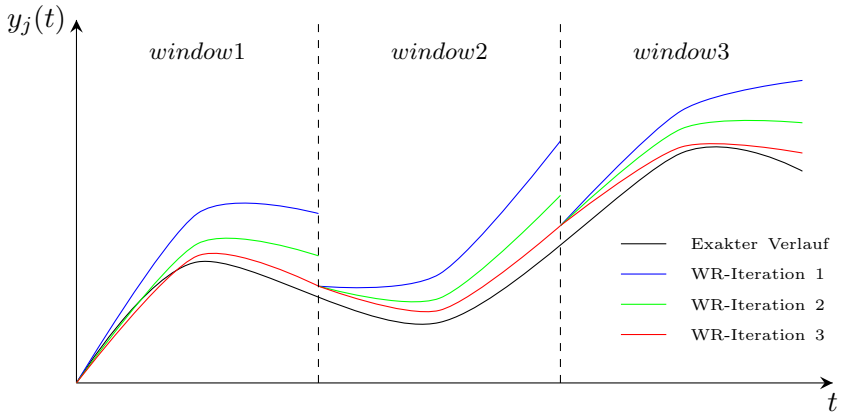

Abbildung 2.1: Beispielhafter Berechnungsverlauf einer Systemkomponente j

3 GPU-Architektur und CUDA

Da für die Implementierung des WR-Verfahrens CUDA verwendet wird, werden in diesem Kapitel kurz die Grundlagen der GPU-Programmierung mit CUDA erläutert. Dieses Kapitel ist stark angelehnt an den CUDA C Programming-Guide [6]. CUDA ist eine Programmier-Plattform für GPGPU, also dem Durchführen von allgemeinen Berechnungen auf der GPU, auch Device genannt.

Der Einsatz von CUDA beschränkt sich auf NVIDIA-GPUs, was aber auch den Vorteil hat, dass spezifische Eigenschaften von NVIDIA-GPUs bestmöglich ausgenutzt werden können. CUDA erlaubt es Nutzern, C mit einigen speziellen Erweiterungen als höhere Programmiersprache für die Programmierung der GPU zu benutzen.

3.1 Vergleich von CPU und GPU

Die GPU ist ein massiv paralleler Manycore-Prozessor, der sich durch sehr große Rechenleistung und Speicherbandbreite auszeichnet. Auf einer GPU stehen deutlich mehr Transistoren für die Datenverarbeitung als auf einer CPU zur Verfügung (siehe Abbildung 3.1), da viele Transistoren auf CPUs für Caching und Kontrolllogik verwendet werden.

Daher eignet sich die GPU besonders für Berechnungen, in welchen dasselbe Programm auf unterschiedlichen Daten parallel ausgeführt werden muss (SIMD). Das Ausführen desselben Programms für jedes Datenelement erlaubt es, Speicherzugriffslatenzen durch Berechnungen zu überdecken. Daher benötigt die GPU, anders als die CPU, nicht so große Daten-Caches.

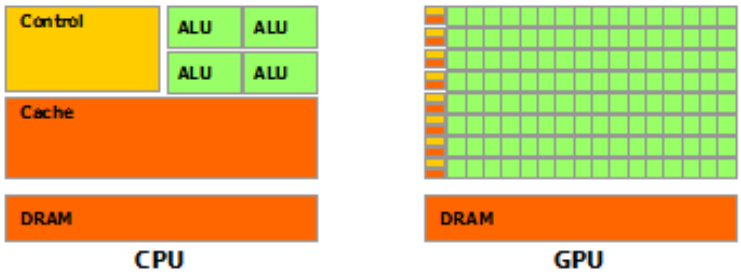

Abbildung 3.1: Transistorverteilung auf CPU und GPU (Quelle: [6])

Grund für das unterschiedliche Design von CPU und GPU sind verschiedene Ziele:

- Ziel der CPU ist es die Latenz für einen Thread zu minimieren
- Ziel der GPU ist es den Durchsatz aller Threads zu maximieren

3.2 NVIDIA-GPU-Architektur

Im Folgenden wird kurz auf die Architektur einer modernen NVIDIA-GPU eingegangen. Im Kern besteht eine solche GPU aus einer Menge von Streaming Multiprozessoren (SMs). Der Aufbau eines solchen SM (siehe Abbildung 3.2) ist abhängig von der Architektur und die Anzahl der SMs ist abhängig vom jeweiligen Modell.

In dieser Arbeit wurden primär GPUs mit Maxwell-Architektur der zweiten Generation verwendet (Compute-Capability 5.2). Bei dieser Architektur besitzt jeder SM u.a. folgende Komponenten [3]:

- 96 KB Scratchpad-Speicher (NVIDIA sagt dazu Shared-Memory)
- 24 KB kombinierten L1- und Texture-Cache
- Vier Warp-Scheduler, die pro Warp jeden Taktzyklus zwei Instruktionen ausgeben können
- Vier Blöcke mit je 32 CUDA-Cores, welche die Gleitkomma-Operationen ausführen

3.3 CUDA-Programmiermodell

Das CUDA-Programmiermodell besteht aus drei Abstraktionsebenen mit denen ein datenparalleles GPU-Programm, auch Kernel genannt, beschrieben wird. Auf der obersten Ebene steht das Grid, welches von einem Kernel ausgeführt wird. Ein Grid wird ein- zwei- oder dreidimensional in Threadblöcke unterteilt, welche selbst wiederum ein- zwei- oder dreidimensional in Threads unterteilt werden (siehe Abbildung 3.3). Jeder Threadblock hat einen Index innerhalb des Grid und jeder Thread hat einen lokalen Index innerhalb seines Threadblocks. Daraus lässt sich dann auch leicht der globale Index eines Threads innerhalb des Grid bestimmen. Diese Indices können im Kernel zur Laufzeit abgefragt werden, womit ein vom ausführenden Thread abhängiger, dynamischer Kontrollfluss ermöglicht wird.

Threads innerhalb eines Threadblocks können Daten über Shared-Memory austauschen und mithilfe von Barriers synchronisiert werden. Für Threads aus unterschiedlichen Threadblöcken ist jedoch keines von beidem möglich.

3.3.1 Kernelausführung

Wenn ein Hostprogramm ein Kernel-Grid startet, werden die Threadblöcke des Grid auf die verschiedenen SMs (siehe Abschnitt 3.2) aufgeteilt. Alle Threads eines Threadblocks werden auf einem SM parallel ausgeführt. Ebenso können mehrere Threadblöcke parallel auf einem SM ausgeführt werden. Wenn ein Threadblock auf einem SM abgearbeitet ist, wird ein neuer, dem SM zugeteilter Threadblock ausgeführt.

Abbildung 3.2: Aufbau eines SM der zweiten Maxwell-Generation (Quelle: [3])

Vor der Ausführung eines Threadblocks teilt der SM alle Threads des Threadblocks in Gruppen von je 32 Threads, Warps genannt, auf. Jeder Warp wird dann von einem Warp-Scheduler für die Ausführung eingereiht. Alle Threads eines Warps führen immer dieselbe Instruktion aus. Um die volle Effizienz zu erreichen, müssen daher alle 32 Threads eines Warps denselben Codepfad haben. Ist dies nicht der Fall, führt der Warp sequentiell jede Verzweigung aus und deaktiviert dabei jeweils die Threads, die die Verzweigung nicht nehmen (Warpdivergenz).

Für die Performance eines Kernels ist es daher wichtig, Warpdivergenz bestmöglich zu vermeiden. Ebenso sollte die Threadblockgröße möglichst immer ein Vielfaches der Warpgröße sein, da sonst ein Warp pro Threadblock $threads\_per\_block \mod 32$ Threads komplett deaktivieren muss.

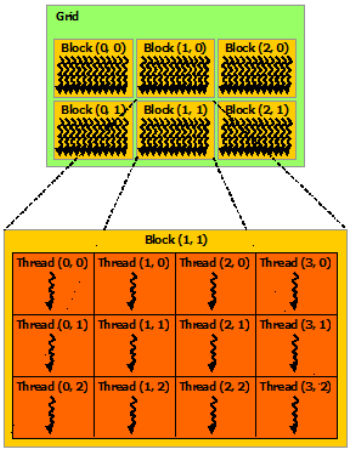

Abbildung 3.3: Beispiel einer 2D-Grid mit 2D-Threadblöcken (Quelle: [6])

3.3.2 Speichermodell

CUDA-Threads können auf verschiedene Speicherbereiche während der Ausführung zugreifen. So hat jeder Thread Registerspeicher für automatische Variablen in Registern und lokalen Speicher für automatische Variablen auf dem Stack, für welche keine Register mehr zur Verfügung stehen (Registerspilling).

Jeder Threadblock verfügt über Shared-Memory, auf welchen jeder Thread des Threadblocks Zugriff hat und über den Daten zwischen den Threads des Threadblocks ausgetauscht werden können. Shared-Memory besitzt eine hohe Bandbreite und eine geringe Latenz gegenüber lokalem und globalem Speicher. Zudem ist Shared-Memory in sogenannte Bänke unterteilt, welche ein spezielles Zugriffsmuster benötigen, da es sonst zu Bankkonflikten kommt und die Speicherzugriffe sequentialisiert werden.

Alle Threads einer Grid haben Zugriff auf den globalen Speicher. Der globale Speicher dient der Speicherung von persistenten Daten über mehrere Kernelaufrufe hinweg. Er wird etwa vom Hostprogramm für Device-Allokationen und Kopieroperationen zwischen Host und Device verwendet. Auf den globalen Speicher wird in Blöcken zugegriffen, welche an Vielfachen von 32/64/128 Byte beginnen. Es wird immer ein kompletter Block geladen, selbst wenn nur ein einzelnes Speicherwort des Blocks gelesen wird (Coalescing). Je mehr ungenutzte Speicherworte geladen werden müssen, desto geringer ist der Befehlsdurchsatz und desto mehr Bandbreite geht verloren.

Zusätzlich gibt es noch Textur- und Konstantenspeicher, welcher wie der globale Speicher persistent ist und auf den von allen Threads einer Grid zugegriffen werden kann. Da dieser aber in der Implementierung nicht explizit benutzt wurde, wird hier nicht näher darauf eingegangen.

3.3.3 Streams

Ein Hostprogramm kann für jede GPU mehrere Streams erstellen. Ein Stream ist eine Sequenz von Befehlen an die GPU, welche von dieser nach dem First-Come-First-Serve Prinzip sequentiell abgearbeitet werden. Die Befehle werden von potentiell mehreren verschiedenen Host-Threads in einen Stream eingereiht. Operationen in verschiedenen Streams können auf der GPU parallel ausgeführt werden. Das ermöglicht etwa das parallele Ausführen von Kernels, falls diese in verschiedenen Streams gestartet werden.

Für diese Arbeit besonders interessant ist aber, dass mithilfe von Streams eine Überlappung von Kernelausführung und Datenaustausch mit der GPU ermöglicht wird. Falls beim Datenaustausch Host-Speicher beteiligt ist, muss dieser, damit eine Überlappung stattfinden kann, gepinnt sein. Bei gepinntem Speicher ist sichergestellt, dass dieser im RAM gespeichert ist und somit kein Seitenfehler beim Anfordern des Speichers auftreten kann.

3.3.4 Events

Streams müssen sowohl untereinander, als auch mit dem Hostprogramm synchronisiert werden können. Das geschieht mithilfe von Events, die im Folgenden beschrieben werden. Ein Event wird vom Hostprogramm erstellt und in einen Stream eingereiht. Sobald alle Operationen im Stream, die vor dem Event eingereiht wurden, abgearbeitet sind, tritt das Event ein. Andere Streams oder das Hostprogramm können nun auf das Eintreten des Events warten und so mit dem Stream des Events synchronisieren. Ein Stream, der auf das Eintreten eines Events in einem anderen Stream wartet, muss nicht auf demselben Device wie der Stream des Events sein. Auf diese Weise lassen sich daher auch Devices untereinander synchronisieren.

Events können zudem auch für präzise Zeitmessung verwendet werden. Dazu wird zu Beginn und zum Ende der Zeitmessung ein Event in den Stream eingereiht und auf das Eintreten der beiden Events gewartet. Anschließend kann die Zeitdifferenz zwischen dem Eintreten des ersten und des zweiten Events bestimmt werden.

4 Implementierung

Es sind im Rahmen dieser Bachelorarbeit verschiedene Varianten des WR-Verfahrens mithilfe von CUDA implementiert worden. Im folgenden Kapitel werden diese Implementierungen vorgestellt. Der Code wurde in die bestehende Bibliothek des Lehrstuhls integriert, in der bereits einige Testprobleme sowie ein einheitliches Interface zur Verfügung standen.

Insbesondere war in dieser Bibliothek bereits ein mit pthreads parallelisiertes WR-Verfahren vorhanden, welches im Rahmen einer vorangegangenen Bachelorarbeit (siehe [7]) entstanden ist und welches als Orientierung diente.

