FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Lehrgebiet Parallelität und VLSI

Parallelisierung des Particle-in-cell-Codes PATRIC mittels GPU-Programmierung

Diplomarbeit in Informatik

vorgelegt von:

Jutta Fitzek
Im Oberen Rech 14
64823 Groß-Umstadt
Matrikelnummer 7266545

Betreuer:	Dr. Sabrina Appel GSI Helmholtzzentrum für Schwerionenforschung
Erstgutachter:	Prof. Dr. Jörg Keller FernUniversität in Hagen
Zweitgutachter:	Prof. Dr. Oliver Boine-Frankenheim Technische Universität Darmstadt

Darmstadt, im September 2013

Entstanden am
GSI Helmholtzzentrum für Schwerionenforschung

GSI

Zusammenfassung

Auf Grund von Grenzen in der Weiterentwicklung der Computerhardware in Bezug auf die Geschwindigkeit und Forderungen nach der Lösung immer größerer Berechnungsprobleme spielen parallele Architekturen und Programmiermethoden eine immer wichtigere Rolle in der Informatik. Neben verteilter Verarbeitung werden auch immer häufiger lokale Ressourcen des einzelnen Knotens wie Grafikkarten für Datenparallelität genutzt. In der physikalischen Grundlagenforschung sind Simulationen häufig ressourcenintensiv und es wird nach Wegen gesucht, Berechnungen zu beschleunigen. Die vorliegende Diplomarbeit befasst sich mit einem sog. Particle-in-cell Verfahren, welches dazu genutzt wird, Teilchen in einem ringförmigen Teilchenbeschleuniger am GSI Helmholtzzentrum für Schwerionenforschung in Darmstadt zu simulieren. Es wird untersucht, wie ein bestehender Algorithmus, der bereits MPI verwendet, so abgeändert werden kann, dass zusätzlich zu der verteilten Berechnung auf den einzelnen Knoten lokal vorhandene Grafikprozessoren integriert werden können. Zum Einsatz kommen dabei NVIDIA GPUs, die mit Hilfe von CUDA C programmiert werden. Mehrere Modifikationen des Algorithmus werden diskutiert, erprobt und bewertet. Das Ergebnis dieser Arbeit ist ein modifizierter und gut wartbarer Algorithmus, der die aktuellen Simulationen schneller durchführen kann und als Basis für weitere Entwicklungen dienen wird. Des Weiteren werden die Möglichkeiten und Grenzen des Einbezugs der GPU aufgezeigt.

Abstract

Due to speed boundaries in computer hardware and demands to solve ever-growing computational problems, parallel architectures and programming methods play a more and more important role in computer science. Besides distributed computing techniques, local resources of the nodes such as GPUs are exploited for data parallelism. In physics research computer simulations are often resource-intense, and researchers seek for a speedup of their computations. This thesis deals with a particle-in-cell simulation code used to simulate particles in a circular particle accelerator at the GSI Helmholtzzentrum for Heavy Ion Research in Darmstadt, Germany. It is evaluated, how an existing algorithm that already uses MPI for parallelism can be modified in such a way, that additionally local GPUs on the distributed nodes can be integrated. NVIDIA GPUs are programmed using CUDA C. Several modifications of the present algorithm are discussed, tested and evaluated. The result of this thesis consists of a modified and well-maintainable algorithm that allows for a faster simulation and will be the basis for future development. Furthermore, the possibilities and limits of GPU integration are being elaborated.

Inhaltsverzeichnis

Tabellenverzeichnis **vii**

Abbildungsverzeichnis **ix**

Quelltextverzeichnis **xi**

Abkürzungen und Symbole **xiii**

1 Einleitung **1**
1.1 Motivation und Problemstellung 1
1.2 Aufbau der Arbeit 3

2 Grundlagen **5**
2.1 Beschreibung der Teilchenbewegung 6
2.1.1 Physikalische Grundlagen 6
2.1.2 Simulation der Teilchenbewegung 7
2.2 Parallelverarbeitung 11
2.2.1 Parallele Rechnerarchitekturen 11
2.2.2 Modelle für Parallelrechner 13
2.2.3 Maße für Parallelität 15
2.2.4 Entwurf paralleler Algorithmen 17
2.2.5 Parallele Softwareentwicklung 21
2.3 Parallelität mit Hilfe von Grafikprozessoren 24
2.3.1 Historie 24
2.3.2 Programmierkonzepte 25
2.3.3 Aufbau einer Grafikkarte 26
2.3.4 Softwareentwicklung 30
2.3.5 Theoretische Einordnung 34
2.4 Mathematische Genauigkeit von Grafikprozessoren 35
2.4.1 Darstellung von Gleitkommazahlen 36
2.4.2 Fehlerbetrachtung 36

3 Simulation mit den bestehenden Particle-in-cell Codes **41**
3.1 Das Simulationsprogramm PATRIC 41
3.1.1 Übersicht und Aufbau 41
3.1.2 Ablauf einer Teilchensimulation 43
3.2 Das Simulationsprogramm LOBO 45
3.2.1 Übersicht und Aufbau 45

3.2.2 Ablauf einer Teilchensimulation 45

4 Parallelisierungsstrategien anderer Particle-in-cell Codes 49
4.1 Einsatz von GPUs im Simulationsprogramm ELEGANT 49
4.1.1 Parallelisierung mit MPI 50
4.1.2 Parallelisierung mit GPUs 51
4.1.3 Bewertung 53
4.2 Einsatz von GPUs im Simulationsprogramm PIConGPU 55
4.2.1 Parallelisierung mit MPI 55
4.2.2 Parallelisierung mit GPUs 56
4.2.3 Bewertung 57
4.3 Einsatz von GPUs in anderen Simulationsprogrammen 58

5 Modifikation des Particle-in-cell Codes 61
5.1 Entwicklungsumgebung 62
5.2 Unverändertes PATRIC Programm 63
5.2.1 Messungen mit MPI 63
5.3 Vorüberlegungen zum Einbezug der GPU 65
5.3.1 Verwendete Datenstrukturen 65
5.3.2 Verwendeter Speicher für statische Daten 66
5.3.3 Verwendeter Speicher für dynamische Daten 67
5.4 Tracking mit PATRIC 68
5.4.1 Einfacher Ansatz, Transportschritt auf der GPU 69
5.4.2 Teilchen auf der GPU halten 71
5.4.3 Ausgabe von Zwischenergebnissen 74
5.4.4 Berechnung von Strahlgrößen auf der GPU 76
5.4.5 Gleitkommadarstellung und Genauigkeit 79
5.5 Kollektive Effekte 80
5.5.1 Messungen mit dem Originalprogramm LOBO 80
5.5.2 Interpolation mit atomaren Operationen 81
5.5.3 Interpolation mit Vorsortierung 83
5.6 Integration der vorgenommenen Parallelisierungen 84
5.7 Einbeziehung mehrerer MPI-Knoten 85

6 Diskussion 87
6.1 Zusammenfassung und Bewertung der Ergebnisse 87
6.2 Ausblick 89

Anhang 91
A Programmierung 91
A.1 Übersicht über die CUDA C Spracherweiterungen 91
A.2 Zeitmessung 92
A.3 Programmauszüge 93
B Formeln 104
B.1 Transportmatrizen 104
B.2 Fehlerformeln 105

C Simulationsprogramme 106
C.1 Die Konfigurationsdatei von PATRIC 106
C.2 Die Konfigurationsdatei von LOBO 107
D Inhalt der beigefügten DVD 108

Literaturverzeichnis **109**

Selbständigkeitserklärung **115**

Tabellenverzeichnis

5.1 Spezifikation der verwendeten Grafikkarte 62
5.2 Transportschritt auf der GPU: Koordinate oder Teilchen pro Thread . . . 70
5.3 Teilchen auf der GPU: Vergleich Anzahl Threads pro Block 71
5.4 Teilchen auf der GPU: Laufzeitvergleich bei 50% Teilchenverlust 73
5.5 Teilchen auf der GPU: Laufzeitvergleich bei der Nutzung von Streams . 75
5.6 Teilchen auf der GPU: Vergleich einfache und doppelte Genauigkeit . . . 79
5.7 Teilchen und Gitter auf der GPU: Laufzeiten der einzelnen Schritte . . . 82
5.8 Teilchen und Gitter auf der GPU: Vergleich Gittergrößen. 83
5.9 Teilchen und Gitter auf der GPU: Vergleich mit Vorsortierung 84

Abbildungsverzeichnis

2.1 Beschreibung der Teilchen eines Teilchenpakets 8
2.2 Berechnungszyklus einer PIC-Simulation 10
2.3 Shared-memory Modell, das auch der PRAM zu Grunde liegt 14
2.4 Schritte beim Entwurf paralleler Algorithmen 18
2.5 EVA-Prinzip vor dem Hintergrund der Parallelverarbeitung 23
2.6 CUDA Ausführungsmodell 26
2.7 Aufbau einer Tesla C2075 Grafikkarte 27
2.8 CUDA Speichermodell 29
2.9 Gleitkommazahlen einfacher und doppelter Genauigkeit 36

3.1 Objekte des PATRIC Simulationsprogramms 42
3.2 Aufteilung der Teilchen auf MPI-Knoten in PATRIC 43
3.3 Objekte des LOBO Simulationsprogramms 46

4.1 Interpolation im Programm ELEGANT 52
4.2 Datenzerlegung im Programm PIConGPU 56

5.1 Ausführungszeit des Originalprogramms PATRIC 64
5.2 Prozessorzeit des Originalprogramms PATRIC 64
5.3 Darstellung der Teilchen als Strukturen oder Arrays 65
5.4 Ausgabe des Profilers zum Originalprogramm PATRIC 68
5.5 Vergleich Modifikation in PATRIC: Teilchen auf der GPU halten 72
5.6 Vergleich Modifikation in PATRIC: Variierung der Ausgabeschritte . . . 74
5.7 Vergleich Modifikation in PATRIC: Überlappende Ausgabe mit Streams 75
5.8 Schema der Reduktion mittels sequentieller Adressierung 77
5.9 Vergleich Modifikation in PATRIC: Emittanzermittlung auf der GPU . . 78
5.10 Ausgabe des Profilers zum Originalprogramm LOBO 80
5.11 Vergleich Modifikation in LOBO: Interpolation mit atomaren Operationen 82

Quelltextverzeichnis

2.1 Thrust Beispielaufruf 33

3.1 PATRIC Originalversion: Zentrale Schleife des Teilchentransports 44
3.2 LOBO Originalversion: Zentrale Schleife der Berechnung 47

5.1 PRAM Pseudocode: Transportschritt auf der GPU 69
5.2 PRAM Pseudocode: Summierung mittels Reduktion 76

A.1 Zeitmessung der CPU-Programmausführung 92
A.2 Zeitmessung der GPU-Programmausführung 92
A.3 PATRIC: Transportschritt auf der GPU 93
A.4 PATRIC: Transportschritt auf der GPU, konstanter Speicher 94
A.5 PATRIC: Teilchendaten auf der GPU, Datenstruktur 95
A.6 Nutzung von Streams: Definition und Verwendung 96
A.7 Berechnung der Emittanz auf der GPU mit CUDA 97
A.8 Berechnung der Emittanz auf der GPU mit Thrust 99
A.9 LOBO: Kernel zur Bewegung der Teilchen 100
A.10 LOBO: Interpolation, Variante mit atomaren Operationen 101
A.11 LOBO: Variante mit Sortierung, Sortiermechanismus 102
A.12 LOBO: Variante mit Sortierung, Interpolation 103

Abkürzungen und Symbole

FAIR	Facility for Antiproton and Ion Research
GSI	GSI Helmholtzzentrum für Schwerionenforschung GmbH
LOBO	Longitudinal Beam Dynamics Simulations Code, Abteilung Strahlphysik, GSI
PATRIC	PArticle TRackIng Code, Abteilung Strahlphysik, GSI
ELEGANT	ELEctron Generation ANd Tracking Programm, Argonne National Laboratory, USA
PIConGPU	Particle in Cell Code on the GPU, Helmholtz-Zentrum Dresden-Rossendorf
CUDA	Compute Unified Device Architecture, eine von NVIDIA entwickelte parallele Architektur für die Grafikverarbeitung
cuFFT	FFT-Bibliothek für die GPU von NVIDIA
EVA	Eingabe-Verarbeitung-Ausgabe, Prinzip der rechnergestützten Datenverarbeitung
GPU	Graphics Processing Unit, Grafikprozessor
MPI	Message Passing Interface, Standard für den Nachrichtenaustausch in verteilten Computersystemen
PIC	Particle-in-cell Verfahren
PRAM	Parallel Random Access Machine, Modell für einen Parallelrechner
B	Magnetisches Feld
E	Elektrisches Feld
e	Elektrische Elementarladung ($e = 1.6022 \cdot 10^{-19} C$)
F	Lorentzkraft
ϵ	Emittanz

1 Einleitung

Auf Grund von wachsenden Anforderungen zur Lösung immer größerer Berechnungsprobleme aber andererseits dem Erreichen von Grenzen in der Weiterentwicklung der Computerhardware spielen parallele Architekturen und Programmiermethoden eine immer größere Rolle in der Informatik. Neben verteilter Verarbeitung werden auch immer häufiger lokale Ressourcen wie Grafikkarten für Datenparallelität genutzt.

Ziel der vorliegenden Diplomarbeit ist es, bestehende Programme zur Simulation von Teilchen in einem Teilchenbeschleuniger unter Einsatz von Grafikkartenprogrammierung zu parallelisieren.

1.1 Motivation und Problemstellung

In der physikalischen Grundlagenforschung sind Simulationen häufig sehr ressourcenintensiv und es wird nach Wegen gesucht, Berechnungen zu beschleunigen. Die vorliegende Diplomarbeit befasst sich mit bestehenden Programmen, die dazu genutzt werden, Teilchen in einem ringförmigen Teilchenbeschleuniger am GSI Helmholtzzentrum für Schwerionenforschung GmbH (GSI) in Darmstadt zu simulieren.

Teilchenbeschleuniger werden in der Forschung, in der Medizin und auch in der Industrie weltweit eingesetzt. In der physikalischen Grundlagenforschung dienen sie dazu, den Aufbau der Materie zu erforschen. Simulationen spielen zur Beschreibung der Teilchenbewegung eine essentielle Rolle, um Effekte mit vorgegebenen Parametern speziell untersuchen zu können. Simuliert wird dabei über die Zeit oder den Weg der Teilchenbewegung, welche ihrerseits durch den Aufbau des Beschleunigers und durch Effekte der Teilchen untereinander und mit ihrer Umgebung bestimmt ist.

Ziel ist einerseits das Verstehen der Teilchenbewegung in der bestehenden Beschleunigeranlage, hierbei werden die Ergebnisse der Simulationen den real gemessenen Größen gegenübergestellt. Andererseits werden im Rahmen der Entwicklung neuer Teilchenbeschleuniger auch Simulationen für noch nicht gebaute Beschleunigeranlagen durchgeführt. Im Rahmen eines internationalen Projekts entsteht aktuell unter dem Namen Facility for Antiproton and Ion Research (FAIR) eine neue internationale Beschleunigeranlage zur Forschung mit Antiprotonen und Ionen, für die die bestehende GSI Beschleunigeranlage als Injektor dient. In diesem Zusammenhang werden weitreichende Simulationen durchgeführt, deren Ergebnisse Eingang in die Bauplanung finden.

Für die genannten Simulationen wird das Particle-in-cell (PIC) Programm PATRIC (Particle Tracking Code) verwendet. Das Simulationsprogramm wurde in der Abteilung Strahlphysik der GSI entwickelt und ist speziell auf die Fragestellungen zugeschnitten, die sich an den Beschleunigeranlagen der GSI ergeben. Trotz einer Konzentration auf die relevanten Fragestellungen und eines modernen, strukturierten Programms können sich allein auf Grund der zu simulierenden Teilchenmengen und Betrachtungszeiträume lange Simulationsdauern ergeben. Der Wunsch nach einer Beschleunigung der Simulationen ist dabei nicht neu. Im bestehenden Programm PATRIC sind bereits seit 2001 Mechanismen zur verteilten parallelen Berechnung mittels Message Passing Interface (MPI) eingebaut. Damit ist es möglich, die betrachtete Teilchenmenge aufzuteilen und von mehreren Knoten parallel bearbeiten zu lassen, was bei allen länger laufenden Simulationen im Rahmen der vorhandenen Rechnerressourcen auch genutzt wird (typischerweise mit 4-16 Berechnungsknoten). Trotz des Einsatzes von MPI können lang laufende Simulationen mehrere Stunden dauern. Der Wunsch ist deshalb, die Berechnungen auf den einzelnen verteilten Berechnungsknoten weiter zu beschleunigen, um Simulationsergebnisse insgesamt frühzeitiger vorliegen zu haben.

Aktuell werden in der Industrie und in der Forschung zur Unterstützung von rechenintensiven Berechnungen vermehrt Grafikprozessoren (GPUs) eingesetzt. Diese versprechen einen gegenüber einem herkömmlichen Prozessor stark verbesserten Durchsatz bei der massenhaften Ausführung immer gleicher Gleitkomma-Berechnungen mit verschiedenen Daten, so wie es z. B. bei Simulationen vieler Teilchen der Fall ist. In der Forschung gibt es aus diesem Grund mehr und mehr Projekte, die vorhandene Berechnungen auf Grafikkarten portieren; in der Industrie gibt es bereits viele Anwendungen, die für berechnungsintensive Aufgaben Grafikkarten nutzen. Vor diesem Hintergrund entstand in der Abteilung Strahlphysik die Idee, Teile des Simulationsprogramms PATRIC auf Grafikkarten zu portieren, um eine weitere Verkürzung der Simulationsdauer zu erzielen.

Dazu war es Aufgabe im Rahmen der Diplomarbeit zu untersuchen, wie die bestehenden Simulationsprogramme so abgeändert werden können, dass zusätzlich auf den einzelnen MPI-Knoten eine lokal vorhandene Grafikkarte integriert werden kann. Neben dem Programm PATRIC, bei dem einzelne ausgewählte Teile auf die GPU portiert wurden, wird auch das ebenfalls von der Abteilung Strahlphysik entwickelte Programm LOBO (Longitudinal Beam Dynamics Simulations Code) betrachtet, bei dem ein größerer Anteil auf die GPU portiert wurde. Als Vorarbeit wurden dabei zunächst an Hand anderer Simulationsprogramme, die bereits eine GPU-Unterstützung enthalten, generelle Möglichkeiten zur Einbindung von Grafikkarten ermittelt.

Für die eigenen Implementierungen war als Laufzeitumgebung eine Grafikkarte des Herstellers NVIDIA vorgegeben. Die Präferenz für NVIDIA im Umfeld der Teilchensimulationen an der GSI ist vor allem in der breiten Unterstützung durch frei verfügbare Bibliotheken (wie cuFFT, cuBLAS etc.) begründet. Die Ausführungen im Grundlagenkapitel und die Implementierungen im praktischen Teil der Arbeit konzentrieren sich aus diesem Grund auf NVIDIA Grafikkarten; die Änderungen an den bestehenden Simulationsprogrammen wurden mit CUDA C implementiert.

Im Rahmen der Arbeit werden mehrere Modifikationen des bestehenden Algorithmus vorgeschlagen, diskutiert, getestet und gemessen. Das Ergebnis dieser Arbeit ist ein modifizierter Algorithmus, der die aktuellen Simulationen schneller durchführen kann und als Basis für weitere Entwicklungen dienen wird. Des Weiteren werden die Möglichkeiten und Grenzen des Einbezugs der GPU aufgezeigt.

1.2 Aufbau der Arbeit

Kapitel 2 behandelt die Grundlagen, auf denen diese Arbeit aufbaut. Es wird zunächst auf die Simulation von Teilchen in Beschleunigern eingegangen, soweit es zum Verständnis der Arbeit erforderlich ist. Es folgt eine detaillierte Darstellung der Grundkonzepte der Parallelisierung und der im Rahmen dieser Arbeit eingesetzten Methoden. Einen Schwerpunkt bildet dabei am Ende des Kapitels die Betrachtung von Grafikprozessoren.

In Kapitel 3 werden die betrachteten Teilchensimulationsprogramme PATRIC und LOBO in ihrer bestehenden Version beschrieben. Es wird als Überleitung zum praktischen Teil der Arbeit angedeutet, an welchen Stellen generell Parallelisierungsmöglichkeiten bestehen.

In Kapitel 4 werden zur Darstellung des aktuellen Stands der Forschung der Einsatz von Grafikprozessoren in anderen Teilchensimulationsprogrammen beleuchtet.

Den Schwerpunkt dieser Arbeit bildet die Einbindung der GPU in die bestehenden Simulationsprogramme. In Kapitel 5, dem Kernstück der Arbeit, werden deshalb verschiedene Modifikationen der Programme PATRIC und LOBO betrachtet. Dabei werden Möglichkeiten untersucht, die GPU möglichst optimal auf das vorliegende Problem zugeschnitten einzusetzen. Die verschiedenen Modifikationen werden jeweils vorgeschlagen und theoretisch diskutiert; anschließend werden die gewonnenen Messergebnisse vorgestellt und bewertet. Ein kurzer Ausblick am Ende des Kapitels befasst sich mit der Einbeziehung der verteilten Berechnung mittels MPI, welche vorher ausgeklammert wurde.

Das Ende der Arbeit bildet eine abschließende Zusammenfassung und Bewertung der Ergebnisse in Kapitel 6. Vor diesem Hintergrund wird am Schluss ein Ausblick auf zukünftige Weiterentwicklungen gegeben.

2 Grundlagen

Zur Simulation der Teilchenbewegung im Beschleuniger werden am GSI Helmholtzzentrum für Schwerionenforschung GmbH (GSI) selbst entwickelte Programme genutzt. Die damit durchgeführten Simulationen spielen für das physikalische Verständnis der Strahlphysik eine bedeutende Rolle und zeitnahe Ergebnisse der Simulationsläufe sind dabei eine wichtige Voraussetzung. Im Rahmen dieser Arbeit soll untersucht werden, wie die Programme und damit die implementierten Algorithmen unter Einsatz von Grafikkartenprogrammierung parallelisiert werden können.

Als Grundlage ist es wichtig zu verstehen, was Gegenstand der Simulationen ist. Als erstes wird deshalb kurz die dahinterstehende Physik skizziert. Ausgehend von der Beschreibung der Teilchenbewegung in einem Beschleuniger wird erklärt, wie die Teilchen selbst und die Beschleunigerelemente in den Simulationen abgebildet werden. Darauf aufbauend wird auf den generellen Ablauf einer Teilchensimulation eingegangen.

Als Basis für die Implementierung ist es zunächst notwendig, auf die Grundlagen der Parallelisierung einzugehen. Aufbauend auf parallelen Rechnerarchitekturen wird das verwendete Modell für Parallelrechner vorgestellt und die genutzten Messgrößen für parallele Programme erläutert. Als Basis für die parallele Softwareentwicklung wird die Methode zum Entwurf paralleler Algorithmen vorgestellt, die im Rahmen der eigenen Implementierungen eingesetzt wurde. Ein kurzer Abschnitt zu MPI (Message Passing Interface) bildet die Grundlage zum Verständnis der bereits im Programm PATRIC bestehenden Ansätze zur Parallelisierung.

Ziel dieser Arbeit ist die Einbindung der Grafikkarte in die Berechnungen. Um die dadurch bestehenden Möglichkeiten der Parallelisierung zu verstehen, müssen die mit der Grafikkarte verbundenen Programmierkonzepte betrachtet werden. Vorgestellt wird als Basis der Aufbau einer NVIDIA Grafikkarte mit Fokus auf die vorhandenen Ausführungseinheiten und der Speicherhierarchie. Darauf aufbauend wird das mit CUDA (Compute Unified Device Architecture) zur Verfügung stehende Programmiermodell dargestellt, das als Grundlage für die Parallelisierung dient. Vor diesem Hintergrund wird auf einzelne Aspekte der Programmierung eingegangen, die im Rahmen des praktischen Teils eine Rolle spielen.

Ein Randthema bildet die Fragestellung nach der Genauigkeit der durchgeführten Simulationen im Hinblick auf den Einsatz von Grafikkarten, weshalb im Rahmen der Grundlagen die Gleitkommadarstellung in Rechnern und die damit verbundene mögliche Fehlerfortpflanzung angesprochen wird.

2.1 Beschreibung der Teilchenbewegung

2.1.1 Physikalische Grundlagen

Bevor auf die Simulation mit der Particle-in-cell Methode eingegangen wird, soll zunächst kurz die dahinterstehende Physik erläutert werden, insofern es für das Verständnis der Arbeit notwendig erscheint. Dazu wird den Ausführungen in [Wil96] gefolgt.

In einem Teilchenbeschleuniger werden Teilchen mit Hilfe von elektromagnetischen Kräften beschleunigt und auf der gewünschten Bahn gehalten. Ein Teilchen besitzt die Ladung $q = Z \cdot e$, dem Produkt aus der Ladungszahl Z und der Elementarladung e. Wenn dieses elektrisch geladene Teilchen mit der Geschwindigkeit $\overrightarrow{v}$ einen Raum durchfliegt, in dem das magnetische Feld $\overrightarrow{B}$ und das elektrische Feld $\overrightarrow{E}$ herrschen, wirkt auf das Teilchen die Lorentzkraft

$$\overrightarrow{F} = q \cdot (\overrightarrow{v} \times \overrightarrow{B} + \overrightarrow{E}). \tag{2.1.1}$$

Das elektrische Feld ermöglicht eine Energieänderung und damit eine Beschleunigung der Teilchen in longitudinaler Strahlrichtung. Das magnetische Feld hat eine transversale, rein ablenkende bzw. bahnführende Wirkung, da das magnetische Feld senkrecht zur Bewegungsrichtung des Teilchens steht [vgl. Wil96, S. 3f.]. N gleichartige Teilchen bilden dabei den Teilchenstrahl, der entweder gleichförmig ist oder in Teilchenpakete aufgeteilt vorliegt.

Der Schwerpunkt der Untersuchungen in der Abteilung Strahlphysik liegt auf Kreisbeschleunigern. Einfache Kreisbeschleuniger besitzen viele Magnete, um die Teilchen transversal abzulenken und somit auf ihrer Bahn zu halten und eine Hochfrequenzanlage, um die Teilchen longitudinal zu beschleunigen. Bei dem bei der GSI vorhandenen Kreisbeschleuniger handelt es sich um ein sog. Synchrotron. Wie der Name andeutet, muss bei diesem Beschleunigertyp das Magnetfeld synchron mit der Energie hochgefahren werden, um die Teilchen während ihrer Beschleunigung auf der Bahn zu halten [vgl. Wil96, S. 23ff.]. Daneben gibt es an der GSI einen Speicherring. Dieser arbeitet nach demselben Prinzip, jedoch werden hierbei die Teilchen über einen längeren Zeitraum gespeichert und evtl. sogar abgebremst. Im Rahmen des Aufbaus der neuen FAIR-Anlage werden bei der GSI weitere Kreisbeschleuniger hinzukommen, diese sind bereits jetzt Gegenstand der Betrachtungen.

Im Rahmen der Diplomarbeit wurde in drei Schritten zuerst als Schwerpunkt der Aspekt der transversalen Teilchenbewegung behandelt, danach wurde die longitudinale Bewegung der Teilchen betrachtet sowie abschließend Überlegungen zur gesamten Teilchenbewegung angestellt.

Um die transversale Bewegung der Teilchen zu verstehen, müssen die magnetischen Felder betrachtet werden. Magnetische Kräfte in einem Beschleuniger werden durch starke Magnete wie Dipole, Quadrupole, Sextupole und Oktupole erzeugt, welche im Beschleuniger eingebaut sind. Diese Magnete haben einen Einfluss auf die transversale

Teilchenbewegung, der sich durch die Methoden der Optik beschreiben lässt, weshalb dabei auch von „Strahloptik“ gesprochen wird. Dipole dienen dazu, Teilchen in einem Kreisbeschleuniger auf der Kreisbahn zu halten. Quadrupole kann man sich als optische Linsen vorstellen; sie dienen der Fokussierung von Teilchen, wobei sie nur in einer Ebene fokussieren (und in der anderen defokussieren). Sextupole und Oktupole dienen zur Kompensation von Abweichungen. Die Struktur der Anordnung der Magnete im Beschleuniger (engl. „lattice“) ist typischerweise symmetrisch, wobei sich fokussierende und ablenkende Magnete abwechseln. Diese Anordnung ist Grundlage für die Berechnung der transversalen Teilchenbewegung. Im Rahmen der Arbeit wurden idealtypische Beschleunigerkomponenten in Form einer vereinfachten linearen Strahloptik (d. h. nur mit Dipolen und Quadrupolen) verwendet.

Um die longitudinale Teilchenbewegung zu verstehen, kann man sich die Bewegung eines ganzen Teilchenpakets auf einer elektromagnetischen Welle vorstellen. Ein Teilchen besitzt eine gewisse Frequenz und Phase und erhält beim Durchlaufen der Hochfrequenzanlage eine bestimmte Energie [vgl. Hin08, S. 308]. Teilchen mit einer Impulsabweichung sehen bei ihrem Eintreffen eine andere Phase und erhalten so weniger oder mehr Energie als das synchrone Teilchen. Die Teilchen nähern sich im Folgenden der Sollphase an und führen Schwingungen um diese aus. Bei kleinen Schwingungsamplituden können diese Phasen- und damit Energieschwingungen durch einen harmonischen Oszillator beschrieben werden. Bei zu großen Schwingungsamplituden wird der stabile Bereich verlassen und die Teilchen gehen verloren [vgl. Hin08, S. 311].

Bisher wurden äußere elektromagnetische Felder beschrieben, die auf die Teilchen einwirken. Daneben existieren auch elektromagnetische Selbstfelder, die durch die Teilchen untereinander und in der Verbindung mit dem umgebenden Beschleuniger erzeugt werden [vgl. Rei08, S. 163ff.]. Die Effekte dieser Felder werden kollektive Effekte genannt. In der physikalischen Betrachtung werden sie in ihre Einzelkomponenten aufgeteilt. Die Stärke dieser Felder hängt von der Teilchendichte ab und ist umso größer, je niedriger die Energie und je höher die Teilchendichte ist [vgl. Hin08, S. 27]. Die Effekte dieser Felder können zu einem Auseinanderdriften der Teilchenbahnen und damit einer Defokussierung des Strahls und einem Anwachsen der Emittanz führen und sogar schlimmstenfalls den gesamten Strahl destabilisieren. Da es gerade im Hinblick auf die neue FAIR-Anlage auf hohe Strahlintensitäten ankommt, müssen diese Effekte genau untersucht werden und spielen deshalb bei den Simulationen eine wichtige Rolle.

2.1.2 Simulation der Teilchenbewegung

Die Simulation der Teilchenbewegung dient dazu, theoretische Annahmen zu überprüfen, Experimente vorzubereiten oder auch zukünftige Beschleunigeranlagen zu entwerfen. Betrachtet werden dabei Makroteilchen, die viele einzelne Teilchen repräsentieren[1]. Ein Teilchen wird dabei relativ zum vorher erwähnten synchronen Teilchen betrachtet. Das synchrone Teilchen ist das zentrale Teilchen eines Teilchenpakets und

[1] Im Folgenden wird dennoch von Teilchen gesprochen, obwohl eigentlich Makroteilchen gemeint sind.

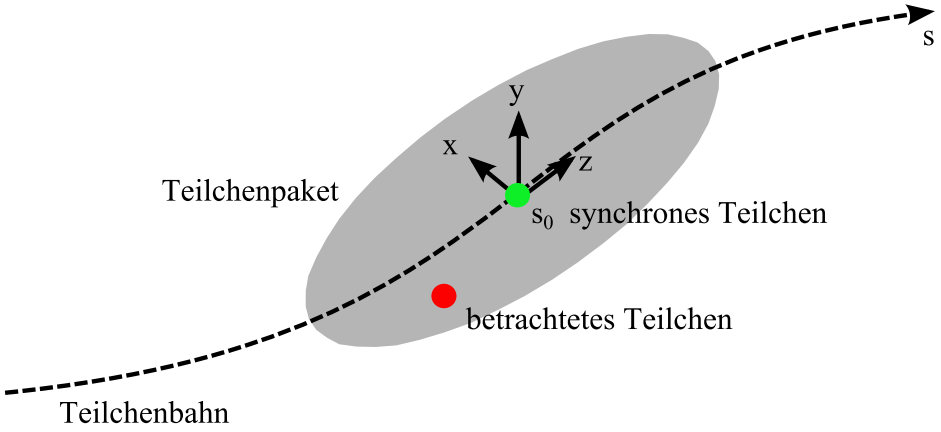

Abbildung 2.1: Die betrachteten Teilchen eines Teilchenpakets werden durch ihre Koordinaten in Bezug auf das synchrone Teilchen beschrieben.

bewegt sich auf der Idealbahn (oder auch Sollbahn) durch den Beschleuniger [vgl. Hin08, S. 117]. Die Idealbahn ist durch die Energie des Teilchens sowie die Stärke der ablenkenden Dipole bestimmt.

Die Position eines Teilchens im Beschleuniger wird durch ein mitbewegtes Koordinatensystem beschrieben, dessen Ursprung sich an der aktuellen Position des synchronen Teilchens auf der Idealbahn befindet. Die x-Achse zeigt horizontal zur Strahlrichtung nach links, die y-Achse stellt die auf der Teilchenbahn vertikal stehende Achse dar, die z-Achse ist die Tangentialachse entlang der Strahlrichtung, siehe Abb. 2.1. Dabei werden x und y als transversale Koordinaten und z als longitudinale Koordinate bezeichnet.

Ein einzelnes Teilchen wird durch einen 6-komponentigen Vektor beschrieben

$$v(s) = \begin{pmatrix} x \\ x' \\ y \\ y' \\ z \\ v \end{pmatrix} \tag{2.1.2}$$

mit x und x' als horizontale Orts- und Richtungsabweichung, y und y' als vertikale Orts- und Richtungsabweichung, z als longitudinale Ortsabweichung und v als Implusabweichung im Vergleich zum synchronen Teilchen. In Einheiten werden x, y, v in mm, x', y' in mrad und v in Promille ausgedrückt [vgl. Hin08, S. 122].

Auf dem Weg durch den Beschleuniger üben die eingebauten Magnete Kräfte auf die Teilchen aus. Diese lassen sich als Transformation des Teilchenvektors ausdrücken. Dabei wird das Koordinatensystem mit Hilfe von Matrizen transformiert, mathematisch handelt es sich um Transfermatrizen, in der Strahlphysik spricht man hingegen von Transportmatrizen, um auf den Transport der Teilchen durch den Beschleuniger

hinzuweisen. Auch hier soll im Folgenden von Transportmatrizen die Rede sein. Eine entsprechende Herleitung des Matrixformalismus findet sich bei [Hin08, S. 123ff.].

Die Bewegung eines Teilchens entlang des Rings wird als Transformation des Teilchenvektors v durch die Transportmatrizen M der durchlaufenen Magnete dargestellt. Das Ergebnis bildet dann die neue Position und Bewegungsrichtung des Teilchens

$$v' = M \cdot v \tag{2.1.3}$$

wobei die Transportmatrix M die Grundform für die lineare Optik

$$M = \begin{pmatrix} M_{11} & M_{12} & 0 & 0 & 0 & M_{16} \\ M_{21} & M_{22} & 0 & 0 & 0 & M_{26} \\ 0 & 0 & M_{33} & M_{34} & 0 & 0 \\ 0 & 0 & M_{43} & M_{44} & 0 & 0 \\ M_{51} & M_{52} & 0 & 0 & 1 & M_{56} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1.4}$$

besitzt. Die beiden quadratischen Untermatrizen stellen die eigentliche transversale Abbildung (radial und axial) dar. Die Elemente M_{16} und M_{26} stellen die Orts- und Winkeldispersion dar, da durch eine Ablenkung auch immer eine Aufspaltung der Teilchenbahnen wie durch ein Prisma erfolgt. Die Elemente M_{51}, M_{52} und M_{56} bilden Weglängenunterschiede auf Grund der unterschiedlichen Position der Teilchen im Strahl ab. Eine Auflistung der Transportmatrizen ist in Anhang B.1 gegeben.

Neben der Beschreibung der externen Kräfte in Form von Matrizen spielen auch die Selbstfelder eines Strahls mit Raumladung eine wichtige Rolle. Dabei gilt es, diese Felder zu berechnen und ihre Auswirkungen auf die Teilchen zu beschreiben. Die Abbildung dieser kollektiven Effekte im Rahmen der Simulation soll im Folgenden skizziert werden.

Um die Berechnungen im Rahmen der Simulation zu reduzieren, kommt zur Beschreibung der Raumladungskräfte, die von den Teilchen ausgehen oder auf diese wirken, ein Gitter zum Einsatz. Statt also den Einfluss jedes Teilchens auf jedes andere Teilchen berechnen zu wollen (sog. n-Body Simulation mit einem Aufwand von $O(n^2)$), werden Felder und Kräfte nur auf den nächstliegenden Gitterpunkten diskretisiert betrachtet. Diese Technik wird Particle-in-cell (PIC) Methode genannt. Die Methode wird bereits seit den 1950er Jahren für Plasmasimulationen eingesetzt [vgl. BL05, S. 3]. Der Begriff Particle-in-cell veranschaulicht, dass sich die einzelnen Teilchen im Rahmen der Simulation in den Gitterzellen aufhalten. Bei sog. 1D-Simulationsprogrammen wird nur der longitudinale Raum betrachtet, bei 2D-Programmen betrachtet man den transversalen Raum und bei 3D-Programmen den vollständigen Raum. Als Mischform existiert auch die sog. 2,5D-PIC-Simulation, hierbei werden die Teilchenkoordinaten in 3D simuliert, für die Berechnung der Felder wird die z-Koordinate in Scheiben zusammengefasst, da es für manche Teilchenverteilungen sehr komplex ist, die Selbstfelder im vollen 3D-Raum zu berechnen.

Der Berechnungszyklus eines diskreten Zeitschritts im Rahmen einer PIC-Simulation ist in Abb. 2.2 dargestellt [vgl. BL05]. Der Startpunkt ist eine Teilchenverteilung im

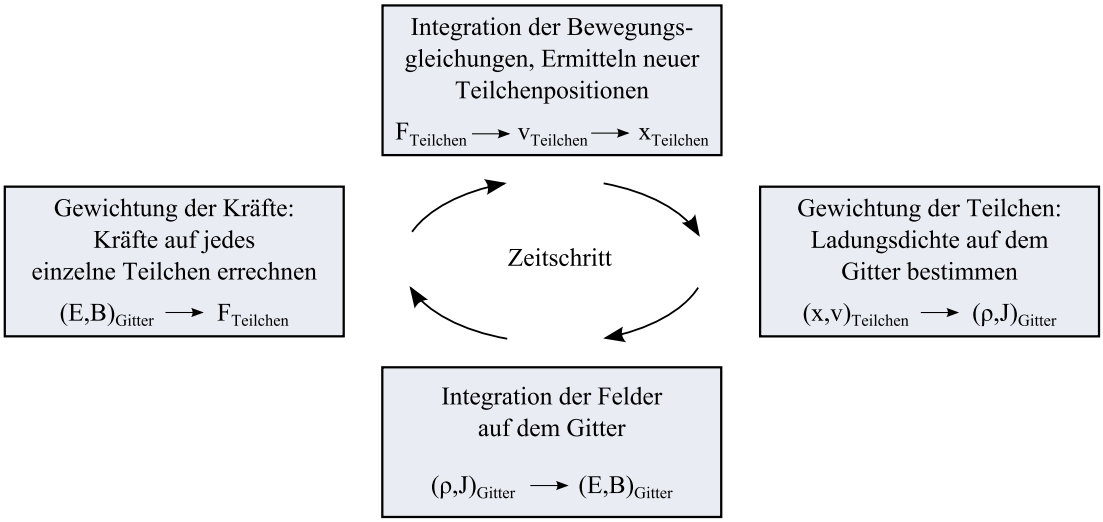

Abbildung 2.2: Berechnungszyklus einer Particle-in-cell Simulation (basierend auf [BL05, Abb. 2-3a]).

Raum. Ausgehend von der Teilchendichte wird im ersten Schritt eine Ladungs- (ρ) und Stromverteilung (J) auf dem Gitter interpoliert. Daraus wird im zweiten (engl. auch „field solver“ bezeichneten) Schritt das elektrische und magnetische Raumladungsfeld auf dem Gitter bestimmt; dies kann z. B. mit Hilfe einer vorwärts- und rückwärts-FFT Funktion durchgeführt werden. Das elektromagnetische Feld wird integriert und ergibt ein elektrostatisches Potential [vgl. Rei08, S. 173]. Im dritten Schritt werden aus dem Potential Kräfte auf die Teilchen bestimmt, die genau wie ablenkende oder beschleunigende Kräfte auf die Teilchen einwirken [vgl. Rei08, S. 164]. Aus diesen Kräften können dann im letzten (engl. auch „particle pusher“ bezeichneten) Schritt die neuen Teilchenpositionen ermittelt werden.

Im Rahmen der Simulationen werden Strahlgrößen ermittelt und ihre Veränderung über den Simulationsverlauf beobachtet. Im Vordergrund steht dabei die Strahlqualität (Güte des produzierten Strahls) und die Strahlintensität (Zahl der Teilchen pro Zeiteinheit) [vgl. Hin08, S. 6ff.]. Ein Parameter, der die Strahlqualität beschreibt, ist die rms-Emittanz (root mean square Emittanz, 1σ-Abweichung). Sie misst die geometrischen Bündelung des Strahls um die Sollbahn, d. h. Strahlbreite mal Divergenz und ist wie folgt definiert [vgl. Rei08, S. 321] bzw. [vgl. Hin08, S. 171]:

$$\epsilon_x = \sqrt{\langle x^2 \rangle \langle x'^2 \rangle - \langle x x' \rangle^2} \tag{2.1.5}$$

Analog wird die Emittanz in y-Richtung definiert (angegeben in mm $\times$ mrad). Angestrebt wird eine möglichst kleine Emittanz. Die Emittanz als Messgröße soll im Folgenden als beispielhafte Messgröße dienen. Ihre Berechnung enthält die typische Anforderung, dass Werte einzelner Teilchen über den gesamten Strahl zusammengefasst werden müssen. Auf die Berechnung weiterer Strahlgrößen wird in dieser Arbeit verzichtet, da sie keine darüber hinausgehenden Anforderungen an die Parallelisierung stellen.

2.2 Parallelverarbeitung

Zum Lösen großer Berechnungsprobleme mit Hilfe von Computern konnte man sich in der Vergangenheit darauf verlassen, dass die Berechnungen mit der nächsten Computergeneration um ein Vielfaches schneller durchführbar waren. So symbolisiert Moores Gesetz aus dem Jahr 1965, nach dem sich die Anzahl der Transistoren auf einem Chip alle 18 Monate verdoppelt, den rasanten technologischen Fortschritt der Computerindustrie. In den letzten Jahren wurden allerdings zunehmend Grenzen erreicht, über die hinaus eine weitere Erhöhung von Taktraten, Verkleinerung von Transistoren oder Verkürzung von Signallaufzeiten nicht mehr unbegrenzt möglich erscheint [vgl. Tan05, S. 43f.]. Um dennoch die Leistung aktueller Computer weiter zu steigern, wird in zunehmendem Maße Parallelverarbeitung eingesetzt.

Parallelverarbeitung bedeutet, dass mehrere Aufgaben gleichzeitig durchgeführt werden können. Bezogen auf die Informatik versteht man darunter die gleichzeitige Ausführung unterschiedlicher Aufgaben durch zwei oder mehr Prozessoren, entweder durch einen Rechner mit mehreren zentralen Ausführungseinheiten oder durch mehrere Rechner, die über ein Kommunikationsnetzwerk verbunden sind (übersetzt aus "The American Heritage Dictionary of the English Language" und [vgl. WA04, S. 5]).

Im Folgenden sollen zunächst parallele Rechnerarchitekturen vorgestellt werden. Im Anschluss wird das Modell für Parallelrechner vorgestellt, welches den theoretischen Betrachtungen und Abschätzungen zu Grunde liegt. Danach folgt eine Auflistung wichtiger Messgrößen für parallele Programme. Nach diesen theoretischen Betrachtungen wechselt der Fokus hin zur Parallelen Entwicklung. Es wird zunächst die eingesetzte Methode zum Entwurf paralleler Algorithmen vorgestellt, bevor kurz auf die praktische Implementierung mittels MPI eingegangen wird (dem Thema GPU ist später ein eigenes Kapitel gewidmet, weshalb die GPU hier noch nicht angesprochen wird). Abschließend werden spezielle Techniken und Modelle angesprochen, die im Rahmen der Arbeit eine Rolle spielen.

2.2.1 Parallele Rechnerarchitekturen

Bei der Parallelverarbeitung kann bei genauerer Betrachtung die eigentliche Parallelität in der Ausführung auf unterschiedlichen Ebenen stattfinden: auf Befehlsebene in Form von Pipelining, auf Prozessorebene in Form von Feldrechnern oder Vektorrechnern, auf Ebene des einzelnen Computers in Mehrprozessorsystemen, oder auf der Ebene vieler Computer in Form von lose gekoppelten Multicomputersystemen [vgl. Tan05, S. 80ff.]. Im Folgenden werden die zwei grundlegenden Arten von Parallelrechnern, die Mehrprozessorsysteme und Multicomputersysteme, sowie das zu Grunde liegende Einprozessorsystem näher betrachtet [vgl. WA04, S. 14ff.].

Ein herkömmlicher Rechner ist ein *Einprozessorsystem*. Er basiert auf der Von-Neumann-Architektur aus den 1940er-Jahren bestehend aus einem zentralen Steuer- und Rechenwerk, einem Speicher für Programm und Daten und einer Ein- und Ausgabeeinheit. Mit einem solchen System ist keine Parallelverarbeitung möglich. Um dem

Benutzer dennoch den Eindruck einer gleichzeigtigen Ausführung mehrerer Aufgaben zu geben, werden diese im Zeitscheibenverfahren nacheinander ausgeführt.

Bei einem *Mehrprozessorsystem mit gemeinsamem Speicher* (engl. shared memory multiprocessor system) werden in einem Rechner mehrere Prozessoren eingesetzt. Alle Prozessoren können auf den gemeinsamen Speicher zugreifen, dieser wird als ein Adressraum zur Verfügung gestellt und beinhaltet Programm und Daten. Je nachdem, ob auf diesen gemeinsamen Speicher von allen Prozessoren gleich schnell zugegriffen werden kann, oder der Zugriff auf entfernte Speicherstellen länger dauert, unterscheidet man dabei die Untertypen UMA und NUMA (engl. für (non) unified memory access). Jeder Prozessor besitzt zusätzlich lokale, schnelle Speicher (sog. Caches) in denen bereits gelesene Daten gehalten werden, um nicht wiederholt auf den gemeinsamen, i. A. langsameren Speicher zugreifen zu müssen. Der Vorteil bei der Programmierung eines solchen Systems liegt darin, dass von jedem Prozessor aus auf alle Daten zugegriffen werden kann. Der Nachteil liegt darin, dass Mechanismen zur Konfliktbehandlung bei gleichzeitigen Zugriffen auf den gemeinsamen Speicher und zur Herstellung von Cache-Kohärenz nötig sind [vgl. WA04, S. 14 ff.].

Bei einem *Multicomputersystem mit verteiltem Speicher* (engl. distributed memory multicomputer system) werden mehrere unabhängige Rechner über ein Verbindungsnetzwerk zusammengeschlossen. Jeder einzelne Rechner kann wiederum ein Ein- oder Mehrprozessorsystem sein, d. h. jeder Rechner des Multicomputersystems hat einen oder mehrere eigene Prozessoren und seinen eigenen lokalen Speicher. Auf den Speicher anderer Rechner kann nicht zugegriffen werden, es handelt sich um sog. verteilten Speicher. Daten werden zwischen den einzelnen Rechnern per Nachrichtenaustausch übertragen. Der Vorteil bei einem Multicomputersystem ist die einfache Skalierbarkeit des Gesamtsystems, es können jederzeit weitere Rechner hinzugefügt werden, der Engpass ist hierbei das Verbindungsnetzwerk. Nachteil ist die kompliziertere Programmierung des Nachrichtenaustauschs. Außerdem müssen Überlegungen zur Einbettung in die bestehende Netzwerktopologie bereits beim Algorithmenentwurf einbezogen werden.