4.1 Bedienung und Interface

Es sind insgesamt drei CUDA-Löser, welche das WR-Verfahren verwenden, implementiert worden:

- cuWR: Verwendet nur eine GPU
- cuWR_mgpu: Verwendet alle GPUs innerhalb eines Rechners
- dm_cuWR: Verwendet alle GPUs von mehreren Rechnern mithilfe von MPI

Ausgewählt wird ein solcher Löser mit der Kommandozeilenoption `-solver:<name>`. Des Weiteren kann man für jeden Löser folgende Optionen einstellen:

```
-ode:<name>                    Testproblem welches gelöst wird.
-impl:<name>                   Implementierungsvariante die verwendet wird.
-h:<float>                     Schrittweite.
-H:<float>                     Integrationsintervall.
-wr_steps:<uint>               Anzahl der WR-Schritte.
                               Darf nicht mit -epsilon kombiniert werden.
-epsilon:<float>               Konvergenzkriterium für dynamische Anzahl an
                               WR-Schritten.
                               Darf nicht mit -wr_steps kombiniert werden.
-windows:<uint>                Anzahl der Windows.
-blocksX:<uint>                X-Komponente der Griddimension.
-blocksY:<uint>                Y-Komponente der Griddimension.
-threads_per_blockX:<uint>     X-Komponente der Blockdimension.
-threads_per_blockY:<uint>     Y-Komponente der Blockdimension.
```

Für die Grid- und Blockdimension gibt es Defaultwerte, die sicherstellen, dass für jede Systemkomponente ein Thread gestartet wird, wovon im Folgenden auch ausgegangen wird. Zudem sollte die Schrittweite nicht zu groß eingestellt werden (im Bereich 0.0001),

da nur mit 32-Bit Gleitkommazahlen gerechnet wird, um die GPU voll auslasten zu können. Ansonsten kann es zu einem Überlauf kommen, was an inf-Werten im Ergebnisvektor zu erkennen ist.

4.2 Testproblem

Als Haupttestproblem wurde das zweidimensionale Brusselator-Problem gewählt, welches die Reaktion zweier chemischer Substanzen unter Berücksichtigung von Diffusion beschreibt (Löseroption -ode:BRUSS2D-MIX). Dabei wird davon ausgegangen, dass die Eingabesubstanzen auf einer quadratischen Fläche vermischt liegen. Es wird dann ein Diskretisierungsgitter der Größe $N \times N$ über die Fläche gelegt und an jedem Gitterpunkt die Konzentration der Reaktionsprodukte im Verlauf der Zeit berechnet. Die Berechnung an Gitterpunkt (i, j) mit $i, j \in [1, N]$ erfolgt durch Lösen folgender ODEs [2]:

$$u'_{i,j} = B + u^2_{i,j} v_{i,j} - (A+1)u_{i,j} + \alpha(N-1)^2(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})$$

$$v'_{i,j} = Au_{i,j} - u^2_{i,j} v_{i,j} + \alpha(N-1)^2(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{i,j})$$

Mit Anfangswerten

$$u(i, j) = 0.5 + j, v(i, j) = 1 + 5i$$

und Konstanten

$$A = 3.4, B = 1, \alpha = 0.002$$

Dabei bezeichnen u und v die Konzentration der Reaktionsprodukte, A und B die Konzentration der Eingabesubstanzen, die zur Vereinfachung als konstant angenommen wird, und $\alpha = d/L^2$, wobei d der Diffusionskoeffizient und L die Reaktionslänge ist.

Die Komponenten werden in der Reihenfolge $u_{1,1}, v_{1,1}, u_{1,2}, v_{1,2}, ..., u_{N,N}, v_{N,N}$ im Speicher abgelegt, womit sich als Systemdimension $2N^2$ und als Zugriffsdistanz $2N$ ergibt.

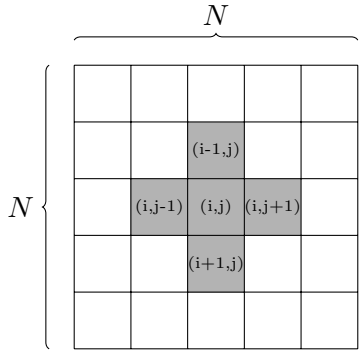

Abbildung 4.1: Zugriffsmuster des Brusselators

In Abbildung 4.2 ist ein beispielhafter Reaktionsverlauf des Brusselators zu sehen. Es wurde als Gittergröße $N = 10$ und als Schrittweite $h = 0.0001$ gewählt. Die Konzentration der Reaktionsprodukte wurde zu den Zeitpunkten $t = 0, 3, 6, 9$ berechnet. Zu erkennen ist die Oszillation der Konzentration u und v der Reaktionsprodukte, wenn die Gesamtreaktion weit vom Gleichgewichtszustand entfernt ist.

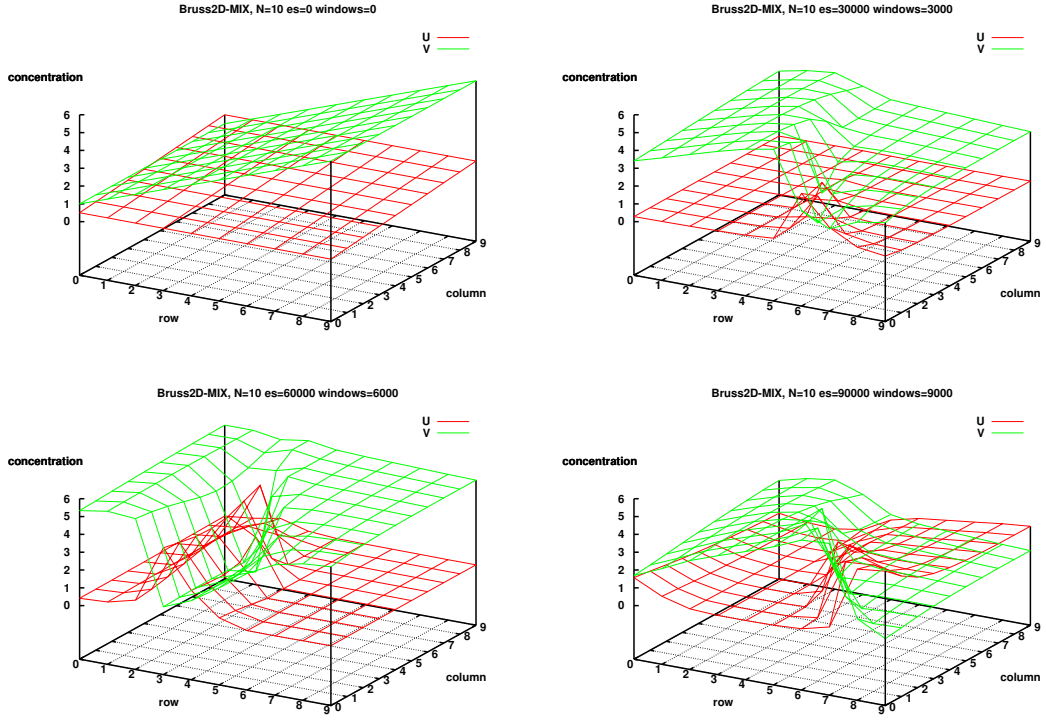

Abbildung 4.2: Reaktionsverlauf des Brusselators mit $N = 10$

4.3 Aufbau der Implementierungen

Alle Implementierungen bestehen aus einem Hostteil und einem Kernelteil. Der Hostteil wird von der CPU ausgeführt und initialisiert zunächst den CUDA-Treiber. Anschließend wird ein CUDA-Kontext für jede GPU erstellt und der Kernelteil als String geladen, welcher mithilfe der von NVIDIA zur Verfügung gestellten NVRTC-Library (siehe [4]) in PTX-Assembler kompiliert wird. Der PTX-Code wird anschließend vom CUDA-Treiber geladen und in ein von der GPU ausführbares Programm (Kernel) übersetzt. Der Kernel wird dann im Laufe des Host-Programms gestartet und berechnet je nach Löser einen Eulerschritt oder einen WR-Schritt auf der GPU.

4.4 Single-GPU Löser

Zuerst sind mehrere Single-GPU Varianten in diesem Löser (-solver:cuWR) implementiert worden. Die Implementierungen unterscheiden sich im Wesentlichen nur in den Kernels und den Startkonfigurationen der Kernels. Deshalb teilen sich die Implementierungen große Teile des Host-Codes.

Das Host-Programm (siehe Listing 4.1) teilt zunächst das Integrationsintervall in die vorgegebene Anzahl an Windows auf und bestimmt die Anzahl der Eulerschritte s pro Window. Danach alloziert es zwei $[s+1] \times d$ Matrizen Ycur und Ynew auf der GPU und initialisiert sie mit den Startwerten des Testproblems. Es stehen somit für jeden Eulerschritt zwei Vektoren der Länge d zur Verfügung.

$$\begin{bmatrix} y_1(t_0)^{(w-1)} & y_2(t_0)^{(w-1)} & \cdots & y_d(t_0)^{(w-1)} \\ y_1(t_1)^{(w-1)} & y_2(t_1)^{(w-1)} & \cdots & y_d(t_1)^{(w-1)} \\ \vdots & \vdots & \ddots & \vdots \\ y_1(t_s)^{(w-1)} & y_2(t_s)^{(w-1)} & \dots & y_d(t_s)^{(w-1)} \end{bmatrix} \begin{bmatrix} y_1(t_0)^{(w)} & y_2(t_0)^{(w)} & \cdots & y_d(t_0)^{(w)} \\ y_1(t_1)^{(w)} & y_2(t_1)^{(w)} & \cdots & y_d(t_1)^{(w)} \\ \vdots & \vdots & \ddots & \vdots \\ y_1(t_s)^{(w)} & y_2(t_s)^{(w)} & \dots & y_d(t_s)^{(w)} \end{bmatrix}$$

Abbildung 4.3: Speicherlayout von Ycur (links) und Ynew (rechts)

Anschließend iteriert der Host-Code in einer äußeren Schleife über die Windows. Eine weitere innere Schleife berechnet so lange die WR-Schritte des aktuellen Windows, bis das Konvergenzkriterium erfüllt ist. Für jede dieser WR-Iterationen startet der Host-Code den implementierungsspezifischen Kernel, der die Eulerschritte dieser Iteration berechnet. Dieser Kernel führt zusätzlich eine Berechnung des lokalen Fehlers $E = \max_{i \in [1,s]} ||Ynew[i] - Ycur[i]||_2$ aus. Ist der Fehler größer als der benutzerdefinierte Wert epsilon, so ist die Konvergenz nicht erfüllt. In diesem Fall beginnt der Host mit der nächsten WR-Iteration, die das Ergebnis der vorherigen WR-Iteration als Eingabe nimmt und dadurch verfeinert. Ist die Konvergenz für den WR-Schritt erfüllt, so verlässt das Hostprogramm die innerste Schleife und beginnt mit dem nächsten Window, welches das Ergebnis des vorherigen Windows als Startwert verwendet.

```
1  //get number of eulersteps per window
2  double H_window = H / windows;
3  int euler_steps = H_window / h;
4  //copy start values from host to device
5  copyHtoD(&d_Ycur[euler_steps], y0_window);
6  d_Yresult = d_Ycur;
7  for (int w = 0; w < windows; w++) {
8    //copy result of last window in first row of Ycur and Ynew
9    copyDtoD(d_Ycur, &d_Yresult[euler_steps]);
10   copyDtoD(d_Ynew, &d_Yresult[euler_steps]);
11   //execute wr-steps
12   int wr_steps = 0;
13   do{
```

```
14      cuMemset(d_convergence, 0);
15      launchKernel(wr_step_kernel, euler_steps, d_Ycur, d_Ynew,
            d_convergence);
16      swap(d_Ycur, d_Ynew);
17      copyDtoH(convergence, d_convergence);
18      wr_steps++;
19    }while(convergence >= epsilon);
20    //determine in which matrix the results where stored
21    d_Yresult = (wr_steps % 2) ? d_Ynew : d_Ycur;
22  }
23  //copy result from device to host
24  copyDtoH(y0_window, &d_Yresult[euler_steps]);
25  y = y0_window;
```

Listing 4.1: Hostseitiger Code der Single-GPU Versionen

4.4.1 Erste allgemeine Implementierung

Ziel dieser Implementierung (Löseroption -impl:general) ist es, eine im allgemeinen Fall, d.h. für alle Testprobleme, funktionierende Version zu erstellen, welche als Basis für Optimierungen dient. Bei allen Versionen besteht ein 1:1-Mapping zwischen den globalen Indices der Threads und den Systemkomponenten, d.h. ein Thread mit Index $j \in [1, d]$ berechnet auch die Systemkomponente $j \in [1, d]$ für alle Eulerschritte. Diese Implementierung verwendet neben den oben beschriebenen Matrizen Ycur und Ynew noch eine dritte Matrix Yevalcomp der Dimension $d \times d$, in welcher jeder Thread j sich einen Vektor $y_1(t_i)^{(w-1)} \dots y_j(t_i)^{(w)} \dots y_d(t_i)^{(w-1)}$ in jedem Eulerschritt i erstellt (siehe Listing 4.2).

$$\begin{bmatrix} y_1(t_i)^{(w)} & y_2(t_i)^{(w-1)} & \cdots & y_d(t_i)^{(w-1)} \\ y_1(t_i)^{(w-1)} & y_2(t_i)^{(w)} & \cdots & y_d(t_i)^{(w-1)} \\ \vdots & \vdots & \ddots & \vdots \\ y_1(t_i)^{(w-1)} & y_2(t_i)^{(w-1)} & \dots & y_d(t_i)^{(w)} \end{bmatrix}$$

Abbildung 4.4: Speicherlayout von Yevalcomp

Dieser Vektor wird der Auswertungsfunktion des Testproblems ode_eval_comp() übergeben, weil er alle Werte, die für die Berechnung des Komponentenwertes im nächsten Eulerschritt benötigt werden, enthält. Da das Erstellen des Vektors die meiste Zeit in Anspruch nimmt und zudem der Speicherbedarf von Yevalcomp quadratisch mit der Systemdimension wächst, ist diese Variante sehr ineffizient (vgl. Abschnitt 5.1.1).