Die beiden oben beschriebenen Arten von Parallelrechnern stellen die beiden klassischen Kategorien dar. Da die Art des Speichers – also gemeinsamer Speicher (shared memory) oder verteilter Speicher (distributed memory) – losgelöst ist von der Rechnerarchitetur selbst, gibt es weitere Mischformen. So kann mit Hilfe einer Softwareabstraktionsschicht auch auf einem Multicomputersystem dem Entwickler gegenüber der Eindruck eines gemeinsamen Speichers vermittelt werden (sog. distributed shared memory), genauso können umgekehrt auf einem Mehrprozessorsystem die einzelnen Prozesse über Nachrichten kommunizieren, ohne auf einem gemeinsamen Speicher aufzusetzen.

Neben einer Beschreibung des Aufbaus als Mehrprozessor- oder Multicomputersystem und einer Einordung nach Art des Speichers als gemeinsamer oder verteilter Speicher können Parallelrechner allgemein nach ihren Befehlsströmen (einer oder mehrere: single/multiple instruction stream) und ihren Datenströmen (einer oder mehrere:

single/multiple data stream) mittels der Klassifikation von Flynn (1966) kategorisiert werden [vgl. Tan05, S. 621]:

- SISD (ein Befehlsstrom, ein Datenstrom): klassischer Einprozessorrechner mit Von-Neumann-Architektur
- SIMD (ein Befehlsstrom, mehrere Datenströme): Feldrechner oder Vektorrechner, die Steuereinheit gibt hierbei einen Befehl vor, der mit einer Menge von Daten gleichzeitig operiert, typischerweise liegen die Daten in Form eines Arrays vor
- MISD (mehrere Befehlsströme, ein Datenstrom): keine Realisierung bekannt
- MIMD (mehrere Befehlsströme, mehrere Datenströme): hierzu zählen die oben beschriebenen Multiprozessor- und Multicomputersysteme

Aufbauend auf der Klassifikation nach Flynn existieren heute weitere Unterklassen auf die speziell im Zusammenhang mit der GPU in Kap. 2.3.5 eingegangen wird.

Im Rahmen dieser Arbeit sind unterschiedliche parallele Architekturen von Interesse. Das Simulationsprogramm PATRIC beinhaltet bereits eine Implementierung für Multicomputersysteme, wobei die einzelnen Knoten per Nachrichtenaustausch kommunizieren. Dieser Aufbau mit mehreren eigenständigen Rechnern gehört zur Kategorie MIMD. Die im Rahmen dieser Arbeit betrachtete GPU kann zunächst als spezialisierter Prozessor, dessen Aufgabe die massiven Berechnungen im Rahmen der Grafikverarbeitung ist, in Analogie zu [Tan05, S. 601, S. 610] als eine Art eigenständiger Koprozessor angesehen werden. Bei näherer Betrachtung der Arbeitsweise der GPU selbst finden sich Parallelen zum Vektorrechner. Die einzelnen Verarbeitungseinheiten auf der GPU fallen in die Kategorie SIMD, da hier ein Programm mit unterschiedlichen Daten parallel ausgeführt werden kann, zum Datenaustausch existiert ein gemeinsamer Speicher. Auf diese zunächst grobe Einordnung wird in den jeweiligen Kapiteln noch näher eingegangen.

2.2.2 Modelle für Parallelrechner

Beim Algorithmenentwurf wird von der konkreten Hardware abstrahiert, stattdessen wird als Grundlage ein theoretisches Modell des Rechners verwendet. Diese Vorgehensweise ermöglicht es, Algorithmen zu entwerfen und zu formulieren, die einfach beschreibbar sind, ohne bereits auf die Laufzeitumgebung, vorhandene Programmschnittstellen, usw. Rücksicht nehmen zu müssen. Neben der Unterstützung des Entwurfs soll diese Abstraktion auch eine einfache theoretische Analyse der Algorithmen ermöglichen, die für alle konkreten Implementierungen auf Parallelrechnern Gültigkeit besitzt. Im Vordergrund stehen dabei Aussagen zu Laufzeit, Rechenaufwand, Speicherbedarf und Anzahl benötigter Prozessoren.

Als theoretisches Modell für Parallelrechner mit gemeinsamem Speicher existiert das Modell der PRAM (Parallel Random Access Machine) [vgl. JáJ92, S. 9ff.], das auf dem Modell der RAM (Random Access Machine) aufsetzt, welche zur Analyse sequentieller Programme verwendet wird [vgl. AHU74, S. 5ff.]. Eine PRAM besteht aus einer

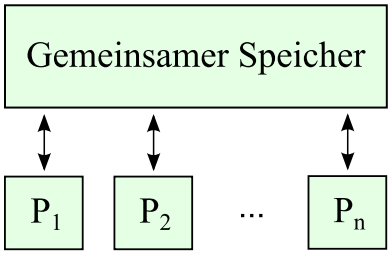

Abbildung 2.3: Modell mit gemeinsamem Speicher (shared-memory), das auch der PRAM zu Grunde liegt (basierend auf [JáJ92, Abb. 1.3]).

Reihe von Prozessoren, die einen synchronen Takt haben. Sie sind unterscheidbar durch einen eindeutigen Index, auf den im Programm zugegriffen werden kann. Jeder Prozessor besitzt einen eignen lokalen Speicher. Daneben existiert ein gemeinsamer globaler Speicher, auf den alle Prozessoren Zugriff haben und über den Daten ausgetauscht werden können, siehe Abb. 2.3. Hier wird der Untertyp der CRCW PRAM (concurrent read, concurrent write) verwendet, der einen gleichzeitigen Zugriff aller Prozessoren auf die Speicherstellen erlaubt. Lesen ist dabei tatsächlich gleichzeitig möglich, ein konkurrierender Schreibzugriff ist entweder nur erlaubt, wenn alle beteiligten Prozessoren denselben Wert schreiben wollen oder führt dazu, dass nur ein Prozessor schreiben darf (zufällig oder prioritätsgesteuert). Die Behandlung eines konkurrierenden Schreibzugriffs ist nicht nur bei der theoretischen Betrachtung nötig, um Schreibkonflikte aufzulösen, sondern muss auch bei der tatsächlichen Programmierung von Parallelrechnern berücksichtigt und mit geeigneten Mitteln gelöst werden.

Die Formulierung von Algorithmen für die PRAM erfolgt in Form von Pseudocode, wobei das Schlüsselwort `pardo` genutzt wird, um anzuzeigen, dass der folgende Quelltextblock parallel ausgeführt werden soll. Dabei wird die Notation aus [JáJ92] verwendet.

Bei der theoretischen Analyse von Algorithmen interessiert man sich für deren Komplexität in Bezug auf eine bestimmte Problemgröße; hierzu abstrahiert man von einer konkreten Eingabe und betrachtet die Komplexität als die größte Komplexität bei einer gegebenen Eingabelänge [vgl. Akl89, S. 22]. Interessant sind vor allem die Laufzeit und der Aufwand eines parallelen Algorithmus.

Die (parallele) *Laufzeit* kann dabei bestimmt werden durch die maximale Anzahl der Befehle, die einer der parallelen Prozessoren zur Abarbeitung seines Teils benötigt (d. h. vom Beginn der Berechnung bis zum Programmende). Der *Aufwand* ist die Summe aller Befehle, die zur Ausführung des gesamten Algorithmus auf allen beteiligten parallelen Prozessoren nötig sind [vgl. JáJ92, S. 31]. Hierbei sei ein uniformes Kostenmaß zur Ermittlung der Laufzeit einzelner Befehle zu Grunde gelegt, d. h. alle Befehle werden als gleich lang dauernd angesehen [vgl. AHU74, S. 12]. Da man generell an der ungünstigsten Komplexität bei einer bestimmten Eingabelänge interessiert ist (worst-case Analyse), steht eine asymptotische Betrachtung im Vordergrund, bei der konstante Faktoren unterdrückt werden.

Die Einordnung der Algorithmen erfolgt in Klassen, welche mittels der O-Notation angegeben werden (Ordnung). Dabei ist die Laufzeit T (bzw. der Aufwand W) in Abhängigkeit von der Eingabelänge n definiert als [JáJ92, S. 5]:

$$\begin{aligned} &T(n) = \mathrm{O}(f(n)) \text{ wenn es positive Konstanten } c \text{ und } n_0 \text{ gibt,} \\ &\text{so dass } T(n) \leq c \cdot f(n) \text{ für alle } n \geq n_0 \text{ gilt.} \end{aligned} \tag{2.2.1}$$

Die O-Notation wird im Folgenden zur Abschätzung der Komplexität von Algorithmen verwendet.

Neben dem beschriebenen PRAM Modell für Parallelrechner existieren weitere Modelle, speziell auch für solche mit Nachrichtenaustausch (z. B. das BSP-Modell, bulk synchronous parallel model [vgl. Val90]), die auch die Kommunikations- und Synchronisationskosten bei verteilten Systemen mit einbeziehen. Trotz der Verwendung von Multicomputersystemen basierend auf verteiltem Speicher und Nachrichtenaustausch in einem Teil der verwendeten Programme liegt der Schwerpunkt dieser Arbeit auf dem Einbezug der lokal vorhandenen GPU. Zur Analyse der Algorithmen auf der GPU wird entsprechend das Modell der PRAM verwendet, siehe auch Kap. 2.3.5.

2.2.3 Maße für Parallelität

Neben der Analyse der Algorithmen auf Basis eines Modells des Parallelrechners spielen Messgrößen bei der Beurteilung der parallelen Algorithmen und Programme eine wichtige Rolle. Mit ihnen soll z. B. gemessen werden, um wieviel schneller das parallele Programm gegenüber dem sequentiellen tatsächlich ist. Im Folgenden werden die wichtigsten Messgrößen vorgestellt. Im praktischen Teil dieser Arbeit finden sie Verwendung bei der Analyse der Programme.

Eine zentrale Messgröße ist die *Beschleunigung* (engl. speedup), die angibt, um wieviel schneller der vorliegende parallele Algorithmus mit Laufzeit t_p im Verhältnis zum besten sequentiellen Algorithmus[2] mit Laufzeit t_s ist [vgl. WA04, S. 6]:

$$S(p) = \frac{t_s}{t_p} \tag{2.2.2}$$

Die Laufzeit kann dabei in der theoretischen Untersuchung der Anzahl der Berechnungsschritte oder Befehle entsprechen (vgl. Kapitel 2.2.2) oder aber bezogen auf ein vorliegendes Programm die gemessene Laufzeit bei dessen Ausführung darstellen.

Die maximale Beschleunigung mit p Prozessoren entspricht bei Betrachtung der Gleichung zunächst p (lineare Beschleunigung). In seltenen Fällen kann z. B. auf Grund größerer Caches in Parallelrechnern eine superlineare Beschleunigung ($S(p) > p$) auftreten [vgl. WA04, S. 7]. Im Allgemeinen ist die Beschleunigung jedoch kleiner als

[2] Es sei generell angemerkt, dass bei Messungen im praktischen Teil der Arbeit als „bester sequentieller Algorithmus“ immer das jeweils gegebene sequentielle Programm angenommen wird.

die Zahl der im parallelen Fall eingesetzten Prozessoren ($S(p) \leq p$). Dies ist bedingt dadurch, dass im parallelen Algorithmus weiterer Aufwand anfällt, z. B. durch zusätzliche lokale Berechnungen, Kommunikationsaufwand zwischen den Prozessoren oder einfach der Tatsache, dass die vorhandene Arbeit nicht gleichmäßig aufgeteilt werden kann und deshalb einzelne Prozessoren teilweise unbeschäftigt sind [vgl. WA04, S.8] und [vgl. Sch00, S.42].

Zur Frage nach der maximal erreichbaren Beschleunigung eines Algorithmus existieren zwei sehr unterschiedliche Ansätze. Amdahls Gesetz (1967) beantwortet die Frage bezogen auf den parallelisierbaren Anteil des Algorithmus. Der Algorithmus besteht demnach aus einem seriellen Teil f, der nicht parallelisiert werden kann und einem Teil $(1-f)$, bei dem dies möglich ist. Ausgegangen wird hier von einer fixen Problemgröße und von keinem zusätzlichen Aufwand durch die Parallelisierung. Entsprechend ist hiernach die Beschleunigung auch definierbar als [vgl. WA04, S. 8]:

$$S(p) = \frac{t_s}{f \cdot t_s + \frac{(1-f)\cdot t_s}{p}} = \frac{p}{1+(p-1)f} \qquad \text{mit} \qquad \lim_{p\to\infty} S(p) = \frac{1}{f} \tag{2.2.3}$$

Nach Amdahls Gesetz hängt also die maximal mögliche Beschleunigung fest vom parallelisierbaren Anteil des Algorithmus ab, egal wie viele Prozessoren eingesetzt werden. Diese Betrachtungsweise der Möglichkeiten der Parallelisierung wird als zu pessimistisch angesehen [vgl. Sch00, S.43]. Nichtsdestotrotz spielt sie dennoch eine wichtige Rolle beim Algorithmenentwurf. Sie macht deutlich, dass der paralleliserbare Anteil eines Algorithmus so groß wie möglich gestaltet werden sollte, um eine hohe Beschleunigung zu erreichen.

Gustafson (1988) vertritt demgegenüber einen anderen Ansatz bei der Bestimmung der maximal erreichbaren Beschleunigung. Dieser geht davon aus, dass der serielle Teil des Problems statisch ist und somit insb. nicht größer wird, wenn man die Problemgröße erhöht. Bei diesem Ansatz wird von einer festen Zeit ausgegangen, die der Algorithmus benötigen soll; eine Erhöhung der Anzahl der eingesetzten Prozessoren führt hierbei zu der Möglichkeit, ein größeres Problem bearbeiten zu können. Die Definition der Beschleunigung und die maximale Beschleunigung ergeben sich hiernach zu [vgl. WA04, S. 12]:

$$S(p) = \frac{f \cdot t_s + (1-f)\cdot t_s}{f \cdot t_s + \frac{(1-f)\cdot t_s}{p}} = p + (1-p)\cdot f \cdot t_s \qquad \text{mit} \qquad \lim_{p\to\infty} S(p) = p \tag{2.2.4}$$

Der parallele Anteil wächst also, je mehr Prozessoren eingesetzt werden.

Beide Ansätze haben in der parallelen Entwicklung ihre Berechtigung. In der Realität wird die maximal erreichbare Beschleunigung zwischen diesen beiden Extrembetrachtungen liegen, da der nicht parallelisierbare Anteil typischerweise mit dem Einsatz einer größeren Anzahl an Prozessoren zwar geringer wird, aber wie oben angedeutet weiterer Aufwand für die Parallelverarbeitung hinzukommt. Beide Ansätze müssen demnach bei der Entwicklung paralleler Algorithmen berücksichtigt werden, zeigen sie doch deutlich die Grenzen der Parallelisierbarkeit auf.

Eine weitere Messgröße für parallele Algorithmen ist die *Effizienz*, die angibt, wie lang im Schnitt die einzelnen Prozessoren an der Berechnung beteiligt sind, d. h. ob im parallelen Fall alle Prozessoren ausgelastet sind.

$$E = \frac{t_s}{t_p \cdot p} \tag{2.2.5}$$

Bei Algorithmen für Multicomputersysteme spielt zusätzlich die Kommunikationszeit eine bedeutende Rolle. Messgrößen in diesem Zusammenhang sind die Latenz (Zeitspanne vom Senden des Pakets bis zur Antwort) und Bandbreite (welche Datenmenge das System pro Zeiteinheit übertragen kann) [vgl. Tan05, S. 676]. Speziell für die Abschätzung, wie groß die einzelnen Anteile an der Gesamtberechnung sein sollen, die einzelne Berechnungsknoten bearbeiten, ist das Verhältnis von eigentlicher Berechnungszeit t_{comp} zu Kommunikationszeit t_{comm} von Bedeutung:

$$\text{Rechenzeit zu Kommunikationszeit} = \frac{t_{comp}}{t_{comm}} \tag{2.2.6}$$

Hierbei muss eine gute Balance gefunden werden zwischen der Möglichkeit zur parallelen Berechnung einerseits und der mit zunehmender Parallelisierung steigender Kommunikation andererseits.

2.2.4 Entwurf paralleler Algorithmen

Der Schlüssel zur Parallelverarbeitung ist das Existieren von „ausnutzbarer" Parallelität [vgl. MSM05, S. 3]. D. h. ein gegebenes Problem muss nebenläufige Teilprobleme enthalten, aber auch in einer Art und Weise formulierbar sein, dass diese Teilprobleme tatsächlich parallel abgearbeitet werden können. Das Ziel ist ein paralleler Algorithmus, d. h. eine Lösungsmethode für ein gegebenes Problem, die dafür ausgelegt ist, auf einem Parallelrechner ausgeführt zu werden [vgl. Akl89, S. 3]. Aus diesem Grund ist die Aufgabe des Entwicklers bei dem Entwurf paralleler Algorithmen, die Nebenläufigkeit in dem gegebenen Problem zu entdecken und den Algorithmus entsprechend zu strukturieren. Danach folgt die Implementierung in einer passenden Programmierumgebung, bevor das Programm dann in einer parallelen Laufzeitumgebung ausgeführt werden kann [vgl. MSM05, S. 3]. Der Schlüssel steckt also im Entwurf der Algorithmen, speziell im Erkennen und Formulieren der parallelen Aspekte.

Eine standardisierte Methode zum Entwurf paralleler Algorithmen und der Umsetzung in parallele Programme mit Hilfe von Vorgehensmustern ist in [MSM05] beschrieben. Die Tätigkeiten gliedern sich dabei in verschiedene Schritte, siehe Abb. 2.4. Die Methodik soll im Folgenden kurz vorgestellt werden mit einem Fokus auf den Aspekten, die im Rahmen der Arbeit zum Einsatz kamen. Die Ausführungen folgen dabei [MSM05].

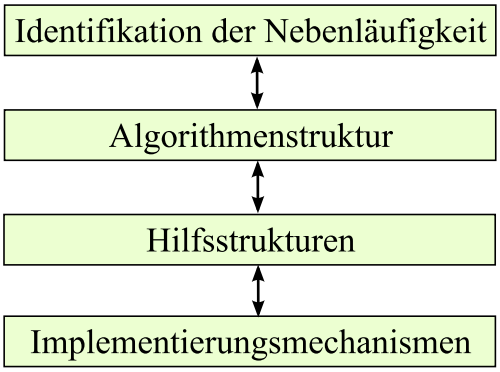

Abbildung 2.4: Vier Schritte beim Entwurf paralleler Algorithmen (basierend auf [MSM05, Abb. 1-1]).

2.2.4.1 Identifikation der Nebenläufigkeit

Bei der Identifikation der Nebenläufigkeit wird zunächst das gegebene Realweltproblem analysiert und es werden parallelisierbare Teile identifiziert. Das Problem wird so strukturiert, dass die Nebenläufigkeit sichtbar und nutzbar gemacht wird. Dies geschieht, indem das Problem zunächst in Teilprobleme zerlegt wird. Als Muster stehen hier entweder die Funktions- oder die Datenzerlegung zur Verfügung.

- Funktionszerlegung: Bei der Funktionszerlegung werden die zur Problemabarbeitung nötigen Schritte betrachtet. Mehrere zusammengehörende Schritte werden zu Funktionen zusammengefasst, die parallel zueinander ausgeführt werden können. Erst als zweites wird überlegt, mit welchen Daten die Funktionen operieren sollen. Diese Zerlegung wird gewählt, wenn die einzelnen Funktionen relativ unabhängig voneinander sind.
- Datenzerlegung: Bei der Datenzerlegung liegt der Fokus auf den Daten und wie diese in geeignete Blöcke zerlegt werden können, die dann parallel bearbeitet werden können. Erst in einem zweiten Schritt wird überlegt, welche Funktionen zur Bearbeitung der Daten nötig sind. Diese Zerlegung wird gewählt, wenn die einzelnen Datenblöcke relativ unabhängig voneinander sind.

Welche Zerlegung gewählt wird, ist nicht immer klar vorgegeben, meist finden beide Aspekte Berücksichtigung, jedoch wird in der Vorgehensweise empfohlen, einer der beiden Zerlegungen den Vorrang zu geben [vgl. MSM05, S. 29].

Da Teilchensimulationen meist aus klar abgegrenzten Einzelschritten bestehen, spielt hierbei vor allem die Funktionszerlegung eine wichtige Rolle. Mit der Überlegung, welche Daten für diese Funktionen benötigt werden, ergibt sich im zweiten Schritt daraus eine geeignete Datenzerlegung (vergleiche auch das Beispiel in [MSM05, S. 29]). Bei speziellen Konfigurationen zur Laufzeit, z. B. in einer verteilten Umgebung mit mehreren Rechenknoten, kann wiederum die Datenzerlegung Verwendung finden. Speziell bei der Datenzerlegung sollten die einzelnen Datenblöcke nicht zu klein werden, damit der Aufwand ihrer Verwaltung nicht zu groß wird [vgl. MSM05, S. 36].

Nachdem das Problem mit Hilfe der Funktions- oder Datenzerlegung in Teilprobleme zerlegt ist, müssen die Abhängigkeiten zwischen diesen Teilproblemen ermittelt werden, um zu entscheiden, welche davon später parallel bearbeitet werden können. Bei der beschriebenen Vorgehensweise werden die Teilprobleme zunächst in solche gruppiert, die gleiche Abhängigkeiten besitzen. Diese Gruppen werden dann entsprechend ihrer Abhängigkeiten in eine zeitliche Abfolge gebracht. In einem nächsten Schritt wird ermittelt, auf welche gemeinsamen Daten die Teilproblemgruppen zugreifen müssen und wie der Zugriff auf diese Daten erfolgen muss.

Eine abschließende Analyse überprüft die gefundene Aufteilung auch im Hinblick auf die gleichmäßige Verteilung der Arbeit. Außerdem wird zum ersten mal in Bezug auf die Zielplattform die Frage beantwortet, ob z. B. für die gefundene Zerlegung genügend Ausführungseinheiten zur Verfügung stehen und ob es Datenstrukturen gibt, um die gemeinsamen Daten verwalten zu können [vgl. MSM05, S. 51].

2.2.4.2 Algorithmenstruktur

Basierend auf dem vorliegenden, in parallel ausführbare Teile zerlegten Realweltproblem folgt im nächsten Schritt der Algorithmenentwurf. Da Programme einen längeren Lebenszyklus besitzen als die Rechner auf denen sie laufen, sollte der Algorithmenentwurf zunächst frei von der Zielplattform erfolgen. Als Randbedingungen in Bezug auf die aktuelle Zielplattform sollten lediglich die Größenordnung der Anzahl der Ausführungseinheiten sowie die Kosten einer Kommunikation zwischen den verschiedenen Ausführungseinheiten einbezogen werden, um einen Algorithmus zu finden, der sich leicht auf der Zielplattform umsetzen lässt [vgl. MSM05, S. 59]. Das Ziel ist ein Algorithmus, der die gefundene Problemzerlegung auf Ausführungseinheiten abbildet.

Je nach Hauptordnungskriterium werden verschiedene Algorithmenstrukturen vorgeschlagen [vgl. MSM05, S. 61]:

- Organisation der Funktionen als Hauptordnungskriterium: für eine lineare Organisation der Funktionsparallelismus (Task Parallelism), für eine rekursive das Muster Teile-und-Herrsche (Divide and Conquer)
- Organisation nach der Datenzerlegung als Hauptordnungskriterium: für eine lineare Datenzerlegung das Muster der geometrischen Zerlegung, für eine rekursive Struktur eine weitere rekursive Datenzerlegung
- Organisation nach der Ordnung der Funktionen und dem damit einhergehenden Datenfluss: bei regulärem Datenfluss das Muster der Pipeline, bei irregulärem Datenfluss eine Koordination mit Hilfe von Nachrichten

Speziell von Interesse bei der Teilchensimulation sind die Algorithmenstrukturen des Funktionsparallelismus und der geometrischen Aufteilung der Daten.

Beim Funktionsparallelismus stehen Fragen nach der Abbildung der Funktionen im Rahmen des Algorithmus, deren Zuordnung zu Ausführungseinheiten, die Berücksichtigung von Abhängigkeiten zwischen den Funktionen und damit die übergreifen-

de Ablaufplanung im Vordergrund [vgl. MSM05, S. 65]. Wichtig ist hierbei, dass es mindestens so viele Funktionen wie Ausführungseinheiten geben sollte (im Idealfall wesentlich mehr, um eine größere Flexibilität bei der Ablaufsteuerung zu erreichen). Die Berechnungen im Rahmen einer Funktion sollten zudem aufwändig genug sein, um den Aufwand zur Verwaltung der Funktionen und der Behandlung der Abhängigkeiten zu rechtfertigen.

Die geometrische Aufteilung der Daten wird immer dann eingesetzt, wenn das zentrale Organisationskriterium beinhaltet, dass zentrale Datenstrukturen in lineare Datenblöcke aufgeteilt werden, auf denen Teillösungen ermittelt werden. Für die Bearbeitung des Problems für die einzelnen Datenblöcke sind dabei typischerweise Daten von wenigen anderen Datenblöcken nötig. Dahinter steckt ein grobgranularer Datenparallelismus basierend auf den Datenblöcken [vgl. MSM05, S. 79].

2.2.4.3 Hilfsstrukturen

Erster Schritt bei der Umsetzung des entworfenen Algorithmus in ein Programm ist die Festlegung auf allgemeine Programmkonstrukte und Strukturen, die es ermöglichen, den parallelen Algorithmus auszudrücken. Dies erfolgt unabhängig von der konkreten Implementierung, dennoch sollten Strukturen gewählt werden, die in der verwendeten Entwicklungsumgebung generell unterstützt werden [vgl. MSM05, S. 119]. Im Fokus stehen hierbei die Programmstrukturen und die Datenstrukturen.

Bei den Programmstrukturen für die Teilchensimulation von Interesse sind vor allem SPMD (single program, multiple data) [vgl. MSM05, S. 128] und der Schleifenparallelismus [vgl. MSM05, S. 152]. Bei SPMD führen verschiedene Ausführungseinheiten dasselbe Programm mit verschiedenen Daten aus. Die einzelnen Knoten sind über eine eindeutige Nummer identifiziert und so kann der Programmablauf von Knoten zu Knoten variieren. Dies ist typisch für verschiedene MPI-Knoten in Systemen mit verteiltem Speicher. Beim Schleifenparallelismus wird nicht das gesamte Programm parallel abgearbeitet, sondern nur einzelne Schleifen, dabei werden voneinander unabhängige Schleifendurchläufe parallel abgearbeitet. Dieses Muster kommt typischerweise zum Einsatz, wenn nur Teile eines bestehenden Programms parallelisiert werden sollen und ermöglicht eine einfache Art der schrittweisen Parallelisierung. (Andere vorgeschlagene Programmstrukturen spielen im Rahmen dieser Arbeit keine besondere Rolle).

Bei den Datenstrukturen spielen bei der Teilchensimulation vor allem die gemeinsam genutzten Daten [vgl. MSM05, S. 174] und speziell die verteilten Arrays [vgl. MSM05, S. 199] eine Rolle. Neben der Einhaltung einer Zugriffsreihenfolge und Synchronisierungsmechanismen bei gleichzeitigem Zugriff auf die Daten muss hierbei speziell darauf geachtet werden, dass die Daten passend zum Ablauf der Berechnung verteilt werden und insb. dort zur Verfügung stehen, wo sie als nächstes für eine Berechnung benötigt werden. (Andere vorgeschlagene Datenstrukturen z. B. gemeinsam genutze Warteschlangen spielen im Rahmen dieser Arbeit keine Rolle).

2.2.4.4 Implementierungsmechanismen

Im letzten Schritt liegt der Fokus auf im Rahmen der Parallelen Programmierung typsichen Implementierungsmechanismen, für die es in den meisten parallelen Entwicklungsumgebungen bereits eine entsprechende Unterstützung gibt. Spezielles Augenmerk wird hierbei auf die Verwaltung der parallel auszuführenden Teilaufgaben (z. B. mittels Prozessen und Threads), auf die Synchronisation (Erzwingen der benötigten Abarbeitungsreihenfolge) und die Kommunikation gelegt [vgl. MSM05, S. 216]. Diese Aspekte müssen vor dem Hintergrund der zur Verfügung stehenden Entwicklungs- und Laufzeitumgebung bei der Realisierung des Programms beachtet werden.

Obwohl im Rahmen dieser Arbeit auf bestehenden Programmen aufgesetzt wurde und so die grundlegenden Algorithmen bereits vorgegeben waren, wurde die vorgestellte Vorgehensweise zum Entwurf paralleler Algorithmen dennoch bei den Aspekten berücksichtigt, die zu parallelisieren waren. Im praktischen Teil wird jeweils die Art der gewählten Strukturierung angesprochen.

2.2.5 Parallele Softwareentwicklung

Zu jedem der in Kap. 2.2.1 vorgestellten parallelen Architekturen gibt es in der Softwareentwicklung eine entsprechende Unterstützung. Da im Rahmen dieser Arbeit die Einbeziehung der GPU im Vordergrund stand, auf welche in Kap. 2.3 näher eingegangen wird, soll an dieser Stelle kurz die Programmierung von Multicomputersystemen mittels Nachrichten-basierter Kommunikation (message-passing) dargestellt werden.

2.2.5.1 Programmierung von Multicomputersystemen mittels MPI

Das Modell der Nachrichten-basierten Kommunikation besteht aus mehreren Prozessen, die jeweils einen lokalen Speicher mit eigenem Adressraum besitzen und die untereinander Nachrichten austauschen können [vgl. GLS99, S. 14]. Ein Datentransfer zwischen den Prozessen bedingt immer Operationen zum Nachrichtenaustausch, die von allen an der Kommunikation beteiligten Prozessen ausgeführt werden müssen [vgl. GLS99, S. 5]. Das zu Grunde liegende Kommunikationsnetzwerk und dessen Aufbau ist nicht Teil des Modells, eine genauere Kenntnis darüber muss aber ggf. beim Algorithmenentwurf einbezogen werden.

MPI (message passing interface) ist eine spezielle Realisierung des Message Passing Modells in Form der Spezifikation einer Bibliothek [vgl. GLS99, S. 13]. Die erste Version des Standards wurde 1994 vom MPI Forum herausgegeben, die aktuelle Version des Standards ist die Version 3.0 vom September 2012 [Mes12]. Durch die Standardisierung von Funktionsnamen, Aufrufsequenzen und Rückgabewerten soll gewährleistet werden, dass die Funktionalitäten der unterschiedlichen MPI-Implementierungen über Programmiersprachen hinweg gleich heissen und sich gleich verhalten. Ein wichtiges Anliegen des Standards ist die Portabilität der Programme.

Speziell für den Nachrichtenaustausch existieren verschiedene Möglichkeiten, so kann dieser blockierend oder nicht-blockierend und gepuffert oder ungepuffert stattfinden. Neben einem einfachen Nachrichtenaustausch über `MPI_Send` und `MPI_Recv` gibt es eine Reihe kollektiver Kommunikationsmöglichkeiten, zum einen für den Datenaustausch (z. B. mittels `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Alltoall`) und zum anderen für kollektive Berechnungen (mittels `MPI_Reduce` mit den Möglichkeiten für Maximum, Minimum, Summe, Produkt, sowie logischen oder bitweisen Verknüpfungen) [vgl. GLS99, S. 18f.]. Als wichtige Möglichkeit zur Synchronisation der einzelnen Prozesse mittels einer Barriere steht der Befehl `MPI_Barrier` zur Verfügung. Für eine vollständige Liste sei auf den Standard verwiesen [Mes12].

Für den MPI-Standard sind eine Reihe sowohl kommerzieller als auch freier Implementierungen verfügbar. Im Rahmen des bestehenden Programms PATRIC wird die freie Implementierung MPICH eingesetzt, welche vom Argonne National Laboratory, verschiedenen Universitäten und weiteren Partnern entwickelt wird. Diese Implementierung existiert für eine Reihe von Architekturen und Betriebssystemen, im Rahmen der Simulationsrechnungen an der GSI kommt sie auf einem Linux-Cluster zum Einsatz. Weiterführende Informationen zu MPICH finden sich in [MPI12].

2.2.5.2 Master-Slave-Modell

Im Rahmen der Parallelisierung einer Datenverarbeitung wird beim EVA-Prinzip (Eingabe – Verarbeitung – Ausgabe) typischerweise nur der Verarbeitungsschritt mit Hilfe vieler Ausführungseinheiten parallelisiert, wie in Abbildung 2.5 dargestellt [vgl. WA04, S. 80]. Voraussetzung für eine Datenparallelisierung im Allgemeinen ist, dass die Daten unabhängig voneinander verarbeitet werden können. Dies bezieht sich also ebenfalls auf den Verarbeitungsschritt. In der Praxis lässt sich meist die Ein- und Ausgabe nicht parallelisieren, d. h. es besteht der Bedarf, die Daten am Anfang initial aus einer Datenquelle zu lesen und auf die parallelen Ausführungseinheiten zu verteilen und zwischendurch oder am Ende der Berechnung die Ergebnisse wieder zentral zu sammeln und auszugeben.

Um dies zu unterstützen, ist ein gängiges Organisationsprinzip das sog. Master-Slave-Modell [vgl. WA04, S. 79f.]. Hierbei übernimmt ein Prozess (der sog. Master) die zentrale Aufgabe, den anderen Prozessen (den sog. Slaves) Berechnungsaufgaben zuzuteilen und den Ablauf zu steuern. Dieses Prinzip wird insbesondere dann angewendet, wenn die zu verteilenden Aufgaben voneinander unabhängig sind und wenn die Dauer der einzelnen Berechnungen schwer vorherzusagen ist [vgl. GLS99, S. 35]. Ersteres ist gerade bei der Datenparallelisierung gegeben, letzteres ist allein dadurch begründet, dass in der Praxis typischerweise die einzelnen Ausführungseinheiten je nach eingesetzter Hardware und aktueller Auslastung ein unterschiedliches Zeitverhalten zeigen. Eine zentrale Fragestellung beim Master-Slave-Modell ist, wie eine Ausbalancierung (Load-Balancing) zwischen den Slaves erreicht werden kann [vgl. WA04, S. 80].

Das Master-Slave-Modell kann im Rahmen der verteilten Berechnung mittels Nachrichten-basierter Kommunikation realisiert werden. Das Verteilen von Einga-

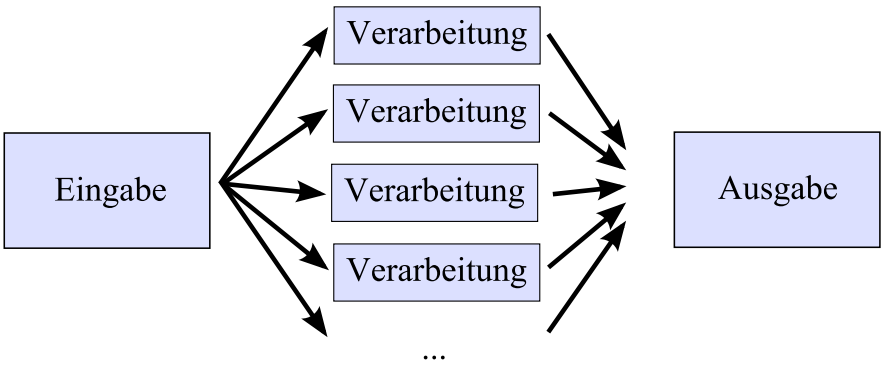

Abbildung 2.5: EVA-Prinzip vor dem Hintergrund der Parallelverarbeitung.

bedaten und das Einsammeln der Ergebnisse bei Verwendung von MPI kann dabei mit Hilfe der einfachen Funktionen `MPI_Send` und `MPI_Recv` zum Versenden und Empfangen der Daten oder mit den speziell für diesen Anwendugngsfall angepassten Funktionen `MPI_Scatter` (verteilen an mehrere) und `MPI_Gather` (Einsammeln der Ergebnisse) bzw. `MPI_Reduce` (Verarbeiten beim Einsammeln, z. B. Summenbildung) umgesetzt werden. Beispiele für die Realisierung des Master-Slave-Prinzips mit MPI gibt [GLS99]. Zum Verteilen der Berechnungsaufgaben findet das Master-Slave-Modell mittels MPI bereits Anwendung im bestehenden Simulationsprogramm PATRIC. Hierbei werden die einzelnen Teilstücke eines Teilchenpakets auf die unterschiedlichen MPI-Knoten verteilt, auf denen dann eine lokale Verarbeitung stattfindet. Auf diesen Punkt wird im Rahmen der Beschreibung der Simulationsprogramme näher eingegangen.

Das Master-Slave-Prinzip findet aber auch bei der Grafikkartenprogrammierung Anwendung: der Host kann als Master angesehen werden, der die Daten bereitstellt und die Berechnung steuert. Die parallelen Ausführungseinheiten der Grafikkarte können als Slaves angesehen werden, die die Berechnungen durchführen. Dass die Slaves hierbei voneinander unabhängig arbeiten können, ist jedoch nicht immer gegeben. So kann eine Berechnung auf der Grafikkarte es erfordern, dass sich alle Ausführungseinheiten Synchronisieren, oder sogar zur Ermittlung eines Gesamtergebnisses miteinander interagieren müssen.

2.3 Parallelität mit Hilfe von Grafikprozessoren

Da Grafikprozessoren dafür ausgelegt sind, eine Aufgabe (z. B. die Berechnung von Pixelfarben) massiv-parallel mit vielen verschiedenen Eingangsdaten abzuarbeiten (sog. Datenparallelismus), sind sie für die Parallelverarbeitung von großem Interesse. Sie versprechen die Möglichkeit paralleler Abarbeitung, ohne weitere Prozessoren oder gar weitere vernetzte Rechner zu benötigen. Entsprechend werden Grafikprozessoren im Bereich der Supercomputer immer häufiger eingesetzt. So sind nach der aktuellen Veröffentlichung der Top 500 Supercomputersysteme bereits 62 davon mit grafischen Koprozessoren ausgestattet, Tendenz steigend [vgl. TOP12].

Durch die mittlerweile hohe Verbreitung in modernen Arbeitsplatzrechnern und der Verfügbarkeit von allgemeinen Programmierschnittstellen und kostenlosen Entwicklungswerkzeugen können Grafikprozessoren auch ausserhalb von Supercomputern von jedermann für massiv-parallele Berechnungen eingesetzt werden. Der geringe Preis und die stetig steigende Leistung machen Parallelverarbeitung überall möglich.

Im Folgenden soll zunächst kurz die Historie von Grafikprozessoren angerissen werden. Danach folgt zum einen die Beschreibung der logischen Konzepte, die bei der Programmierung der GPU eine Rolle spielen. Zum anderen wird der Aufbau einer GPU kurz dargestellt und beschrieben, wie bei der Ausführung eines Programms die logischen Konzepte auf die tatsächlichen Ausführungseinheiten abgebildet werden. Ein Überblick zur Programmierung von Grafikprozessoren und zur theoretischen Einordnung von GPU-Programmen bildet den Abschluss dieses Kapitels.

2.3.1 Historie

Grafikprozessoren gibt es seit Anfang der 1980er Jahre, damals waren sog. Video Display Controller für die Anzeige der Ausgabedaten am Bildschirm zuständig. Das Aufkommen grafischer Oberflächen für Betriebssysteme führte Anfang der 1990er Jahre zu einem Bedarf an 2D-Grafikbeschleunigern. Anfangs hatten diese Grafikbeschleuniger fest eingebaute Algorithmen, die durch den Programmierer nur konfiguriert werden konnten. Daneben kamen 3D-Grafikbeschleuniger auf den Markt, die ihren Weg sowohl in den kommerziellen als auch Mitte der 1990er Jahre in den privaten Bereich fanden. Die erste Schnittstelle für eine 3D-Programmierung ermöglichte 1992 die OpenGL-Bibliothek [vgl. SK10, S. 4ff.].

Grafikkarten boten im Folgenden mehr und mehr eigene Möglichkeiten für Berechnungen. 1999 verwendete NVIDIA erstmals den Begriff der „GPU“, um die Eigenständigkeit des Grafikprozessors hervorzuheben. 2002 kam der Begriff der „GPGPU“ (General Purpose Graphics Processing Unit) auf: im Rahmen von Forschungsprojekten wurde die Rechenkapazität der GPU ausgenutzt; allerdings waren die Möglichkeiten beschränkt durch die vorgegebenen Schnittstellen, die zunächst allein für die Grafikverarbeitung ausgelegt waren [vgl. KH10, S. 7].

Getrieben durch Universitäten und Hersteller kamen Programmierschnittstellen und Entwicklungswerkzeuge auf den Markt, die die Grafikkarten für allgemeine Berechnungen zugänglich machten. 2006 veröffentlichte NVIDIA mit der „GeForce 8800 GTX“-Grafikkarte einen allgemein programmierbaren Grafikprozessor. Dieser unterstützte erstmals Gleitkommaberechnungen nach dem IEEE-Standard, Lese- und Schreibzugriffe auf beliebige Speicherstellen und die Programmiermöglichkeit von der Hochsprache C/C++ aus [vgl. SK10, S. 7].

Neben verschiedenen anderen Entwicklungen dominieren Stand heute (September 2013) am Markt der von der Khronos Group vertretende offene Standard OpenCL (Open Computing Language) und das herstellerspezifische CUDA (Compute Unified Device Architecture) von NVIDIA. Im Umfeld der Teilchensimulationen an der GSI wird CUDA präferiert; Grund hierfür ist vor allem die breite Unterstützung durch frei verfügbare Bibliotheken (wie cuFFT, cuBLAS etc.). Da im Rahmen dieser Diplomarbeit eine NVIDIA Grafikkarte und entsprechend CUDA zur Programmierung zum Einsatz kam, basieren die Ausführungen in den folgenden Kapiteln auf der CUDA Architektur.

2.3.2 Programmierkonzepte

Soll ein Programm den Grafikprozessor einbeziehen, können die im Rahmen des Algorithmenentwurfs identifizierten parallelisierbaren Anteile in Form von parallelen Prozeduren auf die Grafikkarte gebracht werden. Die Ausführung des Gesamtprogramms steuert der Prozessor des Rechners. Er übernimmt damit die Aufgabe des Masters im Master-Slave-Modell. Der Grafikprozessor fungiert in diesem Zusammenspiel als Koprozessor mit eigenem Speicher, er kann seine Teile des Programms dabei mit vielen hundert parallelen Ausführungseinheiten abarbeiten.

Um die Möglichkeiten des Grafikprozessors in einfacher Weise nutzen zu können, wird dieser dem Entwickler gegenüber in Form einer Abstraktionsschicht präsentiert. Bei NVIDIA übernimmt diese Aufgabe die sog. CUDA Architektur, eine Plattform für parallele Berechnungen und gleichzeitig ein Programmiermodell. Im CUDA-Sprachgebrauch[3] wird der eigentliche Rechner mit Prozessor und Hauptspeicher als *Host* bezeichnet, die Grafikkarte als *Device*.

Für den Grafikprozessor werden sog. *Kernel* programmiert. Innerhalb eines solchen Kernels, einer Prozedur, beschreibt der Entwickler die Aufgabe einer Ausführungseinheit. Die einzelnen Ausführungseinheiten nehmen die Aufgabe der Slaves im Master-Slave-Modell wahr. Ein Kernel wird später mit hunderten Ausführungseinheiten in Form vieler paralleler *Threads* (Ausführungsfäden) abgearbeitet. Threads sind dabei die kleinsten parallelen Einheiten. Jeder Thread hat einen eindeutigen Index, der genutzt werden kann, um auf die zu verarbeitenden Daten zuzugreifen. Durch diesen Mechanismus kommt Datenparallelität zustande. Die Threads selbst werden in *Blöcken* zusammengefasst. Alle Threads innnerhalb eines Blocks werden gemeinsam ausgeführt, haben Zugriff auf einen gemeinsamen Speicher und können im Programm

[3] Im Folgenden werden zum besseren Verständnis die (meist englischen) CUDA-spezifischen Begriffe verwendet, es sei denn, es existieren gebräuchliche deutsche Begriffe.

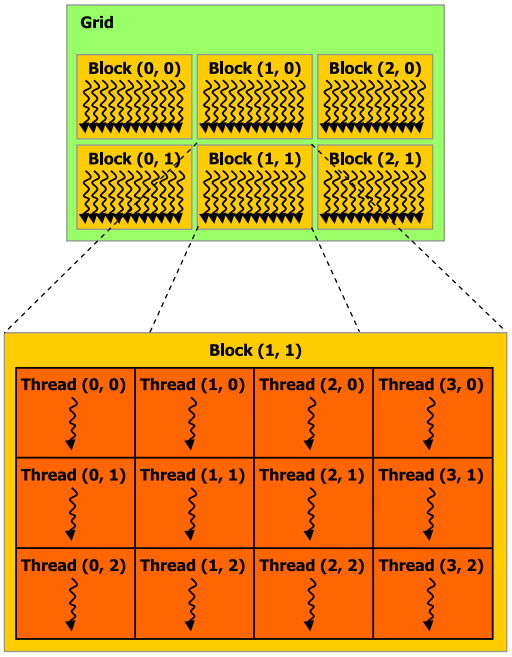

Abbildung 2.6: Das CUDA Ausführungsmodell (Quelle: [NVI13b, Abb. 6]). Dargestellt sind die einzelnen Ausführungsfäden (Threads), die logisch in Form von Blöcken in einem Gitter strukturiert sind.

synchronisiert werden. Verschiedene Blöcke müssen voneinander unabhängig sein. Blöcke wiederum werden in Form eines *Gitters* (grids) organisiert, siehe auch Abb. 2.6.

Die interne Dimensionierung der Blöcke und des Gitters stellt eine rein logische Organisation der parallel ausgeführten Threads dar und hat keinen Einfluss auf die Ausführung. Sie sollte entsprechend des Anwendungsfalls zur programmtechnischen Strukturierung gewählt werden. Um eine Vorstellung von der möglichen Dimensionierung zu geben, sei erwähnt, dass bei der im Rahmen der Arbeit verwendeten Grafikkarte die Blöcke und das Gitter 3-dimensional definiert werden können. Die Dimensionierung eines Blocks darf maximal 1024 x 1024 x 64 Threads betragen, wobei ein Block insgesamt maximal 1024 Threads beinhalten darf. Die Dimensionierung des Gitters darf maximal 65535 x 65535 x 65535 Blöcke betragen.

2.3.3 Aufbau einer Grafikkarte

Neben dem beschriebenen logischen Programmiermodell ist es wichtig, die darunterliegende Hardware der Grafikkarte zu betrachten, um die tatsächlichen Möglichkeiten der Parallelisierung zu verstehen. Ausgehend vom Aufbau einer Grafikkarte soll die im Rahmen einer Programmausführung nötige Zuordnung der logischen Strukturen zu Ausführungseinheiten dargestellt werden. Anschließend wird speziell auf die Speicherhierarchie eingegangen, da die Wahl des genutzten Speichers einen wichtigen Einfluss auf die Laufzeit der Programme hat.

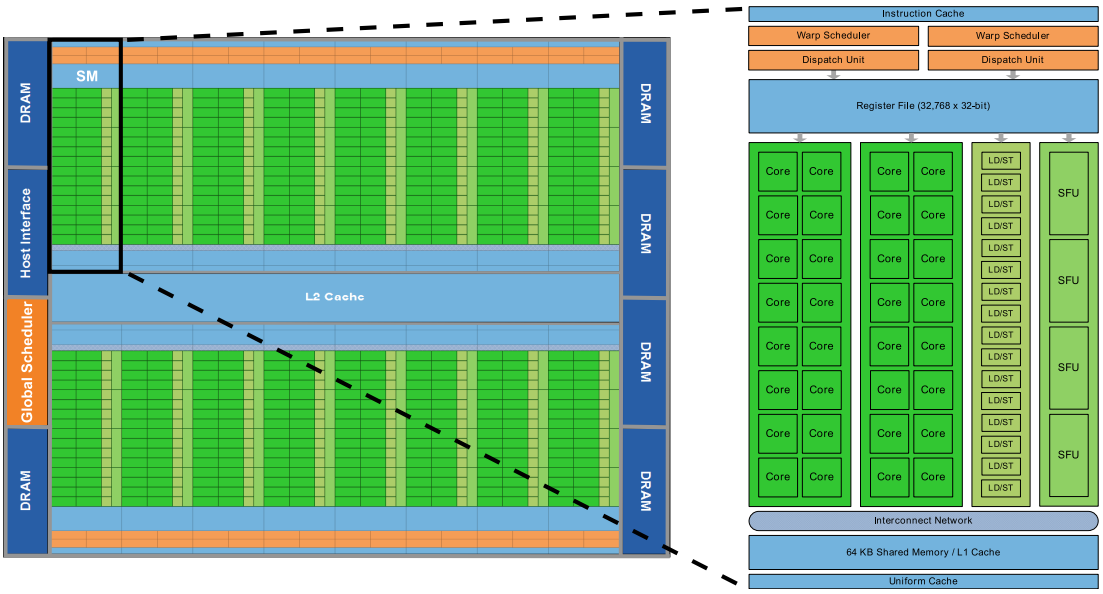

Abbildung 2.7: Aufbau einer Tesla C2075 Grafikkarte mit insg. 14 Multiprozessoren (Streaming Multiprocessors) mit jeweils 32 Ausführungseinheiten (Cores), d. h. insgesamt 448 parallelen Ausführungseinheiten (basierend auf [NVI09a, S. 7f.]).