4.4.2 Anpassung an das Testproblem

Bei dieser Version (Löseroption -impl:bruss) werden nicht mehr alle Werte aus Ycur in Yevalcomp kopiert, wie das in Zeile 5-6 von Listing 4.2 der Fall ist, sondern nur die

```
int j = thread_id;
for (int i = 0; i < euler_steps; i++) {
  float t = t0 + i * h;
  //make per thread vector for evaluation function
  for (int k = 0; k < ode_size; k++)
    Y_eval_comp[j][k] = Y_cur[i][k];
  Y_eval_comp[j][j] = newIJ;
  //compute value in next eulerstep
  Y_new[i+1][j] = Y_new[i][j] + h * ode_eval_comp(j, t, Y_eval_comp[j]);
  ...
}
```

Listing 4.2: Kernel der general Variante

Werte, die das Testproblem tatsächlich auch benötigt. Im Fall des Brusselators sind das die vier Nachbarn der aktuellen Komponente, wie in Abbildung 4.1 zu sehen ist. Damit funktioniert diese Version natürlich nicht mehr im allgemeinen Fall, jedoch verbessert sich die Laufzeit erheblich. Das Problem mit dem hohen Speicherbedarf besteht aber weiter.

4.4.3 Anpassung der Auswertungsfunktion

Eine andere Möglichkeit die Laufzeit zu verbessern, besteht darin, die Auswertungsfunktion des Testproblems anzupassen. Die neue Auswertungsfunktion ode_eval_comp_jacobi() bekommt den Wert einer Komponente j für Eulerschritt i aus dem aktuellen WR-Schritt explizit übergeben (siehe Listing 4.3). Um den Wert im Eulerschritt $i+1$ zu berechnen, wird für die Systemkomponente j der aktuelle Wert für Eulerschritt i (gespeichert in Ynew[i][j]) und für alle anderen Komponenten, auf die zugegriffen werden muss, der alte Wert vom letzten WR-Schritt (gespeichert in Ycur[i][j]) verwendet.

Damit wird Yevalcomp nicht mehr benötigt und der Speicherbedarf somit stark reduziert (vgl. Abschnitt 5.2.1). Zudem funktioniert diese Version (Löseroption -impl:nc) genauso wie die general-Version im allgemeinen Fall, da die Werte aller weiteren Komponenten aus dem vorangegangenen WR-Schritt (gespeichert in Ycur[i]) auch an die Auswertungsfunktion übergeben und bei Bedarf verwendet werden können.

4.4.4 Verdeckung der Speicherzugriffslatenz

Beim Zugriff auf die Werte, welche der Brusselator zur Berechnung einer Komponente benötigt (siehe Abbildung 4.1), muss auf den globalen Speicher zugegriffen werden. Dieser Zugriff ist teuer und erzeugt Wartezeiten durch Speicherzugriffslatenzen, falls die Werte direkt im Anschluss an die Ladeoperationen benötigt werden.

```
int j = thread_id;
for (int i = 0; i < euler_steps; i++) {
  float t = t0 + i * h;
  //compute value in next eulerstep
  float newValue = Y_new[i][j];
  Y_new[i+1][j] = newValue + h * ode_eval_comp_jacobi(j, t, newValue,
      Y_cur[i]);
  ...
}
```

Listing 4.3: Kernel der nc Variante

In dieser Version (Löseroption -impl:bruss_reg) werden der Auswertungsfunktion die Werte des Brusselator-Stencils direkt als einzelne Float-Parameter übergeben. Durch Betrachtung des Assembler-Codes zeigt sich, dass der Compiler dadurch in dieser Version im Vergleich zur nc-Version in der Lage ist Ladeoperationen vorzuziehen. Diese Optimierung bewirkt, dass Speicherzugriffslatenzen zum Teil durch Berechnungen überdeckt werden können. Daher ist diese Version etwas schneller als die nc-Version (vgl. Abschnitt 5.1.1).

4.4.5 Nutzung von Vektortypen

CUDA stellt für alle Basistypen Vektorversionen zur Verfügung (siehe [6] für eine komplette Liste). Ziel dieser Implementierung (Löseroption -impl:bruss2D_nc) ist es, durch Nutzung dieser Vektortypen Warpdivergenz zu reduzieren. Warpdivergenz bedeutet, dass nicht alle Threads eines Warps denselben Codepfad an einer Abzweigung nehmen (siehe Abschnitt 3.3.1).

Beim Brusselator-Problem bietet es sich an, die u und v Werte an einem Gitterpunkt in einen float2 zusammenzufassen und gleichzeitig zu berechnen. Dazu werden Ycur und Ynew als float2-Matrizen interpretiert und die Auswertungsfunktion entsprechend verändert. Damit erreicht diese Version eine leicht verbesserte Laufzeit und als positiver Nebeneffekt wird auch die Konvergenz verbessert, da nun sowohl für u als auch für v Werte aus dem aktuellen WR-Schritt für einen Thread zur Verfügung stehen.

4.4.6 Block-Jacobi mithilfe von Shared-Memory

Bei diesen Versionen wird eine Block-Jacobi-Strategie genutzt, um die Konvergenz zu verbessern. Für die Umsetzung des Blockings wird Shared-Memory genutzt, da ein Thread Werte von Komponenten aus dem aktuellen WR-Schritt verwenden kann, falls die Komponente von einem Thread aus demselben Threadblock berechnet wurde.

Es gibt zwei Implementierungen (siehe Listing 4.4) der Block-Jacobi-Strategie. Eine ist für den Brusselator optimiert (Löseroption -impl:bruss_shm), während die andere, ähn-

lich der nc-Variante, auch für andere Probleme funktioniert (Löseroption -impl:nc_shm). Da Shared-Memory sehr begrenzt ist, darf bei diesen Implementierungen allerdings die Zugriffsdistanz nicht zu groß sein.

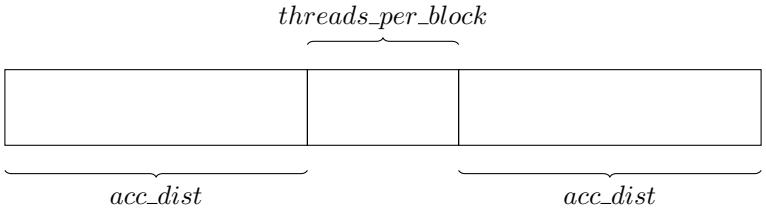

Abbildung 4.5: Shared-Memory-Layout

Im Folgenden wird nur die nc_shm-Version vorgestellt, da sich beide Block-Jacobi-Versionen nur geringfügig unterscheiden. Sei b der erste und e der letzte Thread eines Threadblocks und acc_dist die Zugriffsdistanz. Dann wird zu Beginn eines Eulerschritts i zunächst $y_b(t_i)^{(w)} \dots y_e(t_i)^{(w)}$ in den Shared-Memory geladen, indem jeder Thread seinen im vorherigen Eulerschritt $i-1$ berechneten Wert in den Shared-Memory schreibt. Für die Komponenten $y_{b-acc\_dist}(t_i)^{(w-1)} \dots y_{b-1}(t_i)^{(w-1)}$ und $y_{e+1}(t_i)^{(w-1)} \dots y_{e+acc\_dist}(t_i)^{(w-1)}$ werden Werte aus dem letzten WR-Schritt geladen, da diese von Threads aus anderen Threadblöcken berechnet werden und somit nicht sicher ist, ob die aktuellen Werte schon verfügbar sind (siehe Abbildung 4.5).

Anschließend werden die Threads innerhalb eines Threadblocks synchronisiert, um sicherzustellen, dass alle benötigten Werte in den Shared-Memory geladen wurden. Dann erfolgt die Berechnung, wobei der Auswertungsfunktion der Shared-Memory-Block übergeben wird. Abschließend müssen nochmal alle Threads innerhalb eines Threadblocks synchronisiert werden. Das vermeidet, dass Threads, die bereits im nächsten Eulerschritt sind, während der noch laufenden Berechnung in den Shared-Memory schreiben.

```
1  //shared memory of size 2*acc_dist + threads_per_block
2  extern __shared__ float blockYevalcomp[];
3  int j = thread_id;
4  int block_begin = block_id * threads_per_block;
5  int block_end = block_begin + threads_per_block - 1;
6  float* threadYevalcomp = blockYevalcomp - block_begin + acc_dist;
7  threadYevalcomp[j] = Y_new[0][j];
8  for (int i = 0; i < euler_steps; i++) {
9    float t = t0 + i * h;
10   //get old values at edges of the thread_block
11   for (int k=thread_id_block+1; k<=acc_dist; k+=threads_per_block ){
12     threadYevalcomp[block_begin-k] = Y_cur[i][block_begin-k];
13     threadYevalcomp[block_end+k] = Y_cur[i][block_end+k];
14   }
15   //wait for copy to finish
```

```
16    __syncthreads();
17    //compute value in next eulerstep
18    Y_new[i+1][j] = Y_new[i][j] + h * ode_eval_comp(j, t, threadYevalcomp);
19    //wait for computation to finish
20    __syncthreads();
21    //load new value in shared memory,
22    //so every thread of the block can see it
23    threadYevalcomp[j] = Y_new[i+1][j];
24    ...
25  }
```

Listing 4.4: Kernel der nc_shm Version

4.5 Multi-GPU Löser

Die Implementierungen dieses Lösers (-solver:cuWR_mgpu) können mehrere GPUs innerhalb eines Rechners zur Berechnung nutzen, indem die Systemkomponenten blockweise auf die einzelnen GPUs verteilt werden und jede GPU nur ihren Teil berechnet (siehe Abbildung 4.6 und Listing 4.5). Dazu muss im Host-Programm insbesondere Folgendes berücksichtigt werden:

- Jede GPU muss immer explizit angesteuert werden, indem sie als aktiv markiert wird
- Für jede GPU muss der Device-Code kompiliert werden, da es Unterschiede in der Architektur geben kann
- Für jede GPU müssen deren Allokationen verwaltet werden
- Der Datenaustausch und die Synchronisation zwischen den GPUs muss gesteuert werden

Das Berechnungsverfahren bleibt jedoch grundsätzlich dasselbe wie beim Single-GPU-Löser (siehe Listing 4.1).

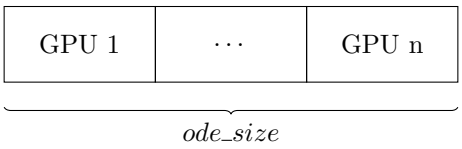

Abbildung 4.6: Aufteilung der Komponenten auf die GPUs im Rechner

Das Hauptproblem, das es zu lösen gilt, ist der Datenaustausch und die Synchronisation zwischen den GPUs, da im allgemeinen Fall jede GPU die Ergebnisse aller anderen GPUs benötigt. Da nach einem WR-Schritt eine globale Threadbarriere nötig ist, müssen die GPUs spätestens hier Daten austauschen. Für den Datenaustausch gibt es zwei

```
1 int start = gpu_id * (ode_size / device_count);
2 int end = (gpu_id == device_count - 1) ? ode_size : start + (ode_size /
      device_count);
3 thread_id += start;
4 if (thread_id >= end)
5   return;
6 ...
```

Listing 4.5: Komponentenaufteilung im Kernel

mögliche Strategien. Die eine ist, einen WR-Schritt vollständig auf jeder GPU zu berechnen (d.h. einen Kernel für einen WR-Schritt zu starten wie bisher) und anschließend alle notwendigen Daten auszutauschen. Das hat jedoch den Nachteil, dass keine zeitliche Überlappung von Berechnungen und Kopiervorgängen stattfinden kann.

Die andere Strategie, welche umgesetzt wurde, kopiert und rechnet parallel, indem für jeden Eulerschritt i ein Kernel gestartet wird und anschließend bereits begonnen wird das Ergebnis asynchron mithilfe von Streams auf die anderen GPUs zu kopieren. Dazu gibt es zwei Streams pro GPU, einen für die Berechnung der Eulerschritte (Compute-Stream) und einen für das Kopieren auf die anderen GPUs (Copy-Stream). Im gleichen Eulerschritt i einen WR-Schritt später, wenn die Daten der anderen GPUs benötigt werden, wird der Compute-Stream der aktuellen GPU synchronisiert. Da die Daten zu diesem Zeitpunkt in aller Regel bereits kopiert wurden, entsteht keine Wartezeit und es kann sofort weitergerechnet werden (siehe Abbildung 4.7).

Im Folgenden wird auf die Berechnung eines WR-Schritts im Multi-GPU Fall eingegangen (siehe Listing 4.6). Zunächst gibt es für jede GPU eine struct vom Typ gpu_state_t, die alle wichtigen Daten einer GPU, wie etwa Allokationen, Streams und Events, enthält. Diese struct wird bei der Initialisierung erstellt. Zu Beginn der Berechnung eines WR-Schritts wird in einer Schleife über alle GPUs iteriert und der entsprechende gpu_state_t geladen, sowie der Context für diese GPU als aktiv gesetzt. Anschließend wird in einer inneren Schleife über alle Eulerschritte iteriert. In dieser Schleife wird zunächst der Kernel zur Berechnung des Eulerschritts im Compute-Stream gestartet. Danach wird sich um den Datenaustausch mit den anderen GPUs und die damit einhergehende Synchronisation der Streams gekümmert.

Die Synchronisation erfolgt mithilfe von zwei CUDA-Events pro Eulerschritt pro GPU. Das eine Event tritt im Compute-Stream ein, wenn ein Eulerschritt i fertig berechnet wurde (Event: eulerstep_finished[i]) und das andere tritt im Copy-Stream ein, wenn die Ergebnisse des Eulerschritts i auf die anderen GPUs kopiert wurden (Event: copy_finished[i]). Bevor mit dem Kopieren begonnen werden kann, muss der Copy-Stream auf das Event eulerstep_finished[i] warten und bevor mit der Berechnung begonnen werden kann, muss der Compute-Stream der aktuellen GPU auf das Event copy_finished[i]

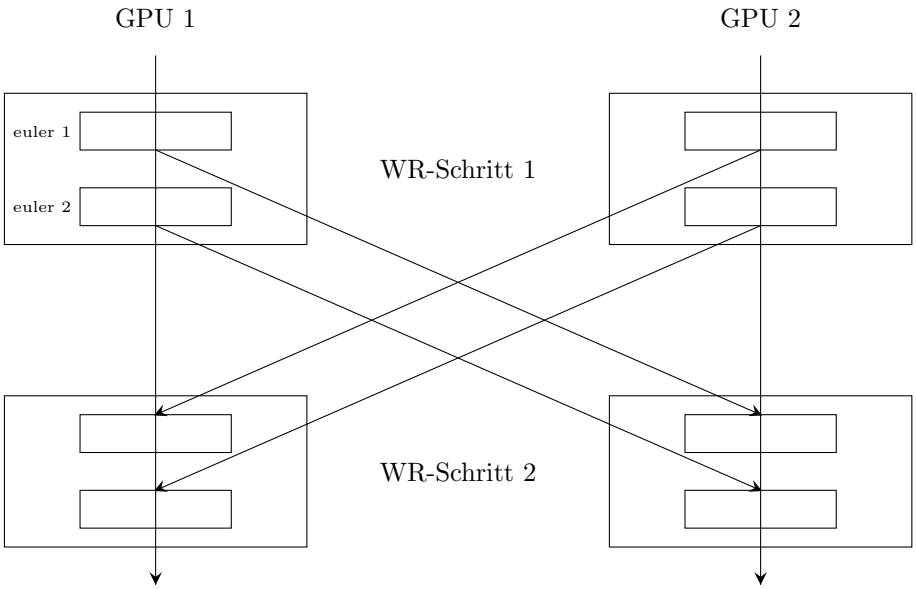

Abbildung 4.7: Datenaustausch zwischen den GPUs

der anderen GPUs aus dem letzten WR-Schritt warten (Cross-Device-Synchronisation). Auf die genaue Umsetzung wird später noch eingegangen, da diese zum Teil versionsspezifisch ist.

Nach der Berechnung eines WR-Schritts wird Ycur und Ynew für alle GPUs vertauscht, und anschließend wird das Konvergenzkriterium überprüft. Da jede GPU nur einen Teil des Ergebnisvektors berechnet, muss der lokale Fehler, den jede GPU für ihren Teil bestimmt, für die Überprüfung des Konvergenzkriteriums addiert werden.

```
1  ...
2  int wr_steps = 0;
3  do{
4    for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
5      gpu_state_t* gpu_state = &gpu_data->states[gpu_id];
6      //set gpu active
7      cuCtxSetCurrent(gpu_state->context);
8      cuMemset(gpu_state->d_convergence, 0);
9      for (int i = 0; i < euler_steps; i++) {
10       launchKernel(euler_step_kernel, i, gpu_state->d_Ycur,
             gpu_state->d_Ynew, gpu_state->d_convergence,
             gpu_state->compute_stream);
11       //event when kernel is finished
12       cuEventRecord(gpu_state->eulerstep_finished[i],
             gpu_state->compute_stream);
13       //wait for copies issued in last wr_step
```

```
14        if (wr_steps > 0)
15          wait_for_local_gpus(gpu_data, gpu_id, i);
16        //broadcast Ynew to other devices
17        send_to_local_gpus(gpu_data, gpu_id, i);
18        cuEventRecord(gpu_state->copy_finished[j],
              gpu_state->copy_stream));
19      }
20    }
21    //swap for all gpus
22    for (int gpu_id = 0; gpu_id < device_count; gpu_id++){
23      gpu_state_t* gpu_state = &gpu_data->states[gpu_id];
24      swap(gpu_state->d_Ycur, gpu_state->d_Ynew);
25    }
26    //compute convergence
27    float convergence = 0;
28    for (int gpu_id = 0; gpu_id < device_count; gpu_id++){
29      gpu_state_t* gpu_state = &gpu_data->states[gpu_id];
30      cuCtxSetCurrent(gpu_state->context);
31      copyDtoH(tmp, gpu_state->d_convergence);
32      convergence += tmp;
33    }
34    wr_steps++;
35 }while(convergence >= epsilon);
36 ...
```

Listing 4.6: Hostseitiger Code der Multi-GPU Versionen

4.5.1 Allgemeine Implementierung

Diese Version (-impl:general_mgpu) ist das Äquivalent zur in 4.4.1 beschriebenen Version für mehrere GPUs. Da diese Version im allgemeinen Fall funktionieren soll, muss jede GPU in jedem WR-Schritt ihre Ergebnisse an alle anderen GPUs broadcasten.

In der Funktion, die die Daten versendet (siehe Listing 4.7) muss dazu erst sichergestellt werden, dass die aktuelle GPU ihren Eulerschritt i fertig berechnet hat. Dazu wartet der Compute-Stream der aktuellen GPU bis das entsprechende Event eulerstep_finished[i] eingetreten ist. Anschließend wird in der Schleife über alle Empfänger iteriert und dabei das Ergebnis an diese Empfänger verschickt. Diese Kopieroperation läuft im Copy-Stream der aktuellen GPU, damit im Compute-Stream parallel dazu der nächste Eulerschritt $i + 1$ berechnet werden kann.

```
1 gpu_state_t* src_state = &gpu_data->states[gpu_id];
2 cuStreamWaitEvent(src_state->copy_stream,
      src_state->eulerstep_finished[i], 0);
3 int copy_offset = gpu_id * (ode_size / device_count);
4 int copy_size = ode_size / device_count;
5 for (int d = 0; d < device_count; d++) {
```

```
6     if (d != gpu_id) {
7       gpu_state_t* dest_state = &gpu_data->states[d];
8       //copy from active GPU with index gpu_id to GPU d
9       cuMemcpyPeerAsync(&dest_state->Ynew[i+1][copy_offset],
            dest_state->context, &src_state->Ynew[i+1][copy_offset],
10            src_state->context, copy_size, src_state->copy_stream));
11    }
12  }
```

Listing 4.7: send_to_local_gpus() im allgemeinen Fall

Die aktuelle GPU benötigt für die Berechnung eines Eulerschritts i selbst die Ergebnisse des Eulerschitts i der anderen GPUs aus dem letzten WR-Schritt. Daher muss, bevor der Kernel zur Berechnung eines Eulerschritts gestartet wird, sichergestellt werden, dass diese Daten bereits angekommen sind. Dazu wird in der wait_for_local_gpus-Funktion (siehe Listing 4.8) über alle anderen GPUs iteriert und der Compute-Stream der aktuellen GPU wartet auf das entsprechende Event copy_finished[i] der anderen GPU. Dabei findet ein Synchronisation über Devicegrenzen hinweg statt.

```
1 for (int d = 0; d < device_count; d++) {
2   if (d != gpu_id) {
3     //cross-device-synchronisation (GPU gpu_id waits for GPU d)
4     cuStreamWaitEvent(gpu_data->states[gpu_id].compute_stream,
          gpu_data->states[d].copy_finished[j], 0);
5   }
6 }
```

Listing 4.8: wait_for_local_gpus() im allgemeinen Fall

4.5.2 Load-Balancing

Ziel dieser Version (-impl:general_mgpu_dynLoad) ist es, die Arbeit auf die GPUs nach deren Rechenleistung zu verteilen. In der vorigen Version bekommt jede GPU einen gleichen Teil an Komponenten zur Berechnung (siehe Listing 4.5). Falls mehrere GPUs mit unterschiedlicher Rechenleistung verwendet werden, müssen die rechenstarken GPUs auf die rechenschwachen GPUs warten und werden so ausgebremst.

Hier werden die Komponenten zunächst auch gleich aufgeteilt, und anschließend werden während des ersten WR-Schritts die durchschnittlichen Zeiten für die Berechnung eines Eulerschritts für jede GPU gemessen. Anhand dieser Zeiten werden die Komponenten dann auf die GPUs neu verteilt, sodass GPUs mit kurzer gemessener Rechenzeit mehr Komponenten zugeteilt bekommen und GPUs mit langer gemessener Rechenzeit weniger Komponenten.

4.5.3 Begrenzte Zugriffsdistanz

Die bisher vorgestellten Multi-GPU-Versionen sind für den allgemeinen Fall gedacht. Im Folgenden soll auf die Änderungen bei den Versionen eingegangen werden, die eine begrenzte Zugriffsdistanz des Testproblems voraussetzen (z.B. -impl:nc_mgpu).

Sei b die erste Komponente und e die letzte Komponente des Komponentenblocks, den eine GPU berechnet. Bei der Berechnung von Problemen mit begrenzter Zugriffsdistanz benötigt diese GPU nur die Komponenten $[b-acc\_dist, b-1]$ und $[e+1, e+acc\_dist]$ von anderen GPUs. Unter der Voraussetzung, dass der Komponentenblock jeder GPU mindestens die Größe der Zugriffsdistanz hat, muss jede GPU nur Daten mit maximal zwei anderen GPUs austauschen, nämlich mit denjenigen, welche die beiden Nachbarblöcke berechnen.

Da ein 1:1-Mapping zwischen GPU-Index und Index des zu berechnenden Komponentenblocks besteht, sind für eine GPU g folgende Kommunikationsoperationen in jedem Eulerschritt nötig:

- Empfangen der Komponenten $[b - acc\_dist, b - 1]$ von GPU $g - 1$ und der Komponenten $[e + 1, e + acc\_dist]$ von GPU $g + 1$ aus dem letzten WR-Schritt
- Senden der Komponenten $[b, b+acc\_dist-1]$ an GPU $g-1$ und der Komponenten $[e - acc\_dist + 1, e]$ an GPU $g + 1$ aus dem aktuellen WR-Schritt

4.6 MPI-Multi-GPU Löser

Die Implementierungen dieses Lösers (-solver:dm_cuWR) können alle GPUs von mehreren Rechnern in einem Netzwerk zur Berechnung nutzen. Die Komponenten werden wie bisher blockweise auf alle GPUs aufgeteilt (siehe Abbildung 4.8). Für den Datenaustausch zwischen den Rechnern wird MPI verwendet. Der Datenaustausch zwischen den GPUs innerhalb eines Rechners erfolgt analog zu den Multi-GPU Versionen.

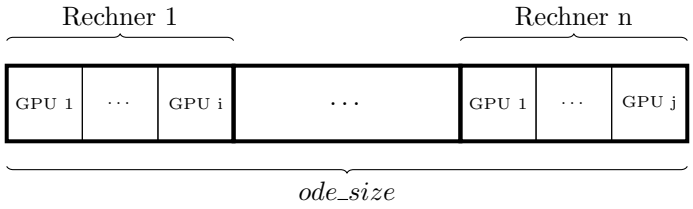

Abbildung 4.8: Aufteilung der Komponenten auf die GPUs im Netzwerk

Für den Datenaustausch mit MPI muss beachtet werden, dass dieser nur hostseitig funktioniert, d.h die Daten müssen im Hostspeicher liegen (siehe Abbildung 4.9). Daher werden nach jedem Eulerschritt die Teilergebnisse aller GPUs eines Rechners zunächst auf den Host in einen Buffer kopiert und anschließend von dort aus mit MPI versendet.

Genauso werden die Daten bei den Empfängern zunächst in einem Host-Buffer zwischengespeichert und dann von dort aus auf die lokalen GPUs verteilt. Die Kopieroperationen zwischen Host-Buffer und GPU laufen im MPI-Copy-Stream der jeweiligen GPU. Damit gibt es insgesamt drei Streams pro GPU, den Compute-Stream für die Berechnung eines Eulerschritts, den Copy-Stream für den direkten Datenaustausch zwischen den lokalen GPUs und den MPI-Copy-Stream für die Kopieroperationen zwischen dem von MPI verwendeten Host-Buffer und der GPU. Die für den Datenaustausch verwendeten MPI-Operationen sind zudem asynchron, da analog dem lokalen Datenaustausch der GPUs innerhalb eines Rechners (siehe Abbildung 4.7) der Empfänger die Daten erst im nächsten WR-Schritt benötigt.