2.3.3.1 Ausführungseinheiten einer Grafikkarte

Intern besteht eine Grafikkarte aus vielen parallelen Einheiten, den sog. *Multiprozessoren* (Streaming Multiprocessors). Die Anzahl der Multiprozessoren variiert zwischen den verschiedenen Grafikkartentypen. Allen Multiprozessoren steht ein globaler Speicher, ein dem globalen Speicher vorgeschalteter gemeinsamer schneller Zwischenspeicher (Level-2-Cache) sowie ein dem jeweils einzelnen Multiprozessor zugeordneter schneller Speicher (Level-1-Cache) zur Verfügung. Der globale Speicher ist nach außen hin sichtbar, insb. wird er dazu genutzt, um während eines Programmablaufs Daten zwischen dem Hauptspeicher des Rechners und der Grafikkarte auszutauschen.

Jeder Multiprozessor selbst beinhaltet eine Reihe paralleler *Ausführungseinheiten* (Cores). Diese haben einen gemeinsamen Befehlsspeicher und führen dasselbe Programm im gleichen Takt mit unterschiedlichen Daten aus. Jede Ausführungseinheit besitzt dazu ihre eigene Integer- und Gleitkommaeinheit. Die Ausführungseinheiten eines Multiprozessors besitzen mit dem *gemeinsamen Speicher* einen Speicherbereich, über den sie untereinander Daten austauschen können. Außerdem teilen sich die Ausführungseinheiten eine gemeinsame Menge an Registern. Abb. 2.7 zeigt den schematischen Aufbau der im Rahmen der Arbeit genutzten Grafikkarte.

2.3.3.2 Ausführung von Programmen

Bei der Ausführung eines Programms auf dem Grafikprozessor werden die in Kapitel 2.3.2 beschriebenen logischen Elemente des Programms (Blöcke und Threads) den

zur Verfügung stehenden Komponenten des Grafikprozessors zugeteilt. Ein globales Steuerwerk (Global Scheduler) weist dabei einem Multiprozessor ein oder mehrere Blöcke des Programms zu. Da die einzelnen Blöcke wie oben beschrieben unabhängig voneinander sind (insb. ihre Ausführungsreihenfolge nicht bekannt ist), kann hiermit eine Skalierbarkeit über verschiedene Grafikkartentypen hinweg erreicht werden: auf einer moderneren Grafikkarte mit einer größeren Anzahl an Multiprozessoren entfallen auf den einzelnen Multiprozessor einfach weniger Blöcke des Programms, sodass das Programm insgesamt schneller abläuft [vgl. Far11, S. 86]. Das Programm skaliert somit über die Anzahl der vorhandenen Multiprozessoren und muss dafür nicht umgeschrieben werden.

Bei der Zuweisung von Blöcken zu Multiprozessoren beachtet das globale Steuerwerk dabei durch die Grafikkarte vorgegebene Einschränkungen; so können z. B. bei der genutzten Grafikkarte einem Multiprozessor maximal 8 Blöcke oder maximal 1536 Threads auf einmal zugeteilt werden (hierbei ist für eine konkrete Ausfürhungskonfiguration der jeweils kleinere Wert ausschlaggebend). Außerdem kann die Anzahl der durch die Threads benötigten Register zu einer weiteren Reduktion der zugeteilten Blöcke führen. Die genannten Werte müssen bei der Festlegung der Dimensionen der Ausführungskonfiguration beachtet werden, um eine möglichst gute Auslastung der einzelnen Multiprozessoren zu erzielen [vgl. KH10, S. 84].

Der einem Multiprozessor zugewiesene Block wird bei der verwendeten Grafikkarte in Untermengen von jeweils 32 Threads, sog. Warps, aufgeteilt. Ein Warp enthält dabei jeweils immer Threads mit aufeinanderfolgenden Thread-Nummern. Der Multiprozessor arbeitet diese Threads mit Hilfe der einzelnen Ausführungseinheiten parallel ab. Aktuelle Grafikkarten bieten die Möglichkeit, dass mehrere Warps gleichzeitig aktiv sind, der Multiprozessor schaltet dabei zwischen diesen um, wenn einzelne Warps auf Ergebnisse warten müssen [vgl. KH10, S. 88]. Dass die Abarbeitung auf der niedrigsten Ebene in Untermengen von 32 Threads erfolgt, bedeutet umgekehrt, dass eine eingeplante Menge von Threads pro Block, welche nicht ein Vielfaches von 32 darstellt, entsprechend mit zusätzlichen leeren Threads aufgefüllt wird. Dies ist bei der Definition der Ausführungskonfiguration zu beachten [vgl. KH10, S. 125].

2.3.3.3 Speicherhierarchie

Bei der Programmierung für den Grafikprozessor stehen eine Reihe von unterschiedlichen Speichern zur Verfügung. Da diese sehr unterschiedliche Zugriffszeiten und -bandbreiten aufweisen, ist die Wahl des zu verwendenden Speichers eine wichtige Entscheidung im Rahmen der Entwicklung. Die Speicherhierarchie (siehe Abb. 2.8) soll im Folgenden kurz vorgestellt werden.

Im unteren Teil der Speicherhierarchie in Abb. 2.8 befindet sich der externe Speicher, auf den von allen Threads und insb. auch vom Host aus zugegriffen werden kann. Extern bedeutet, dass der Speicher ausserhalb des Grafikprozessor-Chips angesiedelt ist. Der mit GDDR5 SDRAM[4] Technologie realisierte Speicher dient dem Datenaustausch mit

[4]GDDR5 SDRAM: graphics double data rate synchronous dynamic random access memory, version 5

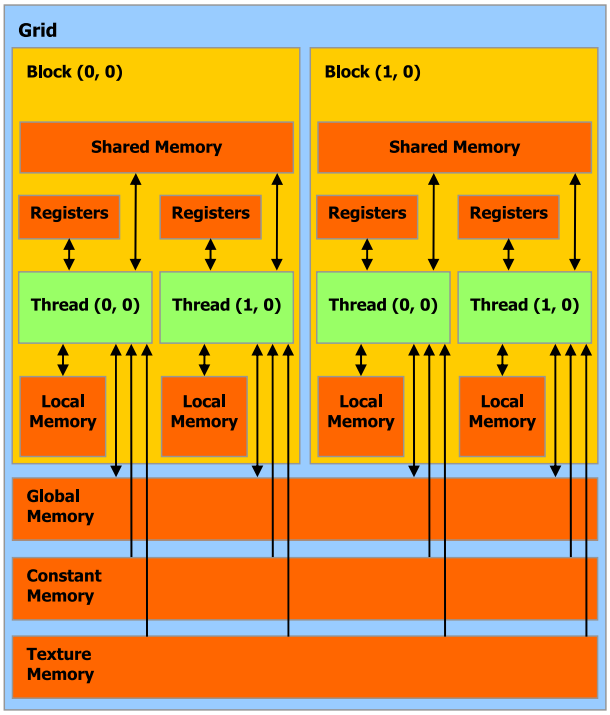

Abbildung 2.8: CUDA Speicherhierarchie (Quelle: [NVI07, Abb. 2-2]).

dem Hostsystem und ist bei der eingesetzten Grafikkarte 5,25 GB groß. Einer dieser externen Speicher ist der *globale Speicher*. Der globale Speicher ist der langsamste Speicher auf der Grafikkarte, auf ihn kann von allen Threads innerhalb des Programms lesend- und schreibend zugegriffen werden. Ebenfalls zu den externen Speichern zählt der *konstante Speicher*. Auch auf ihn kann vom Hostsystem aus lesend und schreibend zugegriffen werden, alle Threads des Programms haben jedoch ausschließlich lesenden Zugriff (analog verhält sich der Texturen-Speicher speziell für Grafikanwendungen, auf den jedoch hier nicht näher eingegangen werden soll). Dadurch, dass nur lesender Zugriff möglich ist, ist der Zugriff auf diesen Speicher zwischengepuffert und sofern alle beteiligten Threads dieselben Daten lesen, ist der Zugriff im Vergleich zum globalen Speicher schneller möglich.

Auf dem Grafikprozessor-Chip befindet sich der *gemeinsame Speicher* (shared memory). Dieser ist pro Block verfügbar, d. h. Threads eines Blocks können hierüber Daten austauschen. Da auf ihn sehr schnell zugegriffen werden kann, wird empfohlen, ggf. Daten aus dem globalen Speicher einmal in den gemeinsamen Speicher einzulesen und dann in der eigenen Berechnung nur auf diese Daten im gemeinsamen Speicher zuzugreifen [vgl. NVI13a].

Ebenfalls auf dem Grafikprozessor-Chip befinden sich die *Register*. Sie bieten einen lokalen Speicher pro Thread, auf den lesend und schreibend mit sehr hoher Bandbreite (ungefähr zwei Größenordnungen mehr als beim Zugriff auf den lokalen Speicher, [vgl. KH10, S. 98]) und geringer Latenz zugegriffen werden kann.

Darüber hinaus gibt es noch den *lokalen Speicher* pro Thread. Dieser existiert nicht physikalisch, sondern ist eine logische Bezeichnung für alle Variablen, die nicht explizit einer der zuvor genannten Speicherorte zugeordnet sind. Da für diese Variablen der Speicherort zum Kompilierzeitpunkt festgelegt wird, und dieser sehr stark variieren kann – von Registern (bei einzelnen skalaren Werten und sofern noch genug Register zur Verfügung stehen) bis hin zum globalen Speicher (für Arrays, bei denen die Zugriffe zum Kompilierzeitpunkt noch nicht alle ermittelt werden können) – sollte diese Art von Speicher bei der Entwicklung nicht verwendet werden, um nicht unnötig Performanzeinbußen zu erzeugen. Es sollte statt dessen immer explizit der Speicherort einer Variable angegeben werden.

2.3.4 Softwareentwicklung

Im Folgenden wird kurz dargestellt, wie ein Grafikprozessor programmiert wird. Dazu wird kurz auf CUDA C eingegangen. Als wichtige Technik bei der Programmierung von Grafikprozessoren wird dabei beleuchtet, wie die Synchronisation verschiedener Threads während der Ausführung erreicht werden kann. Es folgt eine Beschreibung von Streams, mit denen verschiedene Prozeduren teilweise überlappend abgearbeitet werden können. Den Abschluss des Kapitels bildet eine Beschreibung von Thrust, einer Bibliothek für einen abstrakteren, einfacheren Zugriff auf die GPU.

2.3.4.1 CUDA C

CUDA unterstützt eine Reihe von Programmiersprachen[5]. Da die bestehenden Simulationsprogramme, auf denen diese Arbeit aufgesetzt, in C/C++ geschrieben sind, kam CUDA C zum Einsatz. Bei CUDA C handelt es sich um C mit einer Reihe von Spracherweiterungen, die es ermöglichen, Kernel für die GPU zu schreiben, die entsprechenden Aufrufe zu tätigen, Speicher auf der Grafikkarte zu allokieren und Daten in den Speicher der Grafikkarte und zurück zu kopieren. Eine kurze Übersicht der CUDA C Spracherweiterungen findet sich im Anhang A.1.

Der Quelltext wird mit dem `nvcc` Compiler (NVIDIA C Compiler) übersetzt (Aufruf z. B. über `nvcc cudacode.cu`); dieser erkennt die Spracherweiterungen und trennt den Quelltext auf in Anteile für den Prozessor und solche für den Grafikprozessor. Der Quelltext für den Prozessor wird an einen vorhandenen Compiler (unter Linux z. B. `gcc`) weitergegeben. Der Quelltext für den Grafikprozessor wird zunächst in PTX Assembler (Parallel Thread Execution) transformiert, und dann zur Laufzeit durch einen Just-in-Time-Compiler passend zum jeweils installierten CUDA Gerätetreiber in ein Binärformat umgewandelt und ausgeführt [vgl. KH10, S. 44]. Im Hostcode werden die Kernelaufrufe (sog. Stubs bzw. Stellvertreterobjekte für die eigentliche Funktionalität auf der Grafikkarte [vgl. KH10, S. 53]) ersetzt durch entsprechende CUDA C Laufzeitaufrufe, um jeden Kernel entsprechend zu laden und auszuführen.

[5]CUDA 5.0 bietet Unterstützung für C, C++, Fortran, Java, Python, DirectCompute und Directives (OpenACC) [vgl. NVI13b]

Indem man CUDA C verwendet, nutzt man die sog. Laufzeitschnittstelle (Runtime API), die den vollen Funktionsumfang der GPU für allgemeine Berechnungen zur Verfügung stellt. (Darunter liegt nur noch die Treiberschnittstelle, die hardwarenäher ist und für Spezialanwendungen genutzt wird.) Die Laufzeitschnittstelle wird in der einschlägigen Literatur als einfach zu lesende und saubere Schnittstelle bewertet [vgl. Far11, S. 6], und wird deshalb als gute Wahl für die Programmierung der GPU empfohlen, da sie eine noch recht GPU-nahe Programmierung erlaubt, jedoch dem Anwender das oben beschriebene Programmiermodell mit Threads, Blöcken und Gittern präsentiert und darüber hinaus keine Internas der Grafikkarte bekannt sein müssen. Obwohl der so entstehende Quelltext für die GPU tatsächlich gut lesbar ist, enthält das Gesamtprogramm allein schon durch die Aufteilung in Teile für die CPU und Teile für die GPU zusätzlichen technischen Quelltext, der die Les- und Wartbarkeit negativ beeinflussen kann. Um auf einer noch höheren Ebene möglichst auch von dem GPU-Programmiermodell zu abstrahieren, existieren Bibliotheken wie die Thrust API, welche die GPU noch stärker vor dem Entwickler verbergen, siehe auch Kap. 2.3.4.4.

2.3.4.2 Synchronisierungsmechanismen

Im einfachsten Fall arbeiten tatsächlich alle Threads auf der Grafikkarte unabhängig voneinander. Häufig ist es bei realen Problemen jedoch so, dass Threads miteinander kooperieren müssen, z. B. weil sie auf Zwischenergebnissen anderer Threads aufsetzen. Hierbei spielt die Synchronisation der Threads untereinander eine wichtige Rolle. In CUDA C erreicht man die Synchronisierung aller Threads eines Blocks mit Hilfe der Anweisung `syncthreads` [vgl. SK10, S. 78]. Hierbei ist sichergestellt, dass die Threads an dieser Stelle anhalten, bis alle Threads des Blocks diesen Befehl erreicht haben. Erst dann wird die Ausführung fortgeführt. Dieser auch als Barriere bekannte Mechanismus [vgl. KH10, S. 81] separiert die Abarbeitung des Kernels in einzelne, voneinander getrennte Phasen.

Möchte man erreichen, dass Threads auch blockübergreifend an einer Synchronisation teilnehmen, ist dafür zunächst kein Mechanismus vorgesehen, da dies der Skalierbarkeit zuwider sprechen würde, vgl. 2.3.3.2. Um dennoch die Möglichkeit zu haben, alle Threads an einer globalen Synchronisation teilhaben zu lassen, muss der aktuelle Kernel beendet und für die nächste Berechnungsphase ein neuer Kernel gestartet werden. Hierdurch kann eine globale Barriere implementiert werden [vgl. KH10, S. 91].

Da Kernelaufrufe asynchron sind, wird zur Synchonisation zwischen Host und Device der Befehl `cudaThreadSynchronize` eingesetzt. Dieser realisiert eine Barriere und blockiert den Hostprozess, bis alle eingeplanten Prozesse auf der GPU abgearbeitet sind. Erst danach kann der Hostprozess weiterlaufen und z. B. auf Endergebnisse aus den Berechnungen der GPU zugreifen. Dieselbe Art von Synchronisation zwischen Host und Device wird erreicht, wenn ein blockierender Datentransfer per `cudaMemcpy` durchgeführt wird, da hier intern `cudaThreadSynchronize` aufgerufen wird [vgl. Far11, S. 10].

Eine Möglichkeit zur Synchronisation auf Befehlsebene bei einem Read-Modify-Write-Befehl mit mehreren beteiligten Threads stellen die atomaren Operationen (atomics) dar [vgl. SK10, S. 168ff.]. In neueren CUDA Versionen werden atomare Operationen sowohl auf dem globalen Speicher als auch auf dem gemeinsamen Speicher pro Block unterstützt. Per `atomicAdd` ist es möglich, einen Wert atomar zu verändern, ohne dass andere Threads intervenieren können (z. B. dazwischen einen veralteten Wert lesen könnten). Diese Möglichkeit dient bei globalen Operationen der Sicherstellung von Konsistenz. Darüber hinaus existieren Möglichkeiten zur Implementierung eines mutex. Weiteres hierzu findet sich bei [SK10, S. 251ff.]

Die hier kurz vorgestellten Synchronisierungstechniken sind von Bedeutung für die Einhaltung der vorgegebenen Berechnungsreihenfolge bei der parallelen Abarbeitung und finden entsprechend Anwendung im praktischen Teil.

2.3.4.3 Streams

Um einen wie in Kap. 2.2.4 beschriebenen Funktionsparallelismus (task parallelism) zu erreichen, d. h. verschiedene Funktionen oder Aufgaben parallel abzuarbeiten, stehen im Zusammenhang mit der GPU drei Möglichkeiten zur Verfügung [vgl. Far11, S. 18].

Die erste Möglichkeit, um einen Funkionsparallelismus zu erreichen, ist die gleichzeitige Nutzung der GPU und der CPU. Da Kernelaufrufe vom Host aus asynchron sind, kann das Programm während der Kernelabarbeitung weiterlaufen, d. h. der Prozessor kann andere Funktionen bearbeiten, während der Grafikprozessor seinen Teil der Berechnungen durchführt. Beim Entwurf von Algorithmen muss deshalb überlegt werden, welche Aufgaben der Prozessor in dieser Zeit erledigen kann. Gewisse Berechnungen sollten ggf. auf dem Prozessor belassen werden, um diesen ebenfalls auszulasten.

Die zweite Möglichkeit zur Erreichung eines Funktionsparallelismus ist der Einsatz mehrer Grafikprozessoren. Speziell bei Supercomputern aber auch bei Arbeitsplatzrechnern wird von der Möglichkeit Gebrauch gemacht, in einem Rechner mehrere Grafikprozessoren einzusetzen, um Berechnungen noch weiter zu beschleunigen. Da hierbei auf den unterschiedlichen Grafikprozessoren verschiedene Kernel zur Ausführung kommen können, handelt es sich hier ebenso um Funktionsparallelismus. CUDA C sieht z. B. Möglichkeiten vor, Daten direkt zwischen Grafikkarten austauschen zu können [vgl. SK10, S. 213ff.]. Da im Rahmen dieser Arbeit keine solche Konfiguration zum Einsatz kam, soll dies nicht näher ausgeführt werden.

Die dritte Möglichkeit ist die Tatsache, dass innerhalb einer GPU teilweise Funktionen parallel abgearbeitet werden können. Dies geschieht mit Hilfe der sog. Abarbeitungsströme (Streams). Sie bieten die Möglichkeit, in Form einer Pipeline mehrere Funktionen zur Ausführung vorzusehen [vgl. Far11, S. 10]. Alle in einen Abarbeitungsstrom eingereihten Kernelaufrufe bzw. Kopiervorgänge werden in FIFO-Reihenfolge (first in, first out) sequentiell abgearbeitet [vgl. Far11, S. 159]. Funktionen verschiedener Abarbeitungsströme werden jedoch entsprechend der Einplanungsreihenfolge auf eine

hardwareseitige Einheit zur Kernelausführung und eine Einheit zum Datentransfer (welche bei der eingesetzten Grafikkarte zwei parallele Datentransfers zum/vom Host erlaubt) aufgeteilt und können so teilweise parallel abgearbeitet werden. Eine Synchronisierung eines einzelnen Abarbeitungsstroms der GPU mit dem Hostprozess erfolgt durch den Befehl `cudaStreamSynchronize(stream)`. Insgesamt kann durch den Einsatz von Abarbeitungsströmen eine höhere Auslastung der GPU erreicht werden [vgl. SK10, S. 210]. Im praktischen Teil soll an Hand eines Beispiels der Einsatz von Abarbeitungsströmen und ihre Auswirkung auf die Performanz des getesteten Programms untersucht werden.

2.3.4.4 Thrust

Neben der Programmierung mittels CUDA C, welches die volle Kontrolle über die GPU und speziell über die Zuordnung von Programmteilen zu Ausführungseinheiten erlaubt (indem Kernel für einzelne Ausführungseinheiten geschrieben werden und über die Ausführungskonfiguration mitgeteilt wird, wie viele parallele Threads in wie vielen Blöcken gestartet werden), bietet die frei verfügbare Thrust Bibliothek (Thrust parallel template library) einen abstrakteren, einfacheren Blick auf die GPU. Thrust ist Bestandteil des CUDA-Toolkits und steht so allen CUDA-Entwicklern zur Verfügung.

Thrust setzt auf der Standard Template Library (STL) von C++ auf und bietet eine Reihe von generischen Funktionen, die auf den vier fundamentalen parallelen Algorithmen `for_each`, `reduce`, `scan` und `sort` aufsetzen [vgl. KH10, S. 347]. Beispiele von Thrust Funktionen sind die Transformation (`thrust::transform`) und Reduktion (`thrust::reduce`) der Daten, Präfixsummenberechnung, Sortierung, Umstrukturierung von Daten, sowie erweiterte Iteratoren (Möglichkeiten zum Durchlaufen der Daten). Ein Beispiel für einen Aufruf zur Bildung der Summe zeigt Quelltext 2.1.

Quelltext 2.1: Thrust Beispielaufruf (Quelle: [HB12]).

```
1  int sum = thrust::reduce(V.begin(), V.end(),
2              (int) 0, thrust::plus<int>());
```

Grundlegender Datentyp ist der `thrust::device_vector` bzw. der `thrust::host_vector`, der es ermöglicht, mittels einfacher Zuweisungsbefehle Daten zwischen Host und Device auszutauschen, ohne entsprechenden Speicher allokieren zu müssen. Hinter den Thrust Funktionen wie `thrust::reduce` stecken intern Kernel-Aufrufe auf der GPU, diese bleiben jedoch vor dem Entwickler verborgen. Die Möglichkeit, Zeiger auf Thrust Datentypen in Zeiger auf normale Datentypen ineinander umwandeln zu können, macht eine Mischung aus Thrust und CUDA C im Quelltext möglich. Weitere Informationen zu Thrust finden sich auf der Internetseite [HB12] und als Teil der CUDA Toolkit Dokumentation [NVI13d].

Thrust bietet vor allem eine einfache Schnittstelle mit bekannten Funktionen und somit einen leichten und schnellen Einstieg in die GPU-Programmierung. Die Funktionen sind sehr performant implementiert und es werden auch mit zukünftigeren

Versionen Performanzsteigerungen versprochen, von denen bestehende Programme automatisch profitieren können [vgl. Far11, S. 6]. Nachteil an Thrust ist, dass mit den zur Verfügung gestellten Schnittstellen dem Entwickler nur ein Teil der Funktionalität der GPU angeboten wird, zudem isoliert die Schnittstelle den Entwickler stärker von der Hardware und erlaubt keine manuellen Optimierungen [vgl. Far11, S. 6].

Im Rahmen der Arbeit soll im praktischen Teil an Hand eines Beispiels zur Berechnung von Strahlgrößen der Frage nachgegangen werden, inwiefern Thrust im Rahmen der Simulationsprogramme eingesetzt werden kann oder ob die Programmierung mit CUDA C auf Grund der flexibleren Programmiermöglichkeiten besser geeignet ist.