Insgesamt sind innerhalb eines Eulerschritts i folgende Schritte pro Rechner im Netzwerk nötig:

- Empfangen der Ergebnisse des Eulerschritts i der anderen Rechner aus dem letzten WR-Schritt mit MPI
- Verteilen der empfangenen Ergebnisse an die lokalen GPUs
- Berechnen des Eulerschritts i im aktuellen WR-Schritt durch alle lokalen GPUs analog zu den Multi-GPU Versionen
- Kopieren der berechneten Ergebnisse auf den Host
- Versenden der Ergebnisse an die anderen Rechner im Netzwerk mit MPI

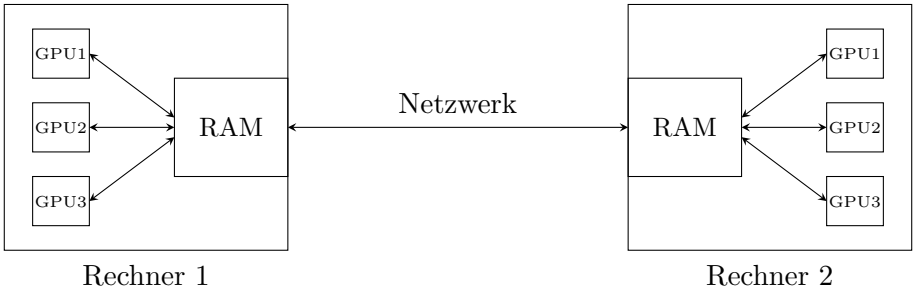

Abbildung 4.9: Datenaustausch zwischen den Rechnern

Nach der Berechnung eines WR-Schritts werden analog zu den Multi-GPU-Versionen Ycur und Ynew für jede GPU vertauscht und der lokale Fehler des Teilergebnisses, den die GPUs des aktuellen Rechners berechnet haben, bestimmt. Für die Überprüfung des Konvergenzkriteriums müssen jedoch noch die lokalen Fehler, die GPUs anderer Rechner berechnet haben, addiert werden. Das erfolgt mithilfe der MPI_Allreduce-Funktion (siehe Listing 4.9).

```
...
int wr_steps = 0;
do{
  for (int i = 0; i < euler_steps; i++) {
    //compute eulerstep i on each local GPU
    for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
      ...
    }
    //get sections from other remote nodes
    if (wr_steps > 0)
      wait_for_remote_nodes(node_data, common_node_data, i);
    //broadcast local Ynew to other remote nodes
    send_to_remote_nodes(node_data, common_node_data, i);
  }
  //swap Ycur and Ynew
  ...
  //compute local convergence
  float global_convergence, local_convergence = 0;
  for (int gpu_id = 0; gpu_id < device_count; gpu_id++){
    ...
  }
  //compute global convergence
  MPI_Allreduce(&local_convergence, &global_convergence, 1, MPI_FLOAT,
      MPI_SUM, MPI_COMM_WORLD);
  wr_steps++;
}while(global_convergence >= epsilon);
...
```

Listing 4.9: Hostseitiger Code der MPI Versionen

4.6.1 Allgemeine Implementierung

Diese Version (-impl:general_mpi_mgpu) ist das Äquivalent zur in Abschnitt 4.5.1 beschriebenen Version mit MPI-Unterstützung. Daher muss jede GPU in jedem WR-Schritt ihre Ergebnisse an alle anderen GPUs im Netzwerk verschicken.

In der Funktion, die die Daten an die anderen Rechner versendet (siehe Listing 4.10), wird zunächst über alle lokalen GPUs iteriert. Der MPI-Copy-Stream der aktuellen GPU wartet dann zunächst darauf, dass der Eulerschritt auf dieser GPU fertig berechnet wurde, indem er auf das Eintreten des entsprechenden eulerstep_finished-Event wartet. Anschließend werden die berechneten Ergebnisse auf den Host kopiert. Bevor mit dem Versenden an die anderen Rechner begonnen werden kann, müssen in einer weiteren Schleife die MPI-Copy-Streams aller lokalen GPUs synchronisiert werden. Damit ist sichergestellt, dass sich die Ergebnisse aller lokalen GPUs im Host-Speicher befinden. Danach wird in einer dritten Schleife über alle Rechner-IDs iteriert. Die Daten werden dabei mit einer asynchronen Broadcast-Operation an alle beteiligten Rechner versendet

und gleichzeitig werden schon die Empfangsoperationen gestartet auf deren Beendigung im nächsten WR-Schritt gewartet werden muss.

```
int copy_offset = gpu_id * (ode_size / device_count);
int copy_size = ode_size / device_count;
for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
  gpu_state_t* gpu_state = &node_data->gpu_states[gpu_id];
  cuCtxSetCurrent(gpu_state->context);
  //wait for euler_step result in Ynew
  cuStreamWaitEvent(gpu_state->mpi_copy_stream,
      gpu_state->eulerstep_finished[i], 0);
  //copy to host
  cuMemcpyDtoHAsync(&node_data->h_Y[i + 1][copy_offset],
  &gpu_state->d_Ynew[i + 1][copy_offset], copy_size,
      gpu_state->mpi_copy_stream);
}
//wait for copies to finish
for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
  gpu_state_t* gpu_state = &node_data->gpu_states[gpu_id];
  cuStreamSynchronize(gpu_state->mpi_copy_stream);
}
//start the remote sending and recieving procedure
for (int n = 0; n < node_count; n++) {
  MPI_Ibcast(&node_data->h_Y[i + 1][copy_offset], copy_size, MPI_FLOAT,
      n, MPI_COMM_WORLD, &mpi_requests[i][n]);
}
```

Listing 4.10: send_to_remote_nodes() im allgemeinen Fall

In der wait_for_remote_nodes-Funktion (siehe Listing 4.11) wird über alle Rechner-IDs iteriert. Zuerst wird dann auf die Beendigung der asynchronen MPI-Operation gewartet, die der Rechner mit der jeweiligen ID im letzten WR-Schritt gestartet hat. Anschließend werden die Daten von jeder beendeten Empfangsoperation auf alle lokalen GPUs kopiert. Das Kopieren auf die lokalen GPUs erfolgt im MPI-Copy-Stream der jeweiligen GPU und daher muss dieser am Ende der Funktion für alle GPUs synchronisiert werden. Damit ist dann sichergestellt, dass alle lokalen GPUs die Daten der anderen Rechner aus dem letzten WR-Schritt im Speicher haben.

```
int copy_offset = gpu_id * (ode_size / device_count);
int copy_size = ode_size / device_count;
for (int n = 0; n < node_count; n++) {
  //wait for nonblocking MPI-calls from last WR-step to finish
  MPI_Wait(&mpi_requests[i][n], MPI_STATUS_IGNORE);
  // copy recieved results to local GPUs
  if (n != node_rank) {
    for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
      gpu_state_t* gpu_state = &node_data->gpu_states[gpu_id];
```

```
10        cuCtxSetCurrent(gpu_state->context);
11        //copy to device (note d_Ycur because of swap)
12        cuMemcpyHtoDAsync(&gpu_state->d_Ynew[i + 1][copy_offset],
13        &node_data->h_Y[i + 1][copy_offset], copy_size,
              gpu_state->mpi_copy_stream);
14      }
15    }
16  }
17  //wait for copies to finish
18  for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
19    gpu_state_t* gpu_state = &node_data->gpu_states[gpu_id];
20    cuStreamSynchronize(gpu_state->mpi_copy_stream);
21  }
```

Listing 4.11: wait_for_remote_nodes() im allgemeinen Fall

4.6.2 Begrenzte Zugriffsdistanz

Bei Problemen mit begrenzter Zugriffsdistanz benötigt eine GPU, unter Annahme, dass der Komponentenblock jeder GPU mindestens die Zugriffsdistanz umfasst, nur die Ergebnisse der beiden GPUs, welche die Nachbarblöcke berechnen (siehe Abschnitt 4.5.3). Falls beide Nachbarblöcke von GPUs desselben Rechners berechnet werden, müssen nur lokal Daten ausgetauscht werden und es ist keine Kommunikation zwischen den Rechnern nötig. Im anderen Fall, wenn einer der Nachbarblöcke auf einer GPU eines anderen Rechners berechnet wird, müssen Daten über das Rechnernetz ausgetauscht werden. Anders als im Abschnitt 4.6.1 beschriebenen allgemeinen Fall ist aber keine Broadcast-Operation nötig, sondern es kann mit einfachen Send- und Receive-Operationen gearbeitet werden, da der Datenaustausch mit maximal zwei GPUs erfolgt.

4.6.3 Load-Balancing bei begrenzter Zugriffsdistanz

Die Berechnung der Lastbalancierung erfolgt in dieser Implementierung (-impl:nc_mpi_mgpu_dynLoad) grundsätzlich wie in Abschnitt 4.5.2 beschrieben. Allerdings kann die Lastbalancierung nicht mehr während der Berechnung des ersten WR-Schritts erfolgen.

Bei begrenzter Zugriffsdistanz wird für jede GPU Ycur und Ynew nur mit Dimension $s \times number\_components$ allokiert, wobei $number\_components$ der Größe des von der jeweiligen GPU berechneten Komponentenblocks plus zwei mal der Zugriffsdistanz entspricht. Deshalb müssen bei Änderung der Größe des von einer GPU zu berechnenden Komponentenblocks Ycur und Ynew reallokiert werden. Dann würden aber die bereits gestarteten lokalen Sendeoperationen (siehe Abschnitt 4.5.1) einen Programmabsturz verursachen, da Ynew bei diesen Sendeoperationen als Sende- bzw. Empfangsbuffer verwendet wird. Daher kann die Größe des Komponentenblocks einer GPU nicht während der Berechnung verändert werden und somit kann auch die Lastbalancierung nicht während der Berechnung durchgeführt werden.

Die Lastbalancierung erfolgt hier noch vor Berechnung des ersten Windows. Dazu wird der Kernel zur Berechnung eines Eulerschritts einige Male auf jeder GPU gestartet und die durchschnittlichen Zeiten gemessen, welche für die Verteilung der Komponenten auf die GPUs benötigt werden (siehe Abschnitt 4.5.2). Die Eulerschritte, die hier ausgeführt werden, dienen nur zur Zeitmessung und sind nicht Teil der eigentlichen Berechnung.

4.6.4 Seperate Kernel für Mitte und Ränder einer Sektion

In den bisher vorgestellten MPI- und Multi-GPU-Versionen wurde für jeden Eulerschritt ein Kernel gestartet, um eine Überlappung von Datenübertragung und Berechnung zu erreichen. In dieser Implementierung (-impl:edge_mpi_mgpu) berechnet ein Kernel, analog zu den Single-GPU-Versionen, einen WR-Schritt. Anders als in den Single-GPU-Versionen werden jedoch die Gleichungen, die eine GPU zugeteilt bekommt, in einen linken und rechten Randbereich, sowie einen Mittelbereich aufgeteilt (siehe Abbildung 4.10).

Diese Bereiche werden dann jeweils mit einem seperaten Kernel in verschiedenen Streams berechnet. Die Anzahl der Gleichungen des linken und rechten Randbereichs entspricht dabei der Zugriffsdistanz des Problems. Dadurch ist die Berechnung des Mittelteils vollkommen unabhängig von Systemkomponenten, die auf anderen GPUs berechnet werden. Während für die Berechnung der Ränder auf Daten von anderen GPUs gewartet wird, kann parallel dazu mit der Berechnung des Mittelteils bereits begonnen werden.

Abbildung 4.10: Aufteilung der Komponenten einer GPU

4.6.5 Polling des Berechnungsstatus eines Randkernels

Das Problem in der im Abschnitt 4.6.4 beschriebenen Version ist, dass die Ergebnisse der Randkernel erst versendet werden können, wenn der komplette WR-Schritt berechnet wurde. In dieser Version (-impl:edge_mpi_mgpu_sendAsync) wird dieses Problem behoben, indem für jede GPU innerhalb eines Rechners ein Host-Thread mithilfe von OpenMP gestartet wird.

Jeder dieser Host-Threads frägt in einer while-Schleife die Anzahl der bereits berechneten Eulerschritte auf der ihm zugeteilten GPU regelmäßig ab und versendet anschließend die Ergebnisse der bereits berechneten Eulerschritte an die anderen GPUs (siehe Listing

4.12). Dabei ist wichtig, dass dies nur für die Randkernel erfolgt, da die Ergebnisse des mittleren Bereiches auf den anderen GPUs nicht benötigt werden (vgl. Abschnitt 4.6.4).

```
1  //start a host-thread for each GPU of the local computer
2  omp_set_num_threads(device_count);
3  #pragma omp parallel for
4  for (int gpu_id = 0; gpu_id < device_count; gpu_id++) {
5    gpu_state_t* gpu_state = &gpu_data->states[gpu_id];
6    cuCtxSetCurrent(gpu_state->context);
7    ...
8    //launch wr-step-kernel for left edge, right edge and middle in
         different streams
9    launchKernel(gpu_state->wr_step_edge_left,...,
10       gpu_state->d_euler_count_left,gpu_state->stream_edge_left);
11   launchKernel(gpu_state->wr_step_edge_right,...,
12       gpu_state->d_euler_count_right,gpu_state->stream_edge_right);
13   launchKernel(gpu_state->wr_step_middle,...,gpu_state->stream_middle);
14   int old_count_left = 0, old_count_right = 0;
15   int current_count_left = 0, current_count_right = 0;
16   while (true) {
17     //poll number of finished euler_steps from running edge kernels
18     cuMemcpyDtoH(&current_count_left, gpu_state->d_euler_count_left,
           sizeof(int));
19     cuMemcpyDtoH(&current_count_right, gpu_state->d_euler_count_right,
           sizeof(int));
20     //send results of finished euler_steps to local and remote GPUs
21     for (int i = old_count_left; i < current_count_left; i++) {
22         send_to_local_gpus_left(..., gpu_id, i);
23         send_to_remote_nodes_left(..., gpu_id, i);
24     }
25     for (int i = old_count_right; i < current_count_right; i++) {
26         send_to_local_gpus_right(..., gpu_id, i);
27         send_to_remote_nodes_right(..., gpu_id, i);
28     }
29     //terminate loop if all euler_steps are completed in the edge kernels
30     old_count_left = current_count_left;
31     old_count_right = current_count_right;
32     if (old_count_left == euler_steps && old_count_right == euler_steps)
33       break;
34   }
35 }
```

Listing 4.12: Überlappung der WR-Kernel-Berechnung mit der Ergebnis-Versendung

5 Analyse

Die im vorherigen Kapitel vorgestellten Implementierungen werden in diesem Kapitel auf Rechenzeit, Speicherverbrauch und Konvergenzverhalten hin untersucht. Zudem wird auch kurz auf die Auslastung der zur Berechnung genutzten GPUs eingegangen und untersucht, ob sich das WR-Verfahren im Vergleich zum Eulerverfahren lohnt. Als Testproblem wird das Brusselator-Problem (siehe Abschnitt 4.2) verwendet.

5.1 Rechenzeit

Im Folgenden wird die Laufzeit der verschiedenen Implementierungen miteinander verglichen. Dazu wird der Brusselator mit aufsteigender Gittergröße N gelöst und jeweils die Laufzeit der verschiedenen Implementierungen gemessen.

5.1.1 Single-GPU Versionen

Zunächst werden nur die Single-GPU Implementierungen betrachtet. Dazu wurde als Schrittweite $h = 0.0001$ und als Integrationsintervall $H = 0.01$ gewählt. Es müssen also 100 Eulerschritte ausgeführt werden. Außerdem ist die Anzahl der WR-Schritte fest mit 8 eingestellt, d.h. die Konvergenz wird hier noch nicht berücksichtigt. Für alle Tests in diesem Abschnitt wurde eine GeForce GTX 980 Ti verwendet.

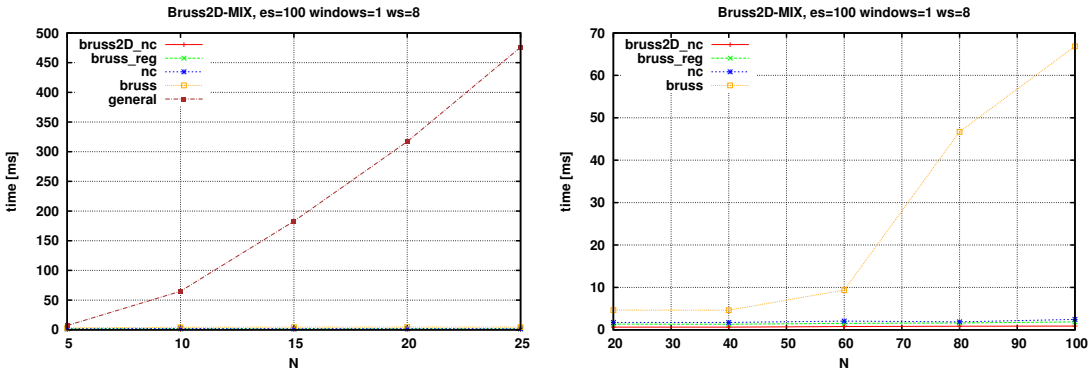

Abbildung 5.1: Laufzeittests mit und ohne general-Version

In Abbildung 5.1 links sind die Berechnungszeiten des Brusselators mit Gittergrößen $N = 5, 10, \ldots, 25$ in Millisekunden zu sehen. Die Zeiten wurden mit den Implementierungen general, bruss, nc, bruss_reg und bruss2D_nc (siehe Abschnitte 4.4.1, 4.4.2, 4.4.3, 4.4.4, 4.4.5) gemessen. Man sieht, dass die general-Version sehr viel mehr Zeit als die anderen Versionen benötigt. Das liegt daran, dass in der general-Version der Vektor $y_1(t_i)^{(w-1)} \ldots y_j(t_i)^{(w)} \ldots y_d(t_i)^{(w-1)}$ für die Auswertungsfunktion in jedem Eulerschritt

i aus den Werten in Ycur und Ynew neu erstellt wird und das mit Abstand die meiste Zeit in Anspruch nimmt.

Ähnlich verhält es sich mit der bruss-Version, wie in Abbildung 5.1 rechts zu sehen ist, wenngleich hier der Vektor für die Auswertungsfunktion nur die fünf Werte des Brusselator-Stencils benötigt. Dort ist die general-Version nicht Teil der Messung, und es wurden die Gittergrößen $N = 20, 40, \ldots, 100$ verwendet.

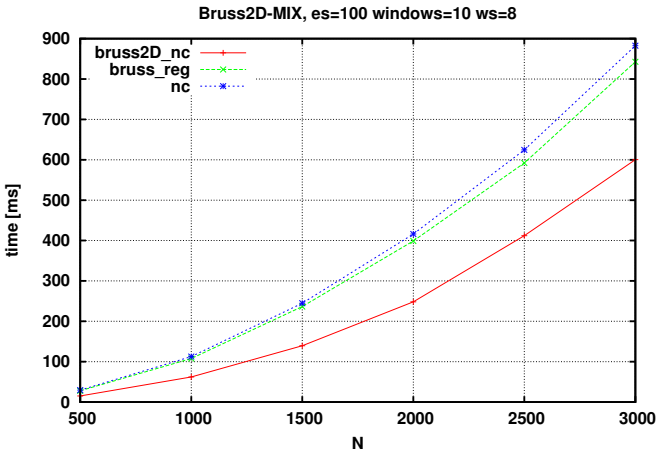

Abbildung 5.2: Laufzeittests ohne general- und bruss-Version

In Abbildung 5.2 sind nur die Laufzeiten der Implementierungen nc, bruss_reg und bruss2D_nc zu sehen. Für die Messung wurden die Gittergrößen $N = 500, 1.000, \ldots, 3.000$ verwendet und das Integrationsintervall in 10 Windows unterteilt. Man erkennt, dass die bruss_reg-Version etwas schneller als die nc-Version ist und dass die bruss2D_nc-Version am schnellsten ist.

Diese drei Versionen werden nun etwas genauer betrachtet. Zunächst wurden mit dem CUDA-Profiler (siehe [5]) die IPC (instructions per cycle), die genutzte Speicherbandbreite für Lesezugriffe und die genutzte Speicherbandbreite für Schreibzugriffe gemessen. Die IPC sind in Abbildung 5.3 oben zu sehen und die genutzte Speicherbandbreite für Lese- bzw. Schreibzugriffe links bzw. rechts unten. Für die IPC gilt bei der für die Messung verwendeten GeForce GTX 980 Ti ein Limit von vier, da ein Block von 32 CUDA-Cores mit einer Instruktion pro Taktzyklus voll ausgelastet wird und ein SM vier solche Blöcke besitzt (siehe Abschnitt 3.2). Für die insgesamt genutzte Speicherbandbreite gibt es ein Limit von 336 GB/s.

Man sieht, dass die IPC von der nc-Version und der bruss_reg-Version im Bereich von 3.8 sind und die insgesamt ausgenutzte Speicherbandbreite im Bereich von 130 GB/s. Damit ist bei diesen Versionen die IPC der limitierende Faktor, d.h. man sollte zunächst versuchen die Anzahl der Instruktionen zu verringern um die Performance weiter zu

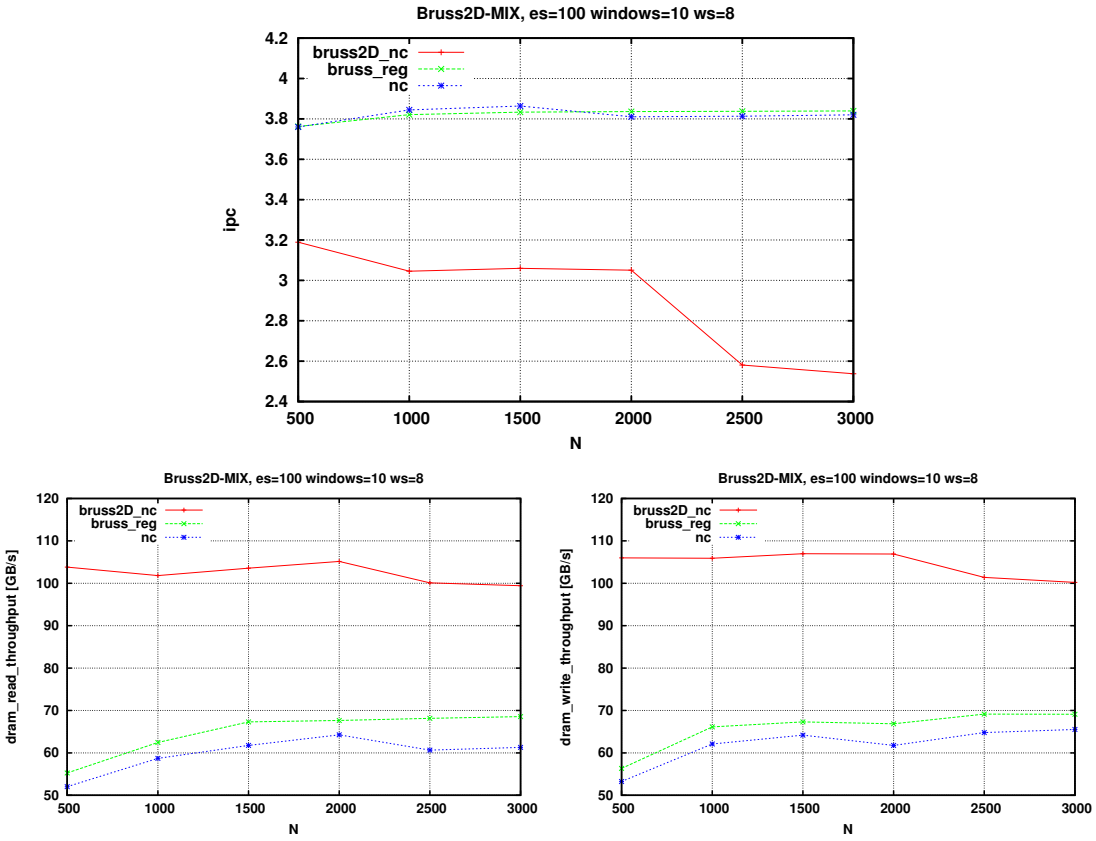

Abbildung 5.3: IPC und genutzte Speicherbandbreite von nc, bruss_reg, bruss2D_nc

verbessern. Bei der bruss2D_nc-Version ist es genau anders herum. Hier nehmen die IPC mit wachsendem N ab und sind bei $N = 3000$ nur noch bei ca. 2.55. Die insgesamt genutzte Speicherbandbreite ist mit ca. 210 GB/s dagegen recht hoch.

5.1.2 Multi-GPU Versionen

In diesem Abschnitt werden die Multi-GPU- und MPI-Implementierungen untersucht. Für die Berechnung standen folgende vier Rechner mit entsprechenden GPUs zur Verfügung:

- node22 mit 4x GeForce GTX 980 Ti
- node23 mit 2x GeForce GTX 670
- 2gpus mit 2x GeForce GTX TITAN Black
- 4gpus mit 2x GeForce GTX 980 Ti und 1x GeForce GTX TITAN X

Zunächst erfolgt eine Gegenüberstellung von Single-GPU und Multi-GPU bei Ausnutzung der begrenzten Zugriffsdistanz. Dazu wird in Abbildung 5.4 die nc-Version mit der nc_mgpu-Version auf node22 verglichen. Man kann erkennen, dass bei geringer Gittergröße die nc-Version noch schneller ist, aber mit zunehmender Gittergröße im Vergleich zur nc_mgpu-Version immer langsamer wird. Der Grund für dieses Verhalten ist der Overhead für den Datenaustausch zwischen den GPUs, der bei kleiner Gittergröße stärker ins Gewicht fällt.

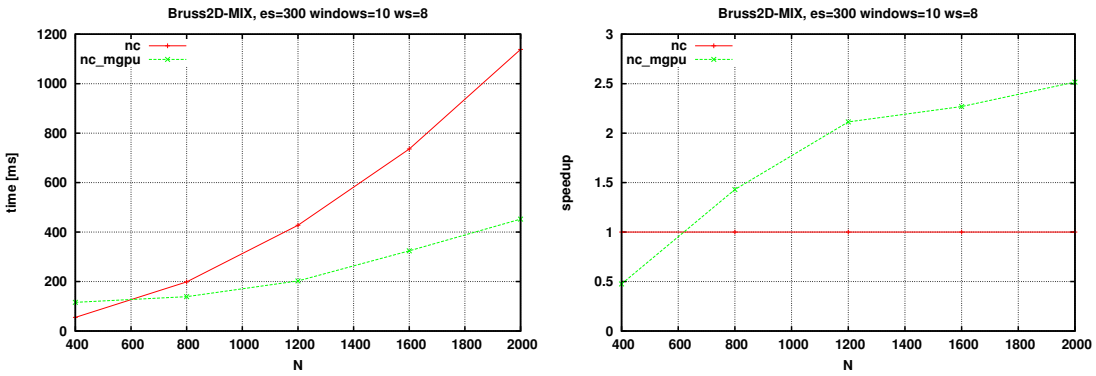

Abbildung 5.4: Vergleich von nc und nc_mgpu

Im Folgenden wird eine auf begrenzte Zugriffsdistanz optimierte MPI-Multi-GPU-Implementierung mit Verwendung von Load-Balancing (-impl:nc_mpi_mgpu_dynLoad) betrachtet. In Abbildung 5.5 ist das Laufzeitverhalten des Brusselators bei jeweils unterschiedlicher Anzahl an beteiligten Rechnern zu sehen. Dabei ist links die Laufzeit des Brusselators für Gittergrößen $N = 1.000, 2.000, \ldots, 5.000$ mit je 500 Eulerschritten und 100 Windows auf 4gpus allein und auf 4gpus zusammen mit node22 zu sehen. Rechts wurden die Gittergrößen $N = 2.000, 4.000, \ldots, 10.000$ mit je 30 Eulerschritten aufgeteilt auf sechs Windows verwendet und auf den Rechnermengen $\{4gpus\}, \{4gpus, 2gpus\}$, $\{4gpus, 2gpus, node22\}, \{4gpus, 2gpus, node22, node23\}$ getestet.