2.3.5 Theoretische Einordnung

Die Frage nach der theoretischen Einordnung in die in Kap. 2.2.1 vorgestellten Kategorien nach Flynn wird in der Literatur uneinheitlich beantwortet. Um einen theoretischen Zugang zur Analyse von Algorithmen für die GPU dennoch zu ermöglichen, sollen im Folgenden die Ansätze kurz vorgestellt, eine Bewertung gegeben und die im Rahmen dieser Arbeit eingenommene Sichtweise erläutert werden.

Einigkeit herrscht bei der Aussage, das prinzipiell jeder Thread so angesehen werden kann, als würde er von einem eigenen logischen Prozessor ausgeführt, der sich in einer Umgebung mit gemeinsamem Speicher befindet [vgl. SK10, S. 66] und [vgl. Far11, S.10]. Dieser Sichtweise entspricht, dass der Programmierer bei der Entwicklung eines Kernels für die GPU den Blick auf die einzelne Ausführungseinheit richtet und deren Aufgabe beschreibt. Dies geschieht vor dem Hintergrund einer späteren massiv-parallelen Ausführung mit vielen Ausführungseinheiten. Eine solche Anschauung ermöglicht es auf einfache Art und Weise, eine Vorstellung von den parallelen Ausführungseinheiten der GPU zu entwickeln.

Natürlich sind die einzelnen Ausführungseinheiten nicht unabhängig voneinander, denn es wird nur ein Kernel – also dieselbe Prozedur – auf allen beteiligten Ausführungseinheiten zur Abarbeitung gebracht. Insofern erinnert die GPU an einen Vektorrechner aus den 1970er Jahren, bei dem eine Vielzahl identischer, gleich getakteter Prozessoren dieselbe Befehlssequenz mit unterschiedlichen Daten ausführten [vgl. Tan05, S. 84]. Dass jedoch die GPU eben kein einfacher Vektorprozessor ist, ist im in Kap. 2.3.3.1 bei der Beschreibung des Aufbaus einer Grafikkarte und bei den Ausführungen in Kap. 2.3.3.2 zur Zuordnung von Programmteilen zu Ausführungseinheiten deutlich geworden. Nichtsdestotrotz ist der Vergleich mit einem Vektorrechner bezogen auf die auf einem Multiprozessor ausgeführten Threads eines Warps möglich: hier existiert ein Befehlsstrom, der von gleich getakteten Ausführungseinheiten verarbeitet wird, die mit unterschiedlichen Daten operieren. Für diese bei der verwendeten Grafikkarte 32 parallelen Threads kann demnach das SIMD Modell Anwendung finden. Diese Sichtweise herrscht auch in der Literatur vor. In [KH10, S. 88] wird für alle Threads in einem Warp für theoretische Analysen das SIMD Modell vorgeschlagen, in [Far11, S.88] wird der Aufbau des Multiprozessors bestehend aus 32 SIMD-Ausführungseinheiten beschrieben.

Bei der Betrachtung der gesamten GPU gehen jedoch die Einordnungen auseinander: [Far11, S. 90] sowie NVIDIA selbst in den CUDA Programmierrichtlinien [NVI13b] führen die neue Unterkategorie SIMT (single instruction, multiple thread) ein, um zu verdeutlichen, dass derselbe Befehl von unterschiedlichen Threads, ggf. zu unterschiedlichen Zeitpunkten ausgeführt wird. Da hier jedoch die Datenströme nicht weiter berücksichtigt werden und damit die Kategorie SIMT im eigentlichen Sinne keiner Einordnung nach Flynn entspricht, soll dieser Sichtweise an dieser Stelle nicht weiter gefolgt werden. [KH10] hingegen ordnet die GPU eher der bereits von MPI-Programmen bekannten Kategorie SPMD (single program, multiple data) zu. Hierbei arbeiten autark getaktete Prozessoren dasselbe Programm auf unterschiedlichen Daten ab. Mit dieser Einordnung ist der asynchronen Abarbeitung der einzelnen Blöcke auf der GPU Rechnung getragen. Obwohl hier die gemeinsam getaktete Abarbeitung aller Threads eines Warps nicht mehr direkt zum Ausdruck kommt, erscheint diese Zuordnung dennoch passend und auch konform zu der Kategorisierung nach Flynn. Im Folgenden soll deshalb davon ausgegangen werden, dass die GPU in die Kategorie SPMD fällt, die Threads eines Warps in die Kategorie SIMD. Eine theoretische Analyse mit dem in Kap. 2.2.2 beschriebenen Modell der PRAM ist deshalb nur für die jeweils 32 Threads eines Warps möglich. Eine Betrachtung auf Ebene der GPU kann auf Grund der unbekannten Treiberimplementierung und damit nicht vorherbestimmten Abarbeitung der Blöcke nicht einfach durchgeführt werden und soll im Rahmen dieser Arbeit nicht erfolgen. Bei der theoretischen Analyse wird sich also auf einen Multiprozessor beschränkt. Aussagen darüber hinaus auf alle Threads können zwar theoretisch getroffen werden, müssen jedoch nicht mit den tatsächlich gemessenen Ausführungszeiten übereinstimmen.

2.4 Mathematische Genauigkeit von Grafikprozessoren

Speziell bei großen Berechnungen im Rahmen von Simulationen spielen Fehlerbetrachtungen eine wichtige Rolle. Kleine numerische Fehler können sich über den Verlauf der Simulation fortpflanzen und so eine signifikante Auswirkung auf das Endergebnis haben. An dieser Stelle soll kurz darauf eingegangen werden, welcher Fehler durch die Umsetzung mit Hilfe des Rechners und speziell durch den Einsatz der GPU zu erwarten ist. Auf andere Fehler, z. B. durch die genutzte Simulationsmethode, soll an dieser Stelle nicht weiter eingegangen werden.

Zur Emittlung des Fehlers müssen die Rundungsfehler bei der Zahlenrepräsentation betrachtet werden. Ihre Auswirkungen auf die Teilchenkoordinaten im Rahmen des Strahltransports werden im Folgenden untersucht. Dabei soll die Frage beantwortet werden, ob die Teilchenkoordinaten als Gleitkommazahl einfacher oder doppelter Genauigkeit abgelegt werden sollen. Zunächst wird deshalb kurz auf die Darstellung von Gleitkommazahlen in Rechnern eingegangen. Danach werden auftretende Rundungsfehler betrachtet und ihre Auswirkungen abgeschätzt. Abschließend wird auf spezielle Aspekte in Zusammenhang mit der GPU eingegangen.

2.4.1 Darstellung von Gleitkommazahlen

Bei der Datenverarbeitung mit Hilfe eines Computers werden Zahlen in Speicherplätzen fixer Größe abgelegt, wodurch die Genauigkeit der abgelegten Zahlen endlich ist (finite-precision numbers [vgl. Tan05, S. 715]). Der darstellbare Wertebereich und die Genauigkeit der abgelegten Zahl lassen sich voneinander trennen, indem für reelle Zahlen die wissenschaftliche Notation $n = \pm m \times b^e$ gewählt wird, wobei m die Mantisse (der gebrochene Anteil), b die Basis und e der (ganzzahlige) Exponent ist. Die Anzahl der Ziffern des Exponenten bestimmen den darstellbaren Wertebereich, die der Mantisse die Genauigkeit [vgl. Tan05, S. 729].

Die Umsetzung dieser wissenschaftlichen Notation in Computersystemen in Form der sog. Gleitkommadarstellung ist im IEEE-Standard 754 [IEE08] definiert. Darin werden drei Formate festgelegt: einfache Genauigkeit (32 Bit), doppelte Genauigkeit (64 Bit) und erweiterte Genauigkeit (80 Bit). Die ersten beiden können im Zusammenhang mit der GPU-Programmierung verwendet werden und sollen kurz dargestellt werden.

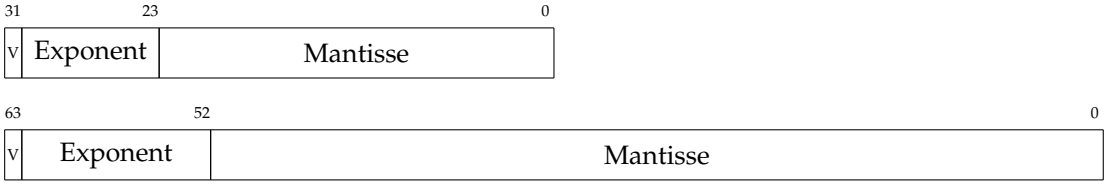

Abbildung 2.9: Gleitkommazahlen einfacher und doppelter Genauigkeit, jeweils mit Vorzeichen, Exponent und Mantisse.

Zur Darstellung von Gleitkommazahlen in einfacher oder doppelter Genauigkeit wird die Basis zwei für die Mantisse verwendet. Der Exponent wird in Exzess-Notation dargestellt, dabei ist das erste Bit das Vorzeichenbit für die Zahl als Ganzes (0 positiv, 1 negativ) [vgl. Tan05, S. 733]. Die Mantisse besteht immer aus der 1 vor dem Komma (normalisierte Darstellung), welche nicht gespeichert wird, danach folgt der Nachkommateil. Die Mantisse kann also Werte im Intervall $[1,0 \ldots 2,0[$ annehmen (diese Definition der Mantisse wird als Signifikant bezeichnet). Die möglichen Dezimalbereiche einer solchen Zahlendarstellung in einfacher bzw. doppelter Genauigkeit decken 10^{-38} bis 10^{38} bzw. 10^{-308} bis 10^{308} ab [vgl. Tan05, S. 734].

2.4.2 Fehlerbetrachtung

Die reellen Zahlen bilden in der Mathematik ein Kontinuum, d. h. zwischen zwei reellen Zahlen lässt sich immer eine weitere reelle Zahl angeben, was allerdings für die Gleitkommazahlen in Rechnersystemen nicht der Fall ist [vgl. Tan05, S. 730]. Viele Zahlen lassen sich deshalb nicht genau ausdrücken; eine Rundung auf die nächste darstellbare Zahl wird nötig. In der Fehleranalyse ordnet man die so entstehenden Rundungsfehler den Rechenfehlern zu [vgl. Übe95, S. 32]. Im Rahmen einer Berechnung wird jedes Teilergebnis gerundet, die Auswirkungen all dieser Fehler auf das Endergebnis werden Rechenfehlereffekt genannt [vgl. Übe95, S. 39].

Der Abstand der Gleitkommazahlen ist für kleine Zahlen deutlich geringer als für große. Drückt man den entstandenen Rundungsfehler jedoch relativ aus, ist dieser für kleine und große Zahlen annähernd gleich groß [vgl. Tan05, S. 730]. Erhöht man die Anzahl der Ziffern der Mantisse, nimmt die Punktdichte zu, so dass sich die Genauigkeit von Näherungen verbessert. Die Effekte treten deshalb bei Nutzung der doppelten Genauigkeit später auf [vgl. Übe95, S. 41].

Der maximale relative Fehler, der bei der Darstellung einer reellen Zahl auftritt, ist definiert als

$$\epsilon = \frac{1}{2} 2^{-(m+1)+1} = 2^{-(m+1)} \tag{2.4.1}$$

und wird als Maschinengenauigkeit bezeichnet. Bei einfacher bzw. doppelter Genauigkeit ergibt sich also eine Maschinengenauigkeit von $\epsilon = 2^{-(23+1)} \approx 6,0 \cdot 10^{-8}$ bzw. $\epsilon = 2^{-(52+1)} \approx 1,1 \cdot 10^{-16}$.

Laut den CUDA C Programmierrichtlinien wird bis auf wenige Abweichungen der IEEE 754-2008 Standard unterstützt, speziell gibt es im Vergleich zu der darin definierten Zahlendarstellung keinen abweichenden Fehler [vgl. NVI13b, Abschnitt C-1]. Voreingestellt ist dabei der Rundungsmodus „round to nearest even“, d. h. es wird zur nächsten darstellbaren Zahl hin gerundet und falls der Wert genau in der Mitte liegt, wird so gerundet, dass die letzte Stelle gerade ist [vgl. WFF11, S. 4]. Aus diesem Grund bezieht sich die folgende Betrachtung sowohl auf die CPU als auch auf die GPU.

Um den Rundungsfehler bei der Darstellung der Teilchenkoordinaten abzuschätzen, der im Verlauf der Simulation auftritt, müssen die Teilchenkoordinaten selbst und die Operationen betrachtet werden, die diese verändern. Da die Koordinaten relativ zum synchronen Teilchen angegeben werden, sind die vorkommenden Zahlen kleiner eins, jedoch auch nicht so klein, dass sie in den Bereich der denormalisierten Zahlen (zwischen der kleinsten darstellbaren Zahl und null) fallen. Es kann weiterhin davon ausgegangen werden, dass die vorkommenden Zahlen dieselbe Größenordnung besitzen und im Verlauf der Berechnungen insb. auch bei der Addition keine Anpassung der Mantisse erfolgen muss [vgl. auch KH10, S. 161].

Im Lauf der Simulation durchläuft ein Teilchen viele Beschleunigerelemente, jeweils repräsentiert durch eine 6×6-Matrix. Die Fortbewegung wird als Multiplikation der Matrix mit dem Teilchenvektor dargestellt, vgl. Kap. 2.1. Bei genauerer Betrachtung der Matrizen in Anhang B.1 fällt auf, dass meist nur zwei Einträge pro Zeile besetzt sind, die eine Auswirkung auf eine Teilchenkoordinate haben, in den restlichen Spalten befindet sich meist eine null. Gehen wir zunächst von einer bereits fehlerhaften Teilchenkoordinate x_1 aus, dargestellt durch $\tilde{x_1}$. Im einfachen Fall einer Driftstrecke ergibt sich die Koordinate beim Durchlaufen der Driftstrecke zu:

$$\tilde{x_2} = 1 \cdot \tilde{x_1} + L \cdot \tilde{x_1}' \tag{2.4.2}$$

Weiter wird von der Annahme ausgegangen, dass die Matrizen selbst exakt sind. Im Folgenden wird Δx verkürzend für $(x - \tilde{x})$ geschrieben. Dann ergibt sich der relative

Fehler von $\tilde{x_2}$ zu:

$$\begin{aligned} x_2 &= \frac{\Delta x_1 \cdot x_1}{x_1} + \frac{\Delta x_1' \cdot x_1'}{x_1'} \\ &= 1 \cdot \epsilon \cdot x_1 + L \cdot \epsilon \cdot x_1' \\ \mathrm{err}_{rel}(x_2) &= 2\epsilon \end{aligned} \tag{2.4.3}$$

Im zweiten Schritt können x_1 und x_1' vernachlässigt werden, da die Koordinaten immer relativ zum synchronen Teilchen betrachtet werden, und es sich deshalb um Größen kleiner 1 handelt. Beim Durchlaufen von k Beschleunigerelementen ergibt sich so ein Rundungsfehler von

$$\mathrm{err}_{abs}(x_n) = x_1 \cdot (1 + 2\epsilon)^n \tag{2.4.4}$$

Möchte man, dass trotz länger laufender Simulation die Teilchenkoordinate bis auf 5% exakt bleibt, so ist dies nach

$$\begin{aligned} x_1 \cdot (1 + 2\epsilon)^n &= x_1 \cdot 1,05 \\ (1 + 2\epsilon)^n &= 1,05 \\ n &= \frac{\log(1,05)}{\log(2^{-24+1} + 1)} \\ n &\approx 8 \cdot 10^5 \end{aligned} \tag{2.4.5}$$

Beschleunigerelementen bei der Nutzung von einfacher Genauigkeit und nach $4 \cdot 10^{14}$ Elementen bei Nutzung doppelter Genauigkeit unterhalb der angegebenen Schranke möglich. Für langlaufende Simulationen kann es deshalb nötig sein, die Teilchenkoordinaten als `double`-Werte zu repräsentieren.

Bei der Ermittlung von Strahlgrößen im Rahmen der Simulation kommt darüber hinaus das Problem hinzu, dass Werte über die gesamten Teilchen gebildet werden sollen. Als Teil der Berechnung werden z. B. Mittelwerte einzelner Koordinaten gebildet. Bei einer solchen Mittelwertbildung kann zunächst das Problem entstehen, dass bei der Summe Zahlen sehr nah bei null auftreten, da die einzelnen Koordinaten durch die Betrachtung relativ zum synchronen Teilchen um null herum schwanken. Um den damit verbundenen Problemen zu umgehen, könnte man zunächst einen positiven und einen negativen Mittelwert bilden, bevor man beide zusammenfasst. Bei einer solchen Mittelwertbildung kann dann jedoch das Problem auftreten, dass zu einer immer größer werdenden Teilsumme immer jeweils nur kleine Beträge hinzuaddiert werden sollen. Diese kleinen Beiträge können dabei auf Grund der nötigen Mantissenanpassung sogar weggeschnitten werden, d. h. gegen Ende der Summenbildung verändert sich die Summe durch die zusätzliche Addition nicht mehr. Zur Lösung dieses Problems wird in der Literatur vorgeschlagen, die Summierung paarweise oder blockweise durchzuführen, um bei der Addition jeweils mit gleich großen Zahlen arbeiten zu können [vgl. Hig02]. Eine Abschätzung der so entstehenden Fehler findet sich bei [Übe95] und ist in Anhang B.2 angegeben. Diese Tatsache wird im Rahmen der praktischen Implementierung bei der Emittanzermittlung auf der GPU insofern berücksichtigt, als dass dort ein binäres Summierungsverfahren zum Einsatz kommt.

Obwohl die oben aufgezeigten Fehler durch die Darstellung als Gleitkommazahl im Rechnersystem Fehler birgt, werden die Fehler in der Praxis nicht den abgeschätzten Größen entsprechen, sondern kleiner sein. Generell ist dabei von einer Kompensation einzelner Rechenfehler auszugehen [vgl. Übe95, S. 41]. Für die Praxis sollte man Rundungsfehler bei der Verwendung der Gleitkommadarstellung jedoch beachten, und z. B. neuere Gleitkommafunktionen auf der GPU wie FMA (fused multiply-add, eine Multiplikation und Addition in einem Schritt mit nur einem Rundungsschritt) mit geringeren Rundungsfehlern einsetzen, wie z. B. in [WFF11, S. 6] vorgeschlagen.

Neben der Fehlerbetrachtung ist in Bezug auf die GPU die Tatsache relevant, dass die Berechnungen mit einfacher Genauigkeit in der Vergangenheit deutlich schneller waren, als mit doppelter Genauigkeit. Allerdings geht hier, bedingt durch die bestehende Nachfrage, die Entwicklung dahin, dass die Performanz der Berechnungen doppelter Genauigkeit mit jeder neuen Grafikkartengeneration besser wird. Bei der im Rahmen der Diplomarbeit verwendeten Grafikkarte betrug die Geschwindigkeit von Berechnungen doppelter Genauigkeit bereits 50%, bei noch neueren neueren Grafikkarten sogar schon 80% der Geschwindigkeit mit einfacher Genauigkeit. Abgewogen werden muss also zwischen den Performanzeinbußen einerseits und der zu erwartenden Verschlechterung der Simulationsergebnisse andererseits. Im praktischen Teil der Arbeit wird zunächst generell mit doppelter Genauigkeit gearbeitet, da dies für die vorliegenden Simulationen als nötig erscheint. Jedoch soll beispielhaft im Rahmen der Parallelisierung des Strahltransports der Laufzeitunterschied im Vergleich zum Einsatz von einfacher Genauigkeit ermittelt werden.

Über die genannten Probleme der Rundungsfehler hinaus ist speziell für den im praktischen Teil der Arbeit nötigen Vergleich der Ergebnisse mit denen der CPU zu beachten, dass auf Grund der Rundung bei der Zahlendarstellung die Gesetze der Arthimetik, nämlich das Assoziativitäts- und Distributivgesetz für Gleitkommazahlen, nicht mehr gelten [vgl. Übe95, S. 158]. Insb. kann also das Endergebnis auf der CPU und GPU bedingt durch den jeweiligen Rechenweg leicht unterschiedlich sein. Dies ist bei der praktischen Implementierung zu beachten.

3 Simulation mit den bestehenden Particle-in-cell Codes

Im Rahmen der vorliegenden Diplomarbeit wird untersucht, wie bestehende Teilchensimulationsprogramme mit Hilfe von Grafikkarten parallelisiert werden können. Gegenstand der Betrachtung sind dabei die bestehenden Simulationsprogramme PATRIC und LOBO. Von beiden Programmen kam dabei jeweils eine reduzierte Version zum Einsatz, die speziell auf die zu parallelisierenden Aspekte konzentriert ist.

In diesem Kapitel werden beide Programme in ihrer reduzierten Originalversion vorgestellt. Dabei wird zunächst der Aufbau beschrieben, um einen Überblick über die zentralen Strukturen des Programms zu geben. Anschließend wird der Ablauf einer Teilchensimulation mit dem jeweiligen Programm vorgestellt. Das Verständnis sowohl der Strukturen als auch der funktionalen Abläufe ist essentiell für die Ansätze zur Parallelisierung im praktischen Teil dieser Arbeit.

3.1 Das Simulationsprogramm PATRIC

3.1.1 Übersicht und Aufbau

Das Simulationsprogramm PATRIC (Particle Tracking Code) wird von der Abteilung Strahlphysik der GSI entwickelt. Es wird bei GSI für viele Arten von Teilchensimulationen eingesetzt, aktuell z. B. um kollektive Effekte in den zukünftigen Beschleunigern der neuen FAIR-Anlage zu untersuchen. Das Programm ist eine rein interne Entwicklung, die speziell auf die Belange und Problemstellungen bei GSI angepasst ist. Weiterführende Informationen zu PATRIC finden sich in [BFK06].

Das Programm enthält Funktionalitäten, um Teilchen mit verschiedenen Verteilungsfunktionen entlang des Beschleunigers zu simulieren und die unterschiedlichsten Effekte zu studieren. Eine erste Version des Programms PATRIC entstand bereits Anfang der 1990er Jahre [vgl. Kal94]; bereits damals wurde auf einen modularen Aufbau des Programms geachtet. Die heutige Version von PATRIC ist in C++ geschrieben. Auf Grund der Modularisierung ist es leicht möglich, eigene Programmteile einzufügen. Entsprechend existieren verschiedene PATRIC-Versionen, die jeweils auf die zu untersuchende Fragestellung hin ausgerichtet sind. Das Programm selbst bietet keine grafische Oberfläche, sondern fokussiert rein auf die physikalischen Routinen zur Teilchensimulation. Die entweder in Binär- oder ASCII-Format erzeugten Ausgabedateien werden mit anderen Programmen (meist auf der Basis von Python) ausgewertet.

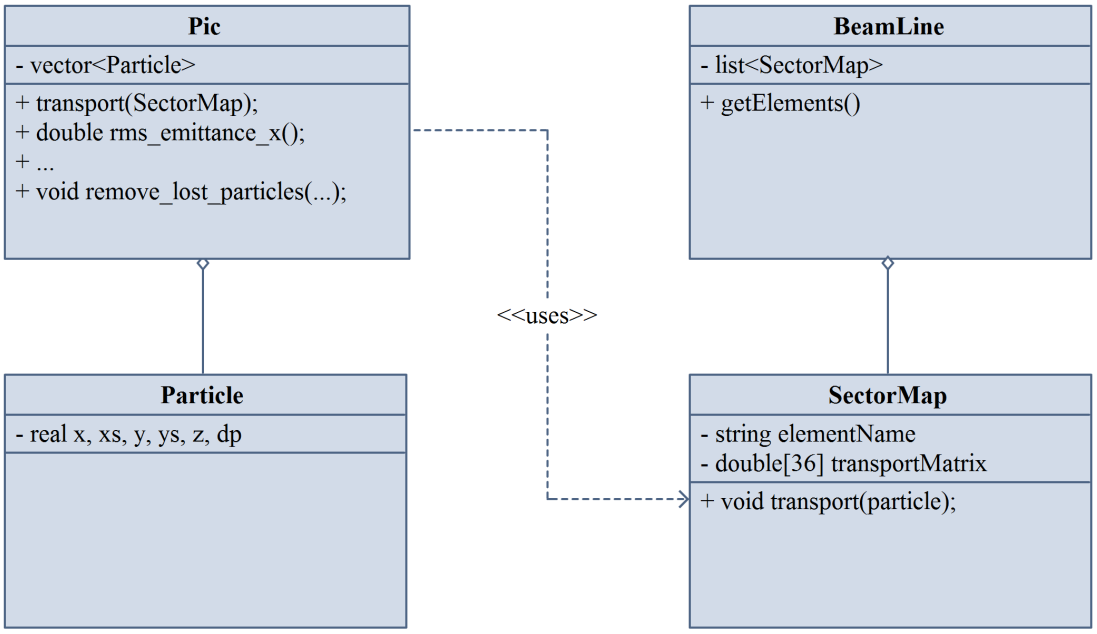

Abbildung 3.1: Objekte des PATRIC Simulationsprogramms.