Man kann erkennen, dass erst bei sehr großen Gittergrößen ein Performancegewinn bei Nutzung mehrerer Rechner zu verzeichnen ist, da beim Datenaustausch über das Netzwerk im Vergleich zum Datenaustausch über den PCI-Express die Übertragungsrate gering und insbesondere die Latenz groß ist.

Um die Auswirkung des Load-Balancing zu zeigen, wurde der Brusselator zuerst auf 4gpus und anschließend auf 4gpus zusammen mit node23 ausgeführt. Da node23 mit den beiden GeForce GTX 670 im Vergleich zu 4gpus nur zwei rechenschwache GPUs besitzt, ist in dieser Konfiguration das Load-Balancing besonders wichtig, was in Abbildung 5.6 gut zu erkennen ist. Ohne Load-Balancing ist die Laufzeit bei Nutzung beider Rechner durchgehend länger, als wenn 4gpus alleine verwendet wird. Mit Load-Balancing hingegen ist bei Verwendung beider Rechner trotz des großen Unterschiedes

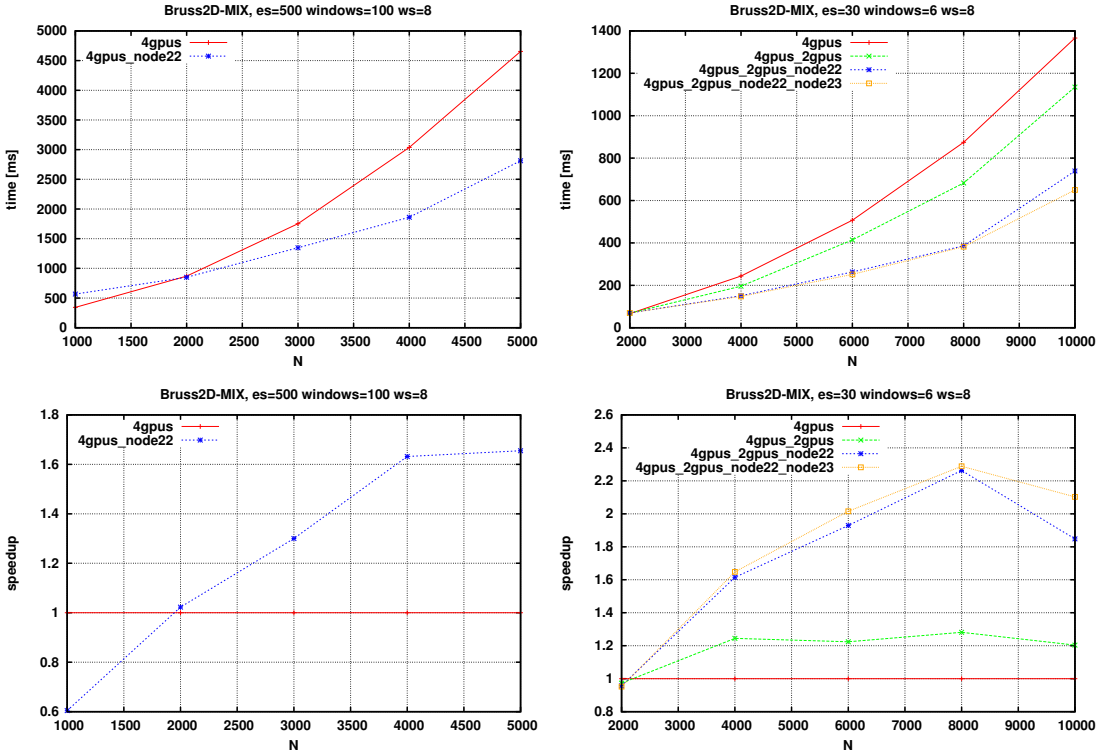

Abbildung 5.5: Laufzeit von nc_mpi_mgpu_dynLoad bei unterschiedlichen beteiligten Rechnern

in der Rechenleistung eine Verbesserung der Laufzeit bei ausreichender Gittergröße zu beobachten.

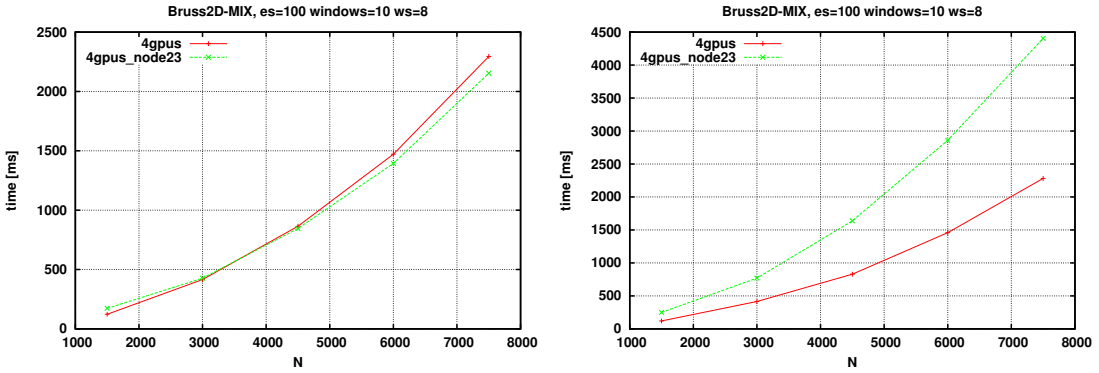

Abbildung 5.6: Laufzeit von nc_mpi_mgpu_dynLoad mit (links) und ohne (rechts) aktiviertem Load-Balancing

Bei optimalem Load-Balancing bekommt jede GPU genau so viele Komponenten, dass die Berechnung eines Eulerschritts auf allen GPUs genau gleich lange dauert. Durch Profiling des in Abbildung 5.6 gezeigten Testfalls mit aktiviertem Load-Balancing zeigt sich, dass die GeForce GTX 670 durchschnittlich 15 Prozent mehr Zeit für einen Eulerschritt benötigt als die GeForce GTX 980 Ti, d.h., dass die GeForce GTX 670 immer noch etwas zu viele Komponenten zugeteilt bekommt.

In Abbildung 5.7 ist ein Vergleich einer MPI-Version, die für jeden Eulerschritt einen Kernel startet (-impl:nc_mpi_mgpu) und einer MPI-Version, die alle Eulerschritte eines WR-Schritts in einem Kernel ausführt (-impl:edge_mpi_mgpu_sendAsync), zu sehen. Man kann erkennen, dass bei Nutzung mehrerer Rechner die edge_mpi_mgpu_sendAsync-Version schneller ist, da der Overhead für das Starten der einzelnen Euler-Kernel wegfällt und durch die Aufteilung des Komponentenblocks einer GPU in Mitte und Rand mehr Parallelität ermöglicht wird. Bei dieser Messung ist zu beachten, dass hier die GPUs von 4gpus und node23 im Vergleich zu den anderen Messungen getauscht wurden.

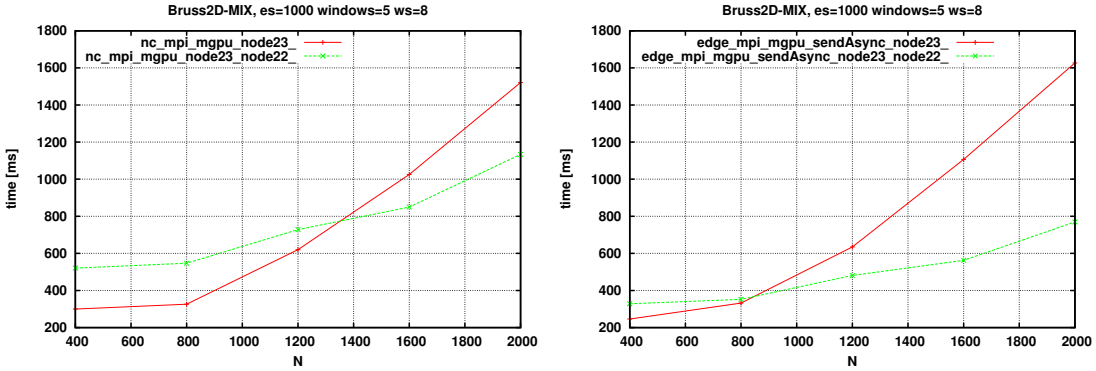

Abbildung 5.7: Laufzeit von nc_mpi_mgpu (links) und edge_mpi_mgpu_sendAsync (rechts)

5.2 Speicherbedarf

5.2.1 Single-GPU Versionen

Es wird nun der Speicherbedarf der Implementierungen betrachtet. Am meisten Speicher benötigt die general-Version, da hier neben den beiden $[s+1] \times d$ Matrizen Ycur und Ynew, welche alle Versionen gemeinsam haben, noch die Matrix Yevalcomp mit Dimension $d \times d$ benötigt wird. Diese speichert für jeden Thread den Vektor für die Auswertungsfunktion (siehe Abbildungen 4.3 und 4.4). In der bruss-Version benötigt jeder Thread j in Yevalcomp nur noch Platz für $4N + 1$ Elemente, da beim Brusselator die Zugriffsdistanz $2N$ ist und der Thread j damit nur Zugriff auf die Komponenten $[j - 2N, j + 2N]$ benötigt. Alle anderen Versionen kommen ohne ein Yevalcomp aus und brauchen nur Ycur und Ynew. In Abbildung 5.8 ist der benötigte GPU-Speicher für

Ycur, Ynew und Yevalcomp bei den verschiedenen Implementierungen in Megabyte zu sehen. Die Werte sind dabei für 10 Eulerschritte und Gittergrößen $N = 10, 100, 1000$.

Impl\N	10	100	1000
general	0,1776 MB	1601,76 MB	16.000.176 MB
bruss	0,0504 MB	33,84 MB	32.184 MB
nc	0,0176 MB	1,76 MB	176 MB
bruss_reg	0,0176 MB	1,76 MB	176 MB
bruss2D_nc	0,0176 MB	1,76 MB	176 MB

Abbildung 5.8: Benötigter GPU-Speicher bei 10 Eulerschritten pro Window

5.2.2 Multi-GPU Versionen

Bei den Implementierungen die mehrere GPUs nutzen kann, aufgrund der blockweisen Aufteilung der Komponenten auf die GPUs, in der Regel auch der Speicherbedarf pro GPU reduziert werden. So muss Yevalcomp nur für die Komponenten, welche auf der zugehörigen GPU berechnet werden eine Zeile enthalten. Bei Ycur und Ynew kann der Speicherbedarf im allgemeinen Fall nicht reduziert werden, da in der Auswertungsfunktion der Zugriff auf alle Systemkomponenten möglich sein muss. Falls von begrenzter Zugriffsdistanz ausgegangen wird, muss in Ycur und Ynew in jeder Zeile nur Platz für die Systemkomponenten $j \in [b - acc\_dist, e + acc\_dist]$ zur Verfügung stehen. Dabei ist b der Beginn des Komponentenblocks der jeweiligen GPU und e dessen Ende.

5.3 Konvergenzverhalten

Im Folgenden wird kurz auf das Konvergenzverhalten der Implementierungen eingegangen. Grundsätzlich ist es so, dass, je größer das Integrationsintervall ist und je größer die Systemdimension ist, desto mehr WR-Schritte sind nötig, um das Konvergenzkriterium zu erfüllen. Unterschiedliches Konvergenzverhalten bei gleicher Anzahl an Eulerschritten und gleicher Systemdimension gibt es nur zwischen den Implementierungen, welche die Block-Jacobi-WR-Methode nutzen (bruss_shm, nc_shm) und denen, die die normale Jacobi-WR-Methode nutzen (alle anderen Implementierungen).

Die folgenden Messungen wurden mit einer GeForce GTX 980 Ti durchgeführt. In Abbildung 5.9 ist ein Vergleich des Konvergenzverhaltens zwischen der nc-Version und der bruss_shm-Version zu sehen. Wie dort zu erkennen ist, konvergiert die bruss_shm-Version schneller, d.h. es sind weniger WR-Schritte zur Erfüllung des Konvergenzkriteriums notwendig. Die geringere Anzahl an WR-Schritten wirkt sich wiederum positiv auf die Laufzeit von bruss_shm aus, wie in Abbildung 5.10 zu sehen ist. Dort wurden die gleichen Parameter wie in Abbildung 5.9 verwendet und die Laufzeit bei verschiedenen

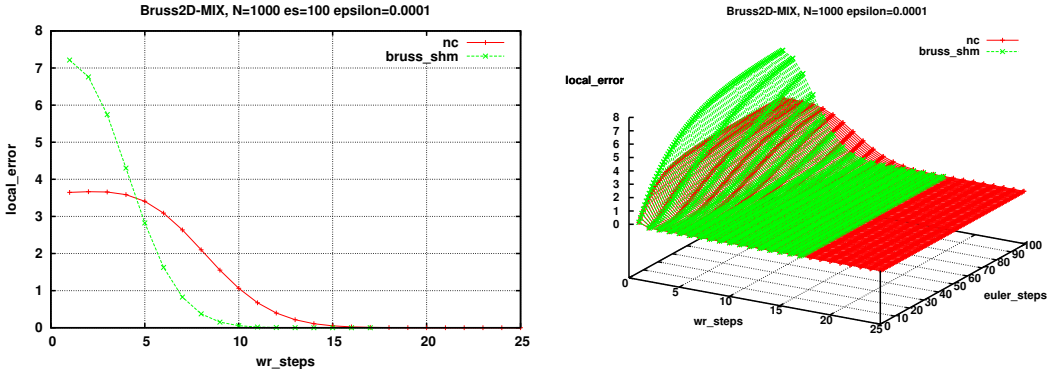

Abbildung 5.9: Konvergenzverhalten von nc und bruss_shm

Gittergrößen gemessen. Im Fall $N = 1000$ hat bruss_shm 17 WR-Schritte und nc 25 WR-Schritte ausgeführt (vgl. Abbildung 5.9).

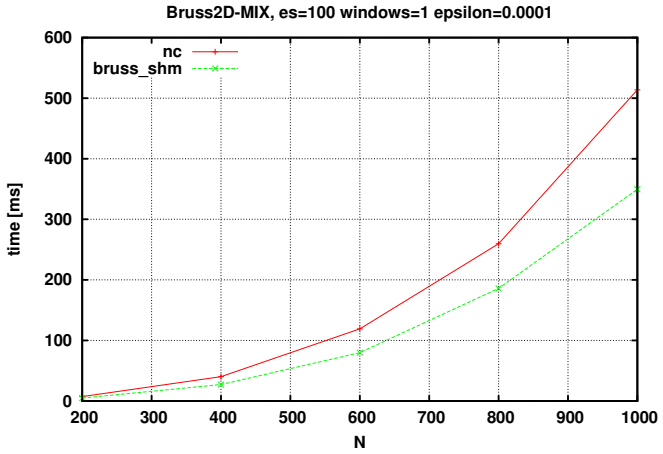

Abbildung 5.10: Laufzeitverhalten von nc und bruss_shm

Auch die Anzahl der Windows beeinflusst das Konvergenzverhalten. Bei einer großen Anzahl an Windows müssen innerhalb eines Windows weniger Eulerschritte ausgeführt werden, was sich positiv auf die Konvergenz auswirkt. Da die Berechnung der Windows jedoch sequentiell ist und der WR-Algorithmus auf jedem Window angewandt werden muss, sollte die Anzahl der Windows auch nicht zu groß sein. In Abbildung 5.11 ist die Laufzeit von bruss2D_nc mit 50.000 Eulerschritten bei 50,250,1.000,10.000 und 25.000 Windows zu sehen. Die beste Laufzeit wird bei 1.000 Windows erzielt, während das Ergebnis bei 50 bzw. 25.000 Windows sehr schlecht ist. Bei weiter steigender Gittergröße sollte die Anzahl der Windows erhöht werden, wie der Graph für 10.000 Windows an-

deutet. Für eine gute Performance darf die Anzahl der Windows demnach weder zu groß noch zu klein gewählt werden. Sie sollte aber bei wachsender Systemdimension erhöht werden.

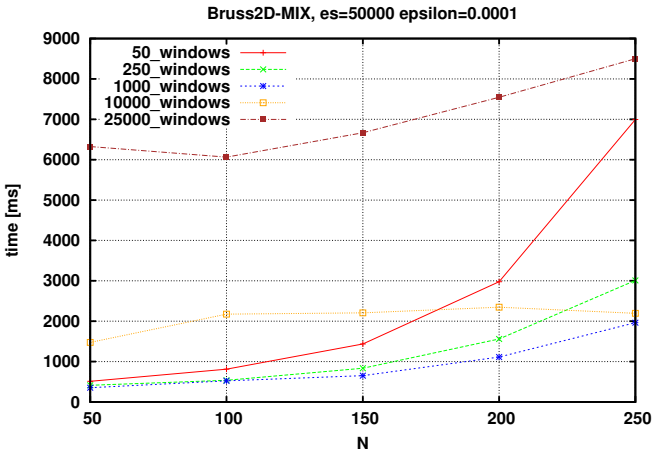

Abbildung 5.11: Laufzeitverhalten von bruss2D_nc bei unterschiedlicher Anzahl an Windows

5.4 Vergleich mit Eulerverfahren

Zum Schluss wird die Effizienz des WR-Verfahrens beurteilt. Dazu wurde ein explizites Eulerverfahren mit CUDA implementiert (solver: cuExplEuler) und die WR-Implementierungen damit verglichen, indem der maximale lokale Fehler des Eulerverfahrens als epsilon-Parameter bei den WR-Implementierungen verwendet wurde. Für die Messungen wurde eine GeForce GTX 980 Ti verwendet.

In Abbildung 5.12 wurde auf diese Weise die Laufzeit vom expliziten Eulerverfahren mit der Laufzeit von bruss_shm und bruss2D_nc verglichen. Dazu ist die Laufzeit sowohl bei unterschiedlicher Anzahl an Eulerschritten $es = 10.000, 60.000, \ldots, 310.000$ als auch unterschiedlichen Gittergrößen $N = 10, 30, \ldots, 90$ (links) bzw. $N = 20, 80, \ldots, 320$ (rechts) gemessen worden. Zudem wurden 1000 Windows verwendet um die Anzahl der WR-Schritte pro Window gering zu halten. Links kann man erkennen, dass die WR-Implementierungen besonders bei großer Anzahl an Eulerschritten und relativ kleiner Gittergröße schneller sind. Bei wachsender Gittergröße wird das Eulerverfahren effizienter, wie rechts zu erkennen ist.

Grund für dieses Verhalten ist, dass beim Eulerverfahren für jeden Eulerschritt ein Kernel gestartet wird, was einen gewissen Overhead mit sich bringt. Beim WR-Verfahren werden dagegen alle Eulerschritte eines WR-Schritts in einem Kernel berechnet. Daher entfällt beim WR-Verfahren der Overhead für das Starten der zusätzlichen Kernels,

dafür werden aber i.d.R. mehrere WR-Schritte zur Erfüllung des Konvergenzkriteriums benötigt.

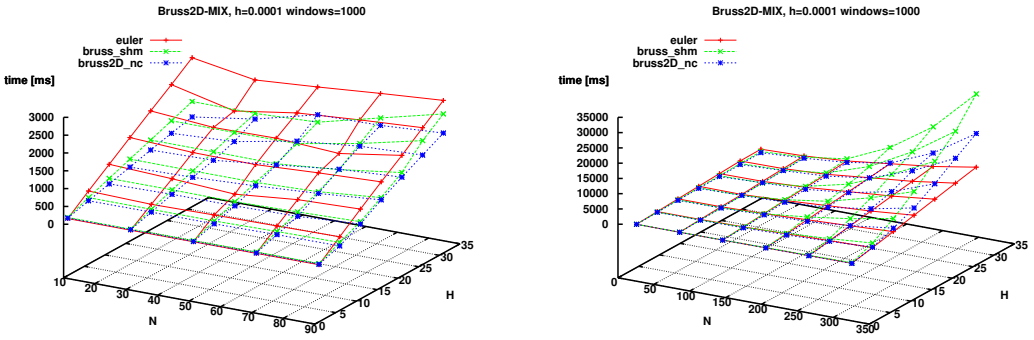

Abbildung 5.12: Vergleich von bruss2D_nc und bruss_shm mit Eulerverfahren

In Abbildung 5.13 wurde der Vergleich mit fester Gittergröße $N = 50$ gemacht. Hier ist schön zu sehen, dass die bruss_shm-Version aufgrund schnellerer Konvergenz die bruss2D_nc Version überholt.

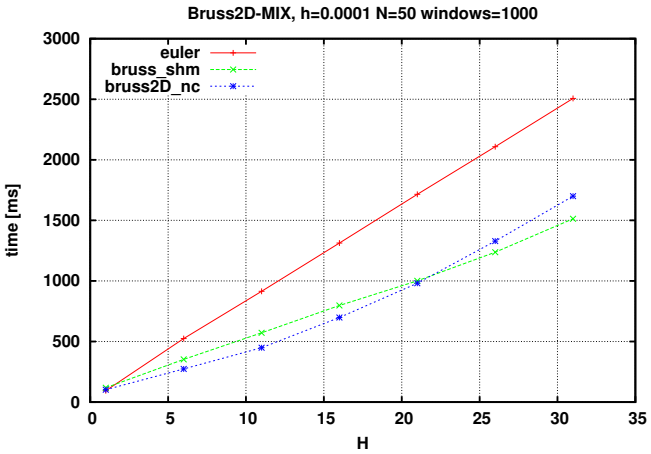

Abbildung 5.13: Vergleich von bruss2D_nc und bruss_shm mit Eulerverfahren bei $N = 50$

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein WR-Verfahren für GPUs mithilfe von CUDA implementiert. Das Verfahren wurde für eine GPU, für mehrere GPUs innerhalb eines Rechners und für alle GPUs innerhalb eines Rechnernetzes umgesetzt. Für die Kommunikation innerhalb des Rechnernetzes wurde MPI verwendet. Zwei Implementierungen (bruss_shm, nc_shm) nutzen die Block-Jacobi-WR-Methode, während alle anderen Versionen die Jacobi-WR-Methode verwenden. Bei allen Versionen wurde das Windowing des Integrationsintervalls umgesetzt und als Zeitschrittverfahren das explizite Eulerverfahren mit fester Schrittweite genutzt.

Die verschiedenen Implementierungen wurden hinsichtlich Laufzeit, Speicherbedarf, Konvergenzverhalten und Effizienz untersucht, wobei als Testproblem das zweidimensionale Brusselator-Problem gewählt wurde. Effiziente Implementierungen konnten erst durch Ausnutzung der begrenzten Zugriffsdistanz des Problems und entsprechender Anpassung der Auswertungsfunktion erreicht werden. Bei konstanter Anzahl an WR-Schritten wurde die beste Laufzeit unter Verwendung von Vektortypen und damit einhergehender Reduzierung der Warpdivergenz erzielt. Der Speicherbedarf ist dagegen bei allen optimierten Implementierungen identisch. Die Verwendung mehrerer GPUs lohnt sich insbesondere bei großen Systemdimensionen. In diesem Fall kann i.d.R. sowohl Laufzeit als auch Speicherbedarf pro GPU reduziert werden. Bei der Laufzeit ist im Durchschnitt ein Speedup von 50-70 Prozent des optimalen Speedups zu verzeichnen und der Speicherbedarf sinkt linear mit der Anzahl der GPUs. Das Konvergenzverhalten der Implementierungen, welche die Block-Jacobi WR Methode verwenden, ist erwartungsgemäß etwas besser als bei den anderen Implementierungen. Der für die Untersuchung der Effizienz erbrachte Vergleich mit dem Eulerverfahren ergab, dass sich das WR-Verfahren vor allem bei einer großen Anzahl an Eulerschritten und einer relativ geringen Systemdimension lohnt. Zudem darf die Anzahl der Windows nicht zu klein sein.

Es gibt noch weitere Varianten des WR-Verfahrens, welche in dieser Arbeit nicht betrachtet wurden. Insbesondere kann man das verwendete Zeitschrittverfahren anpassen. Das Eulerverfahren kann etwa eine dynamische Schrittweitensteuerung nutzen. Auch können Runge-Kutta-Verfahren höherer Ordnung statt des Eulerverfahrens als Zeitschrittverfahren verwendet werden.

Quellen

[1] K. Burrage. Parallel methods for systems of ordinary differential equations. *SIAM News, V.26, N.5*, 1993.

[2] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press Inc., New York, 1995.

[3] NVIDIA. Whitepaper NVIDIA GeForce GTX 980, 2014. URL `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`. Besucht am 30.9.2016.

[4] NVIDIA. NVRTC, 2016. URL `http://docs.nvidia.com/cuda/nvrtc/index.html#axzz4KRmVG2aA`. Besucht am 17.9.2016.

[5] NVIDIA. Profiler User Guide, 2016. URL `http://docs.nvidia.com/cuda/profiler-users-guide/#axzz4Nibd1bLC`. Besucht am 21.10.2016.

[6] NVIDIA. CUDA C Programming Guide, 2016. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide`. Besucht am 16.9.2016.

[7] T. Schödel. Parallele Implementierung und Analyse eines Waveform-Relaxationsverfahrens. Bachelorarbeit, Universität Bayreuth, 2016.

[8] S. Vandewalle. *Parallel Multigrid Waveform Relaxation for Parabolic Problems*. B.G. Teubner, Stuttgart, 1993.

Digitale Abgabe

Die digitale Abgabe enthält die Ausarbeitung in pdf-Form mit den zur Erstellung benötigten Latex-Dateien im Ordner Skript sowie den aktuellen Stand des Repositories im Ordner 2016-fiebig-e2l.

Im Repository ist der Quellcode der Single-GPU-Versionen, Multi-GPU-Versionen, MPI-Versionen und des Eulerverfahrens in den Verzeichnissen /src/lib/solvers/cuWR, /src/lib/solvers/cuWR_mgpu, /src/lib/solvers/dm_cuWR bzw. /src/lib/solvers/cuExplEuler zu finden.

Selbstständigkeitserklärung

Hiermit versichere ich, Alexander Fiebig, dass ich die vorliegende Arbeit und Implementierung selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe. Des Weiteren versichere ich, dass diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht wurde.

Bayreuth, den 16.12.2016

Alexander Fiebig