Die zentralen Objekte des Simulationsprogramms PATRIC sind in Abb. 3.1 dargestellt. Die Pic-Klasse ist ein Container für die zu simulierenden Teilchen. Sie enthält die Teilchen sowie Auswertemethoden für die relevanten Strahlgrößen. Die Teilchen selbst sind dabei nicht als eigene Klasse, sondern als `struct` abgebildet, welches den Teilchenvektor repräsentiert. Die SectorMap Klasse stellt die Beschleunigerelemente wie Magnete zusammen mit ihrer Transportmatrix dar und enthält z. B. die Methode, um eine Menge von Teilchen durch das repräsentierte Element zu transportieren. Diese Stelle bietet sich entsprechend als Einstiegspunkt für die Parallelisierung mit Hilfe der GPU an. Die BeamLine Klasse fasst alle SectorMap-Instanzen in der entsprechend richtigen Reihenfolge zu einem gesamten Beschleuniger zusammen. Das Hauptprogramm von PATRIC kümmert sich um das Einlesen der Konfigurationsdatei, das Aufsetzen des Startzustandes und um die zentrale Steuerung des Ablaufs der Simulation.

Das bestehende PATRIC-Programm nutzt für die Parallelisierung MPI (vgl. Kap. 2.2.5); als Laufzeitumgebung dient das bestehende Linux-Cluster der GSI. Der Parallelisierung liegt das Modell des verteilten Speichers mit Nachrichtenaustausch zu Grunde. Da zur Verfolgung vieler tausend Teilchen eine große Zahl an identischen Berechnungen mit den Teilchendaten ausgeführt werden müssen, wurde für die Verteilung der Berechnungen auf die unterschiedlichen MPI-Knoten eine Datenzerlegung gewählt.

Da Teilchen transversal stärker miteinander interagieren als longitudinal, wird das zu simulierende Teilchenpaket longitudinal in einzelne „Scheiben“ (engl. slices) aufgeteilt. Die einzelnen MPI-Knoten (Anzahl typischerweise 2-16) bekommen jeweils eine Scheibe des zu simulierenden Teilchenpakets, siehe Abb. 3.2. Dabei wird initial eine Lastverteilung vorgenommen, indem jeder Knoten gleich viele Teilchen erhält, d. h.

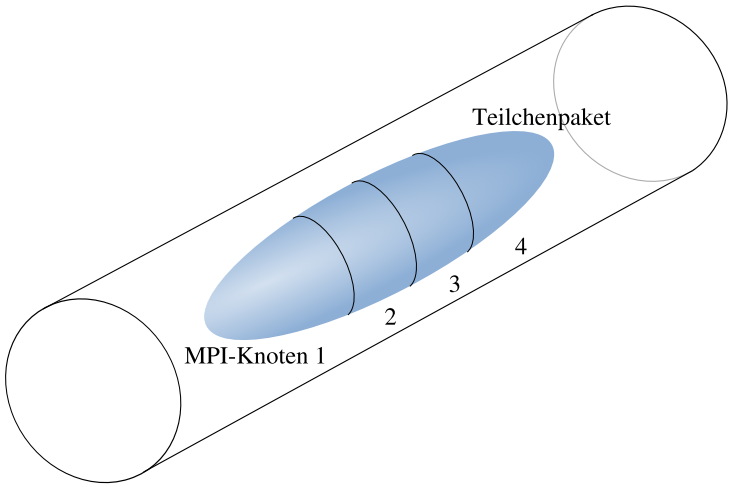

Abbildung 3.2: Aufteilung der Teilchen auf MPI-Knoten in PATRIC: Jeder MPI-Knoten bekommt eine Scheibe des Teilchenpakets, die Variierung der Dicke der Scheibe ermöglicht eine gleichmäßige Verteilung der Teilchen auf die Knoten.

die Dicke der Scheibe variiert zwischen den Knoten. Im Laufe der Berechnung können Teilchen ihre Position so verändern, dass sie in die Scheibe des benachbarten Knotens wandern. Aus diesem Grund erfolgt nach jedem lokalen Berechnungsschritt der Austausch von Teilchen an den Rändern der Scheiben mit den benachbarten MPI-Knoten. Mit der bestehenden Parallelisierung wird bereits ein guter Performanzgewinn erzielt und diese wird routinemäßig eingesetzt.

3.1.2 Ablauf einer Teilchensimulation

Zum Parametrieren der Simulation existiert in PATRIC eine Konfigurationsdatei. Die Einstellungen, wie sie im Rahmen der Arbeit genutzt wurden, finden sich in Anhang C.1. Zu der Initialisierung einer Simulation gehört u. a. das Aufsetzen des Beschleunigers und der Teilchen. Der Beschleuniger wird dabei entweder als konstant fokussierende Optik intern im Programm generiert oder aus einer externen Datei eingelesen, welche den Beschleuniger mit seinen Elementen beschreibt. Danach liegen die für den Transport relevanten (statischen) Transportmatrizen vor. Die zu simulierenden Teilchen werden mit der gewählten Anfangsverteilung im Raum erzeugt. Danach kann die Simulation gestartet werden.

Nach dem Aufsetzen der Simulation und im Fall von MPI ggf. der Verteilung der Teilchen auf die einzelnen MPI-Knoten startet die eigentliche Simulation. Die zentrale Schleife im Programm PATRIC, die die Simulation steuert, ist in Quelltext 3.1 dargestellt.

Quelltext 3.1: PATRIC Originalversion: Zentrale Schleife des Teilchentransports (verkürzt).

```
1  do {
2    if (counter \% Nplot == 0) {
3      // print..
4    }
5
6    // Transport particles through sectormap, update slice position s
7    ds = lattice.get_element()->get_L();
8    s += ds;
9    Pics.transport(lattice.get_element()->get_map(), piperadius);
10
11   // Exchange particles between slices
12   if (counter != 0 && counter \% Nexchange == 0 && numprocs > 1) {
13     MPI_Send(...);
14     MPI_Recv(...);
15     Pics.add_particles(particles_in);
16   }
17
18   // Advance in beam line, go to next element
19   lattice.next_element();
20   counter++;
21
22 } while (counter != cells * Nelements + 1);
```

Die Simulation wird so lange durchgeführt, bis entweder die vorgegebene Anzahl Simulationsschritte erreicht ist, oder der Teilchenverlust größer wird als eine einstellbare Schwelle. Bei einer echten Simulation wäre ein solcher Teilchenverlust natürlich ein valides Ergebnis; für diese Arbeit sind jedoch die Bedingungen so gewählt, dass es zu keinem Teilchenverlust kommen sollte. Ein dennoch auftretender Teilchenverlust könnte so ein Hinweis auf Probleme im Programm sein (z. B. auf Grund von Rundungsfehlern) – dies kam jedoch im Rahmen der Arbeit nicht vor.

Die Endergebnisse einer Simulation werden in Form von Binärdateien ausgegeben. Es handelt sich dabei um die endgültige Teilchenverteilung sowie die Strahlparameter. Diese können mit externen Programmen weiter ausgewertet werden. Neben dieser Endergebnisse sind jedoch auch Zwischenergebnisse von Interesse, so z. B. die Beobachtung der Strahlparameter wie die Emittanz (vgl. Kap. 2.1.2) über den gesamten Verlauf der Simulation. Teilweise werden sogar für genauere Analysen die Positionen aller Teilchen ausgegeben. Die Häufigkeit, mit der Zwischenergebnisse ausgegeben werden sollen, ist dabei parametrierbar.

Speziell für die Parallelverarbeitung sind die Ausgaben dieser Zwischenergebnisse eine Herausforderung, da die Informationen an zentraler Stelle zusammengeführt und ausgegeben werden müssen. Im Fall mit MPI bedeutet dies, dass diese Daten von allen Slaves zentral an den Master kommuniziert werden müssen. Im Fall der Einbeziehung der GPU bedeutet es zudem, dass die gesamten Teilchen oder zumindest die ermittelten Strahlparameter zunächst wieder auf den Host zurückkopiert werden müssen, bevor sie ausgegeben werden können. Da aber die Zwischenergebnisse unabdingbar sind, stellt dies eine zentrale Problemstellung im Rahmen des praktischen Teils dar.

3.2 Das Simulationsprogramm LOBO

3.2.1 Übersicht und Aufbau

Das Simulationsprogramm LOBO (Longitudinal Beam Dynamics Simulations Code) wird von der Abteilung Strahlphysik der GSI entwickelt. LOBO dient speziell der Betrachtung longitudinaler Effekte. Diese werden in Form einer 1D-Simulation beschrieben, in erster Ordnung gibt es hierbei keine Kopplung zum Transversalen, weshalb sich wirklich auf die longitudinale Betrachtung beschränkt werden kann. Äussere Elemente sind hierbei nicht wie im Transversalen die Magnete, sondern die Hochfrequenzanlage mit ihren longitudinalen Kräften. Das Programm bietet dabei eine Reihe an Funktionalitäten. U. a. kann der Einfang der Teilchen simuliert werden, dabei bilden sich ausgehend von einem gleichförmigen Strahl mit eingeschalteter Hochfrequenzanlage Teilchenpakete. Unter Hinzunahme der Raumladungseffekte kann beispielsweise untersucht werden, ob der Strahl unter der Raumladungswirkung als Teilchenpaket verbleibt oder instabil wird. Weitere Informationen zu LOBO und seiner Verwendung finden sich bei [BFH00]. Im Rahmen der Arbeit kam eine reduzierte Version von LOBO zum Einsatz, die die Auswirkungen der Hochfrequenzanlage ausklammert und bei der das Hauptaugenmerk auf dem Einbeziehen der Raumladung liegt.

Das Programm selbst ist in C++ geschrieben und erzeugt Ausgabedateien im Binär- oder ASCII-Format, die mit Hilfe weiterer Programme ausgewertet werden. Das Programm enthält aktuell noch keine parallelen Ansätze.

Die zentralen Objekte des LOBO Programms zeigt Abb. 3.3. Es handelt sich hierbei um die Teilchen selbst, repräsentiert durch einen 2-komponentigen Vektor mit z als longitudinale Ortsabweichung und dp als Implusabweichung. In der Klasse Beam werden alle Teilchen zusammengefasst; hier sind Methoden angesiedelt, um Verteilungen zu ermitteln, die Teilchen auf ein gegebenes Gitter zu interpolieren und die Teilchen zu bewegen. Grid1D und Grid2D stellen die genutzten Gitter dar, wobei das 2D-Gitter rein zu Ausgabezwecken dient. Die Klasse BBImpedance enthält Methoden, um z. B. aus einer gegebenen Teilchenverteilung das elektrische Feld zu berechnen. Das Hauptprogramm von LOBO liest die Konfigurationsdatei ein und steuert wiederum den Simulationsablauf.

3.2.2 Ablauf einer Teilchensimulation

Die grundlegende Konfiguration von LOBO mit u. a. der gewünschten Teilchenanzahl, Teilchenverteilung sowie Gittergröße wird wieder mit Hilfe einer Konfigurationsdatei vorgenommen, die beim Start des Programms eingelesen wird. Eine typische im Rahmen dieser Arbeit verwendete Konfiguration ist in Anhang C.2 dargestellt.

Nachdem die initiale Teilchenverteilung gemäß der Einstellungen generiert wurde, arbeitet das Programm nach dem in Abb. 2.2 dargestellten und dort skizzierten allgemeinen Ablauf einer Particle-in-cell Simulation. Die zentrale Schleife in der Abarbeitung

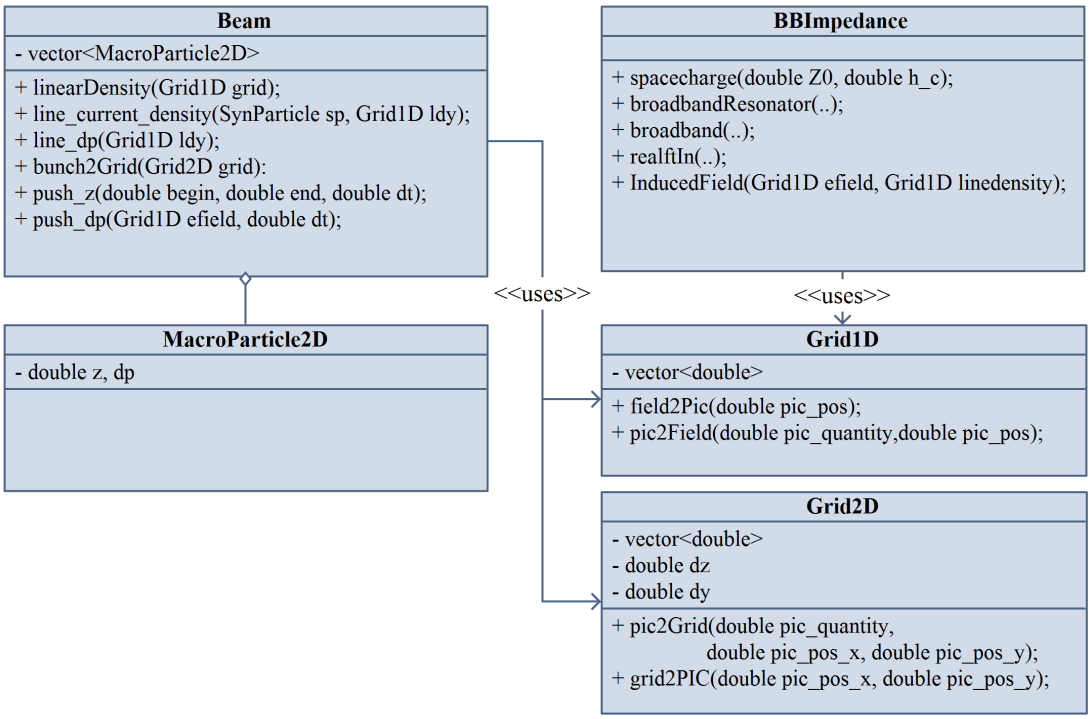

Abbildung 3.3: Objekte des LOBO Simulationsprogramms.

von LOBO ist im Quelltextauszug 3.2 gezeigt. Zyklisch werden die Teilchen zunächst auf Grund des elektrischen Feldes fortbewegt. Zur Ermittlung der neuen Felder und deren Auswirkungen auf die Teilchen werden die Teilchen linear auf die benachbarten Gitterpunkte interpoliert. Dazu wird die Strom- und Geschwindigkeitsverteilung auf einem 1D-Gitter bestimmt, zu Darstellungszwecken wird zusätzlich eine Interpolation auf ein 2D-Gitter vorgenommen. Das resultierende elektrische Feld wird mit Hilfe der schnellen Hin- und Zurück-Fourier-Transformation (FFT) ermittelt. Bei der Ermittlung des Feldes können weitere Komponenten in Form von Impedanzen einbezogen werden. Danach beginnt der Berechnungszyklus erneut mit dem Fortbewegen der Teilchen. Weitere Informationen zum Ablauf der Simulation in LOBO finden sich bei [App11].

Im Rahmen des praktischen Teils sind die beschriebenen Simulationsschritte auf eine mögliche Parallelisierung hin zu untersuchen. Dabei müssen die Laufzeiten der einzelnen Schritte Berücksichtigung finden.

Die Simulation läuft so lang, bis die vorgegebene Anzahl Zeitschritte erreicht ist. Die Ergebnisse bestehend aus Teilchendaten, Feldern und Gitterwerten werden in Form von Binärdateien ausgegeben und stehen dann für weitere Auswertungen zur Verfügung.

Quelltext 3.2: LOBO Originalversion: Zentrale Schleife der Berechnung.

```
1  // time step
2  while (time_n <= t_max) {
3    // print step
4    if (j \% print_step == 0) {
5      print(time_n, grid2D, lineDensity, efield1D, densityDp, beam,
           impedance, diag, synchronousParticle);
6      m++; //printsteps
7    }
8
9    beam.propagate(efield1D, lineDensity, impedance, dt);
10   beam.bunch2Grid(grid2D);
11   beam.LinDens(grid1D);
12   beam.line_dp(densityDp);
13   time_n += dt;
14 }
```

4 Parallelisierungsstrategien anderer Particle-in-cell Codes

Auf Grund der hohen Verbreitung von programmierbaren Grafikkarten und den kostenlos verfügbaren generischen Programmierschnittstellen werden diese in zunehmendem Maße für Parallelisierungsaufgaben eingesetzt. Betrachtet man den aktuellen Stand der Forschung, finden sich gerade im Bereich der Teilchen- und Fluidsimulationen eine Fülle von Untersuchungen zum Einsatz von Grafikprozessoren.

Im Folgenden wird der aktuelle Stand kurz vorgestellt. Dazu wird auf zwei Simulationsprogramme näher eingegangen, die bereits Unterstützung für Grafikprozessoren enthalten und die beide einen engeren Bezug zu den im Rahmen der Arbeit betrachteten Programmen aufweisen. Es handelt sich dabei um das am Argonne National Laboratory in den USA entwickelte Teilchensimulationsprogramm ELEGANT und das am Helmholtz-Zentrum Dresden-Rossendorf entwickelte Programm PIConGPU. Beide stellen unterschiedliche Beispiele für die Einbeziehung der Grafikkarte in Teilchensimulationsprogrammen dar. Beide Programme werden jeweils kurz vorgestellt und in Bezug auf ihre Parallelisierungsstrategien analysiert, um einen Eindruck davon zu erhalten, welche Maßnahmen in anderen Programmen umgesetzt wurden. Anschließend folgt eine kurze Betrachtung beispielhaft ausgewählter Parallelisierungsstrategien aus anderen Veröffentlichungen zum vorliegenden Thema.

4.1 Einsatz von GPUs im Simulationsprogramm ELEGANT

Das Teilchensimulationsprogramm ELEGANT (ELEctron Generation ANd Tracking) wird vom Argonne National Laboratory in den USA entwickelt, unter einer Open Source Lizenz publiziert und kann unter [Bor10] bezogen werden. Das Programm wurde für die Simulation des dortigen Linear- und Ringbeschleunigers entwickelt, ist aber heute ein umfängliches Simulationsprogramm, das allgemein zur Verfügung steht und weltweit eingesetzt wird. An der GSI wird es u. a. zur Simulation der Extraktion im zukünftigen Beschleuniger SIS300 eingesetzt [vgl. SHFP+08].

Das Programm bietet Funktionalitäten für das Design, die Simulation und die Optimierung von Beschleunigern und ist in der Programmiersprache C geschrieben. Obwohl frühe Versionen bereits Mitte der 1980er Jahre entstanden, verfolgt ELEGANT einen modernen, modularen Ansatz. Als Kommandozeilen-Programm ist es auf die eigentliche Berechnung der Teilchensimulation fokussiert und bietet selbst keine grafische Oberfläche. Dies ist der Grund, weshalb es heute genauso wie damals einge-

setzt werden kann [vgl. Bor12, S. 4]. Vor- oder Nachverarbeitung werden mit anderen Programmen erledigt, man bedient sich hier ganzer Programmfamilien, die auf dem standardisierten Ausgabeformat SDDS [vgl. BE95] aufsetzen.

Bei einem Simulationslauf generiert ELEGANT Teilchenverteilungen und simuliert die Teilchenbewegung entlang des Beschleunigers. Grundlegendes Modell sind dabei die sogenannten Elemente, das sind Objekte, die die Teilchen auf ihrem Weg durch den Beschleuniger durchlaufen. Elemente können z. B. Magnete sein, die die Teilchenbahn beeinflussen. Auch kollektive Effekte der Teilchen untereinander werden in ELEGANT mit Hilfe von Elementen modelliert, z. B. existiert ein Element, dass die longitudinale Raumladung repräsentiert. Ein Element kann jedoch auch ein Diagnoseelement sein, das einen bestimmten Messwert wie z. B. die Emittanz ermittelt. Jedes dieser Elemente spiegelt sich als einzelne Datei wieder, was den Quelltext übersichtlich macht. Weitere Informationen zum Aufbau des Programms und zur Verwendung finden sich bei [Bor12] und [WB06].

4.1.1 Parallelisierung mit MPI

Um zeitintensive Berechnungen beschleunigen zu können, entstand mit PELEGANT (Parallelized ELEGANT) 2006 eine Version, die Parallelverarbeitung mittels MPI unterstützt [vgl. WB06]. Beide Programmversionen basieren auf demselben Quelltext; die Parallelverarbeitung ist optional und wird durch Compilerdirektiven ein- oder ausgeschaltet. Bei PELEGANT kommt das in Kap. 2.2.5.2 beschriebene Master-Slave-Modell zum Einsatz. Der Master verteilt die Daten (entspricht einer Datenzerlegung) und sammelt die Ergebnisse wieder ein, die Slaves berechnen Teilergebnisse jeweils auf ihrem Teil der Daten.

Wie angesprochen ist PELEGANT bzw. ELEGANT in Elemente aufgeteilt, die die Teilchen auf ihrem Weg durch den Beschleuniger durchlaufen; dies entspricht einer Funktionszerlegung. Auf Grund dieser Strukturierung und dem Wunsch, das Programm beginnend mit den berechnungsintensivsten Elementen sukzessive zu parallelisieren, wurde bei PELEGANT der Ansatz gewählt, Element für Element zu entscheiden, ob dies bereits parallelisiert ist oder nicht [vgl. WB06, S. 1]. Elemente in PELEGANT werden aus diesem Grund in die folgenden Kategorien unterteilt [vgl. WB06, S. 2]:

- Paralleles Element: Die Berechnung erfolgt ausschließlich auf den Slaves, jeder Slave-Knoten ist für einen Teil der Teilchen zuständig.
- Multiprozessor Algorithmus: Die Berechnung erfolgt auf den Slaves, der Master führt die Teilergebnisse hinterher zusammen (Summe, Durchschnitt, o. ä.).
- Uniprozessor Element: Die Berechnung erfolgt an zentraler Stelle durch den Master, sie verändert die Teilchenkoordinaten (gilt zunächst für alle noch nicht parallelisierten Elemente).
- Diagnose Element: Die Berechnung erfolgt an zentraler Stelle durch den Master, allerdings werden keine Teilchenkoordinaten verändert.

Bei der Bewegung der Teilchen durch den Beschleuniger wird an Hand der Kategorie des nächsten Elements ermittelt, ob die Teilchen entweder an den Master oder an die Slaves verschickt werden müssen, oder dort verbleiben können, wo sie sich gerade befinden. Bei zwei aufeinander folgenden parallelen Elementen können die Teilchen z. B. auf den Slaves verweilen. Bei einem Element zur Diagnose müssen die Teilchen vom Master eingesammelt werden. Da die Diagnose jedoch die Teilchenkoordinaten nicht verändert, müssen die Teilchen danach nicht erneut an die Slaves verteilt werden.

PELEGANT ist so organisiert, dass die einzelnen Slave-Knoten eine möglichst gleich große Anzahl an Teilchen bearbeiten sollen. In der Standardeinstellung wird nach jedem Durchlauf durch den Beschleuniger ein Lastausgleich hergestellt, d. h. Teilchen werden ggf. wieder neu auf die Slave-Knoten verteilt. Zudem besteht die Möglichkeit, die Aufteilung manuell an Hand der Leistungsfähigkeit der einzelnen Slave-Knoten vorzunehmen, um somit heterogene Systeme besser unterstützen zu können.

Die in [WB07] beschriebenen Ergebnisse zeigen für die mit PELEGANT durchgeführten Messungen mit 100.000 Teilchen bis 512 Prozessoren eine Effizienz von über 90%. Mit einer größeren Anzahl an Prozessoren fängt die Kommunikation an, die Berechnung zu dominieren und die Effizienz sinkt stark ab. Die Auswertungen ergaben außerdem eine deutlich gesteigerte Effizienz bei der Simulation mit 1 Mio. Teilchen im Gegensatz zu weniger Teilchen [WB07, S. 3446]. In der Veröffentlichung wird deshalb der Schluß gezogen, die vorhandenen parallelen Ausführungseinheiten möglichst gut auszulasten bzw. ihre Anzahl an die der zu simulierenden Teilchen anzupassen. Interessant ist darüber hinaus der Hinweis, dass Elemente in PELEGANT, die sich mit Einzelteilcheneffekten befassen, relativ gut zu parallelisieren waren, während kollektive Effekte durch ihren deutlich höhren Kommunikationsbedarf schwieriger zu parallelisieren waren. Dies wird auch im Rahmen der vorliegenden Diplomarbeit zu sehen sein. Bzgl. des Kommunikationsbedarfs zwischen den einzelnen Knoten im Zusammenhang mit MPI wird in [WB07, S. 3445] der Hinweis gegeben, nach Möglichkeit Teilergebnisse auf den einzelnen Berechnungsknoten zu bilden und nur diese zu kommunizieren. Dem Performanzproblem durch eine zentrale Ausgabe von Werten durch den Master umgeht man in PELEGANT inzwischen, indem paralleles I/O eingesetzt wird und auch Slaves für Ausgaben zuständig sind [vgl. BSS+09].

4.1.2 Parallelisierung mit GPUs

Im Rahmen eines in den USA staatlich geförderten Forschungsprojekts mit dem Namen „Accelerating Large-Scale Beam Dynamics Simulations with GPUs“ wird seit 2011 untersucht, wie Simulationen in ELEGANT mit Hilfe von Grafikprozessoren unterstützt werden können [vgl. Dep11]. Das Projekt wird von der Firma Tech-X Corporation durchgeführt, das Ergebnis soll ebenso wie ELEGANT unter einer Open Source Lizenz veröffentlicht werden [vgl. Amy]. Das langfristige Ziel im Rahmen des Projekts ist es, eine Bibliothek mit optimierten Kerneln zur Verfügung zu stellen und darauf aufbauend sukzessive die Elemente in ELEGANT auf eine Grafikkartenprogrammierung umzustellen [vgl. RPMA10, S. 287].

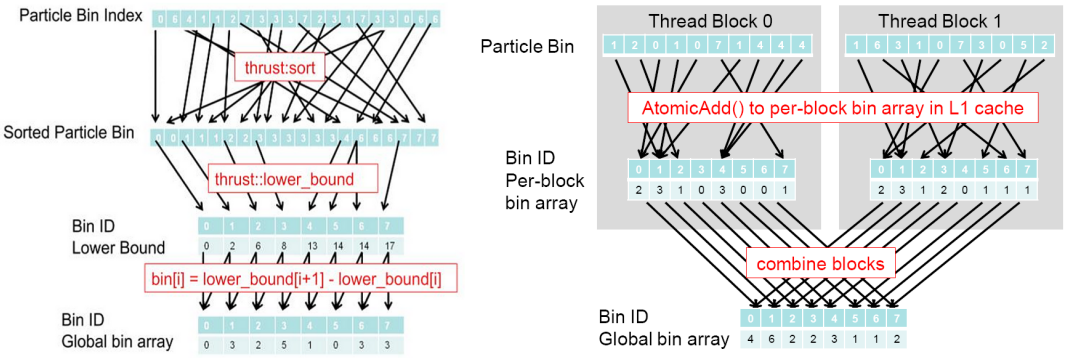

Abbildung 4.1: Interpolation der Teilchen auf die Gitter im Programm ELEGANT. Dargestellt sind eine Thrust-Variante mit einer Vorsortierung der Teilchen und eine CUDA-Variante mit atomaren Operationen (Quelle:[ABP+12a] und [ABP+12b]).

In [RPMA10], dem grundlegenden Papier zur Grafikkartenunterstützung in ELEGANT wird die schrittweise Strategie zur Integration in das bestehende Simulationsprogramm beschrieben: in einem ersten Schritt werden die Teilchen für jedes Element auf die Grafikkarte und wieder zurück kopiert, in einem zweiten Schritt sollen die Teilchen länger auf der Grafikkarte gehalten werden und das finale Ziel in einem dritten Schritt sollte sein, alle Informationen auf der Grafikkarte zur Verfügung zu stellen und mit einem einzigen Aufruf alle Berechnungen auf der Grafikkarte durchzuführen. Was letztendlich die GPU-Version von ELEGANT tatsächlich umsetzen wird, bleibt zunächst abzuwarten, den aktuellen Veröffentlichung nach ist momentan der erste sowie teilweise der zweite Schritt umgesetzt.

In [ABP+12b] werden verschiedene, bereits auf die GPU portierte Elemente vorgestellt. Für die Implementierung des Quadrupols und Dipols konnte dabei jeweils eine Beschleunigung von 100x bzw. 90x gegenüber einer vergleichbaren CPU-Implementierung erreicht werden (reine Berechnungszeit). Die beste Performanz wurde erzielt, wenn die Zwischenwerte in Registern gespeichert wurden und die Matrizen im konstanten Speicher. Eine etwas schlechtere Performanz wurde ermittelt, wenn für die Matrizen nicht der konstante Speicher sondern der globale Speicher mit dem zusätzlichen Schlüsselwort `const` genutzt wurde. Allerdings wurden bei der Implementierung teilweise spezielle Optimierungen vorgenommen. So wurde für die Implementierung des Quadrupols die Konfiguration der Registeranzahl der GPU verändert, um zu erreichen, dass alle Zwischenergebnisse in Registern gehalten werden können und nicht bis in den globalen Speicher der GPU ausgelagert werden.

Speziell im Rahmen dieser Arbeit von Interesse ist auch die Implementierung des Elements zur Berechnung der longitudinalen Raumladung. Für den zeitaufwändigen Schritt der Interpolation der Teilchen auf das Gitter werden in [ABP+12a], [ABP+12b] und [ABP+13] vier verschiedene Implementierungen verglichen. Generell wurde der

Ansatz gewählt, die Teilchen bereits im gemeinsamen Speicher auf das Gitter zu interpolieren. Dabei wurden die Teilchen vorher entweder nach ihrer Position sortiert, oder nicht. Beim zunächst Sortieren der Teilchen und dann Interpolieren auf dem Gitter konnte eine Beschleunigung von 3.4x gegenüber der seriellen Version erreicht werden. Eine Nutzung von Thrust Funktionen zum Sortieren (`thrust::sort` und `thrust::lower_bounds`, siehe auch Abb. 4.1) brachte bereits eine Beschleunigung von 5.3x [vgl. ABP+12b, S. 344]. Werden die Teilchen nicht vorsortiert, erhöht sich zwar die Zeit für den dann irregulären Speicherzugriff, jedoch entfällt der Sortieraufwand. Ohne Vorsortieren der Teilchen und mit direktem Aktualisieren der Gitterinformationen im gemeinsamen Speicher durch atomare Operationen (siehe Abb. 4.1) konnte eine Beschleunigung von 10x erreicht werden. Eine weitere Vorstufe mit mehreren Gittern im gemeinsamen Speicher, die zunächst aktualisiert werden und dann zu einem Gitter zusammengefasst werden, konnte den Speicherzugriff weiter entzerren und führte zu einer Beschleunigung von 23x [vgl. ABP+13].

Neben der Unterstützung von GPUs in ELEGANT ist ein weiterer Schwerpunkt des Projekts der Ausbau der kommerziell von der Firma Tech-X Corporation vertriebenen GPU-Bibliothek GPULib zur Unterstützung der GPU-Implementierung. Die Idee hierbei ist – ähnlich wie bei Thrust – die eigentliche Komplexität der Grafikkartenprogrammierung vor dem Entwickler zu verstecken, aber dennoch einen einfachen Zugang zu den Ressourcen der Grafikkarte zu ermöglichen [vgl. RPMA10]. GPULib bietet dabei eine Zusammenstellung an Bibliotheksfunktionen speziell für den Einsatz in Teilchensimulationen, es bietet Funktionalitäten für Vektoroperationen, Algorithmen der Linearen Algebra, FFT, Zufallszahlengenerator u. a., weitere Informationen finden sich bei [MMG08]. Diese Bibliothek wird im Rahmen der GPU-Entwicklung für ELEGANT eingesetzt, aber auch separat entwickelt und vertrieben. Ohne näher auf GPULib einzugehen ist hier der Ansatz deutlich sichtbar, dass sich die eigentlichen Entwickler von Teilchensimulationsprogrammen möglichst wenig mit der Komplexität der Grafikkartenprogrammierung auseinandersetzen sollen. Die Idee ist, ihnen einen einfachen Zugang zu den Ressourcen der Grafikkarte zu bieten. Die Grafikkarte selbst wird dem Aufrufer als einfacher Vektorprozessor dargestellt, eine Abstraktionsschicht in GPULib ermittelt intern, ob überhaupt eine Grafikkarte zur Verfügung steht und emuliert sonst deren Verhalten transparent für den Aufrufer. GPULib bietet zusätzlich eine Schnittstelle für die Scriptsprachen IDL, Matlab und Python, um eine universelle Einsetzbarkeit in Simulationen zu ermöglichen.

4.1.3 Bewertung

Da bei ELEGANT ebenfalls auf einem vorhandenen Programm aufgesetzt wird, sind allgemein die Erkenntnisse und Erfahrungen relevant für die vorliegende Arbeit.

Im Zusammenhang mit MPI ist der Hinweis auf die nötige sehr hohe Teilchenzahl (1 Million Teilchen) für eine entsprechende Effizienz durchaus relevant. Zu MPI werden zwei Kernaussagen getroffen: einerseits soll die Anzahl der MPI-Knoten an die tatsächliche Anzahl der Teilchen angepasst werden, um eine hohe Effizienz zu erreichen.

Zum anderen sollen Teilsummen möglichst lokal gebildet und nur diese verschickt werden, um den Kommunikationsaufwand zu minimieren. Beiden Aussagen kann uneingeschränkt zugestimmt werden. Speziell vor dem Hintergrund, dass einzelne Berechnungsknoten bei Einbeziehung der GPU für deren Auslastung bereits viele tausend Teilchen bearbeiten sollten, gewinnt die Aussage der Anpassung der Anzahl der MPI-Knoten an die dann noch verbleibenden Aufteilungsmöglichkeiten noch einmal mehr an Bedeutung.

Im Zusammenhang mit der GPU erscheint der Ansatz der schrittweisen Portierung vorhandener Elemente vor allem für die Wartbarkeit sehr gut, obwohl so die Teilchen oft umkopiert werden müssen. Das Problem der damit verbundenen Laufzeiteinbußen wird in [RPMA10, S. 288] besprochen und als zukünftiges Ziel aufgezeigt, dass die Teilchen möglichst lang im Speicher der Grafikkarte gehalten werden sollen. Obwohl längere Laufzeiten zu erwarten sind, kann generell der Ansatz einer schrittweisen Einbeziehung der GPU z. B. für einzelne zeitintensive Berechnungsschritte so durchaus als erster Ansatz übernommen werden, vor allem, wenn auf bereits bestehenden Programmen aufgesetzt wird.

Speziell für den Einsatz von GPUs interessant ist hierbei die Tatsache, dass sich bei den mit GPULib durchgeführten Simulationen der Einsatz von Grafikkarten ab einer Anzahl von mehr als 10.000 Teilchen gelohnt hat. Die Aussage, dass zur Beschleunigung eine Datenumstrukturierung als Basis für eine bessere Vektorverarbeitung und ein Eliminieren von Verzweigungen im Kontrollfluss nötig waren, darf als allgemeingültig bei der Parallelisierung angesehen werden. Bzgl. der Berechnung der kollektiven Effekte erscheinen bei der Interpolation der Teilchen auf das Gitter mit den aktuellen Grafikkartentypen Ansätze ohne Vorsortierung lohnenswerter. Dies kann als Hinweis für die eigene Implementierung dienen. Die Aussage, dass die in den Simulationen wichtigen kollektiven Effekte generell schwieriger zu parallelisieren waren wird auch im Rahmen dieser Diplomarbeit zu sehen sein.

Die in ELEGANT verwirklichten Ansätze zur Einbeziehung der GPU können positiv und grundsätzlich auch als allgemeingültig angesehen werden. Sie zeigen deutliche Laufzeitverbesserungen gegenüber der seriellen Variante, wobei bei den Messungen häufig nur einzelne Berechnungszeiten betrachtet werden und nicht die Gesamtlaufzeit des Programms verglichen wird. Im Detail sind die angewandten Optimierungen jedoch teilweise sehr hardwarespezifisch, bedingt aber vermutlich auch durch die kommerziell vertriebene, sehr stark optimierte Hilfsbibliothek. Auch bei ähnlichen Berechnungen wurden für verschiedene Elemente oft unterschiedliche Optimierungen angewandt, um möglichst hoch effiziente Kernelimplementierungen für einzelne Elemente zu erzielen. Dies kann so für die eigenen Implementierungen im Rahmen dieser Arbeit nicht empfohlen werden, da hier eher das Einbeziehen der GPU möglichst unter Beibehaltung der bestehenden Strukturen, einer leichten Wartbarkeit und einer guten Portierbarkeit auf zukünftige Grafikkartentypen im Vordergrund steht.

4.2 Einsatz von GPUs im Simulationsprogramm PIConGPU

Das Teilchensimulationsprogramm PIConGPU wird vom Institut für Strahlenphysik des Helmholtz-Zentrums Dresden-Rossendorf in Zusammenarbeit mit dem Zentrum für Informationsdienste und Hochleistungsrechnen der Technischen Universität Dresden entwickelt. Das Programm dient der Simulation von Plasmen. U. a. kann damit die Teilchenbeschleunigung simuliert werden, die auftritt, wenn mit Hilfe von Lasern eine geladene Welle in einem Plasma erzeugt wird. Das Besondere bei PIConGPU ist, dass es ein relativ neues Simulationsprogramm ist, bei dem bereits im Entwurf eine verteilte Berechnung per Nachrichtenaustausch und lokal vorhandene GPUs einbezogen wurden. Dadurch ist eine besonders gute Unterstützung der parallelen Abarbeitung zu erwarten. Das Programm selbst wird seit 2009 entwickelt und steht heute für verschiedene GPU-Generationen zur Verfügung. PIConGPU setzt dabei den bekannten PIC-Berechnungsablauf (vgl. Abb. 2.2) um und bietet einen vollen 3D-PIC-Code (auch 2D ist möglich). Weitere Informationen zu PIConGPU finden sich über die Projektwebseite des Instituts [Hel13].

4.2.1 Parallelisierung mit MPI

Bei Laser-Plasma-Simulationen ist der betrachtete Simulationsbereich typischerweise groß, was sich entsprechend in einer hohen Anzahl an Gitterpunkten und Teilchen niederschlägt. Auch für die mit PIConGPU durchgeführten Simulationen übersteigt die Anzahl der betrachteten Teilchen die Kapazität einzelner Rechner mit maximal vier eingebauten GPUs bei weitem [vgl. BWH+10, S. 2834]. Auf der Projektwebseite sind z. B. die Ergebnisse einer normalen Simulation mit 800 Millionen Teilchen auf 16 GPUs dargestellt [vgl. Hel13]. Aus diesem Grund müssen Laser-Plasma-Simulationen Möglichkeiten zum verteilten Rechnen bieten. Bei PIConGPU übernimmt die CPU ausschließlich die Aufgabe der Kommunikation. Sie ist für den Nachrichtenaustausch mit anderen Knoten mittels MPI und den Datenaustausch mit der lokalen GPU zuständig. Der Parallelisierung ist hierbei eine Datenzerlegung der Gitter und Teilchen zu Grunde gelegt.

Aufgeteilt werden die Gitterinformationen dabei in einen „Kern“ an Gitterzellen, der im Rahmen der Simulation rein lokal betrachtet werden kann und sog. „Schließzellen“ (engl. guarding cells), die Daten enthalten, die mit benachbarten Berechnungsknoten (oder anderen GPUs des eigenen Knotens) ausgetauscht werden müssen. Man macht sich dabei zu Nutze, dass die Berechnungen auf dem Kern unabhängig sind von anderen Knoten und nur Werte bzgl. der Schließzellen ausgetauscht werden müssen. Dies kann zu einem passenden Zeitpunkt in der Berechnung geschehen. Dabei wird versucht, Kommunikation und Berechnung soweit möglich zu überlappen. Ein Ansatz ist, die MPI-Kommunikation sogar in einen eigenen Thread auszulagern, um parallel zu der Kommunikation mit anderen Knoten Daten mit der lokalen GPU austauschen zu können. Um u. a. die Implementierungsdetails dieser erweiterten MPI-Kommunikation vor dem Anwender zu verbergen, wurde eine umfangreiche Biblio-

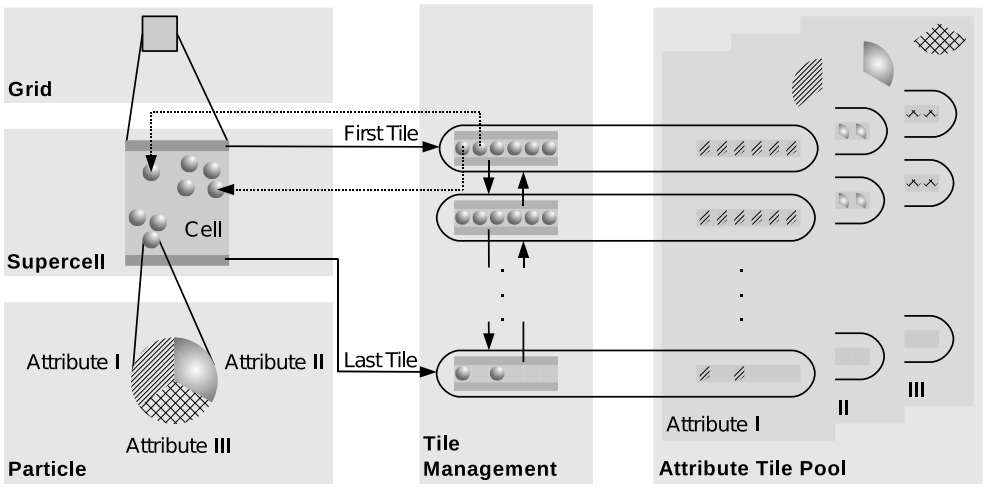

Abbildung 4.2: Datenzerlegung im Programm PIConGPU: Die Gitterzellen werden in Superzellen zerlegt. Pro Superzelle werden die enthaltenen Teilchen zusammen mit ihren Attributen wie den Koordinaten in sog. Kacheln (tiles bzw. frames) gespeichert (Quelle: [HSW+10, Abb. 1]).

thek entwickelt. Eigene Threads werden darüber hinaus auch für andere nebenläufige Aufgaben verwendet, so z. B. auch für die Ausgabe von Daten während der Simulation.

4.2.2 Parallelisierung mit GPUs

Für die Programmierung der lokalen GPUs wurde aus Performanzgründen ebenfalls CUDA eingesetzt. Pro Iterationsschritt wurde eine um 40% verbesserte Laufzeit gegenüber einer Implementierung mit OpenCL gemessen [vgl. JB10, S. 39]. Beim Einbezug der GPU wurde darauf geachtet, die Datenstrukturen möglichst gut an die Berechnungen im Rahmen der Simulation anzupassen. Die Grundidee ist, über effiziente Datenstrukturen zu effizienten Algorithmen zu gelangen.

Die Gitter werden im Rahmen einer Datenzerlegung aufgeteilt. Die Gitterzellen werden dabei zu sog. Superzellen zusammengefasst, diese haben typischerweise die Größe 16x16, um sie später mit jeweils einem Thread auf der GPU bearbeiten zu können (hierbei ist die Anzahl Threads pro Block genau auf 256 gesetzt). Teilchen, die sich in einer solchen Superzelle befinden, werden mit jeweils einem Thread pro Teilchen bearbeitet. Die Teilchen sind dabei in Kacheln zu jeweils 256 Stück zusammengefasst, eine Kachel wird von einem Thread-Block bearbeitet. Ggf. sind mehrere Kacheln und damit Blöcke nötig, falls sich mehr als 256 Teilchen in der Superzelle befinden. Wenn die Teilchenzahl in der Superzelle kein Vielfaches von 256 ist, führt dies zu Divergenz der Threads; die restlichen Threads tun dann nichts. Zu jedem Teilchen werden Attribute wie die Koordinaten und zusätzlich der Index der Gitterzelle, in der es sich befindet, gespeichert. Die Kacheln mit den Teilchen werden als doppelt verkettete Liste verwal-

tet, um die Kacheln leicht durchlaufen zu können und um Teilchen möglichst leicht aufnehmen oder abgeben zu können, wenn diese zwischen den Superzellen wandern.

Das Auslesen der Felder auf dem Gitter, Berechnen der Kraft auf die einzelnen Teilchen und das Aktualisieren der Teilchenkoordinaten (linker Schritt in Abb. 2.2) übernehmen dieselben Threads. Dazu wird jeweils pro Thread-Block zuerst die Gitterinformation der bearbeiteten Superzelle vom globalen in den lokalen Speicher eingelesen, danach werden mit denselben Threads genau die Kräfte auf die Teilchen einer Teilchenkachel berechnet und dann die Teilchenkoordinaten aktualisiert. Das ist möglich, weil die Superzelle in 256 Zellen aufgeteilt ist und auch die Teilchen in Kacheln zu 256 Stück zusammengefasst sind.

Laut aktuellen Veröffentlichungen konnte im Gegensatz zu früheren Implementierungen, bei denen die einzelnen Teilchen als verkettete Liste abgespeichert wurden, eine deutliche Beschleunigung durch das Zusammenfassen der Teilchen mittels der Kacheln (6x schneller) erreicht werden. Die bereits im Rahmen von MPI angesprochene Bibliothek übernimmt auch im Zusammenhang mit der GPU Aufgaben, die die technische Komplexität vor dem Anwender verbergen soll. Dazu zählen die Speicherallokierung auf Grund einer vorgegebenen Gitterkonfiguration, der Datentransport zwischen Host und Device und der Zugriff auf einzelne Datenelemente. Bei der Einbeziehung der GPU werden dabei auf einem Rechner mehrere GPUs unterstützt, was ebenfalls für den Anwender verborgen bleiben soll.

4.2.3 Bewertung

Einen direkten Vergleich mit einer CPU Implementierung mit identischer Funktionalität gibt es für PIConGPU nicht, da das Programm für eine verteilte Umgebung und lokal vorhandene GPUs entworfen wurde. Man kann jedoch davon ausgehen, dass die Struktur des Programms und der Daten die Parallelisierungsmöglichkeiten besonders effizient nutzt. Dadurch, dass das Programm für verschiedene GPU-Generationen zur Verfügung steht, ist ausserdem davon auszugehen, dass die durchgeführten Optimierungen genereller Natur sind und nicht nur die Spezifika einzelner GPUs ausnutzen.

Das Verstecken der Komplexität der hybriden MPI/GPU-Implementierung vor dem Anwender und das damit verbundene zur Verfügung stellen einfach zu handhabender Schnittstellen ist ein großer Vorteil, bedingt natürlich jedoch einen großen Entwicklungsaufwand. Generell sind aber die Verfahren zur Überlappung von Kommunikation und Berechnung und die dabei gemachten Erfahrungen interessant für die Integration von GPUs und MPI. Vor diesem Hintergrund ist auch die bei PIConGPU gewählte Datenzerlegung zu betrachten. Diese erscheint auf den ersten Blick ideal, da sowohl Gitter als auch Teilchen jeweils mit denselben Threads und Blöcken bearbeitet werden können. Sobald eine gleichmäßge Verteilung der Teilchen vorliegt, kann die Arbeit auch gleichmäßig auf die MPI-Knoten und GPUs verteilt werden. Konzentrieren sich die Teilchen jedoch in gewissen Zellen, kann dies dazu führen, dass die Arbeit sehr ungleichmäßig verteilt wird. Bereits in [vgl. JB10, S. 63] wird als noch offenes Problem die Lastverteilung angeführt. Hierauf ist allgemein ein Augenmerk zu richten:

nicht nur die lokalen Zugriffe und Berechnungen auf der GPU müssen effizient sein, bei der Einbeziehung verteilter Berechnungsknoten muss auch die Frage nach einer möglichst gleichmäßigen Verteilung der Arbeit in den Vordergrund rücken.

Interessant sind die Ansätze zur Bereitstellung möglichst optimierter Datenstrukturen. Dies ist im Rahmen der vorliegenden Diplomarbeit so nicht möglich, da auf bereits vorhandenen Daten- und Programmstrukturen aufgesetzt wird. Allerdings sollte die bessere Anpassung der Datenstrukturen an die GPU für zukünftige Neuentwicklungen berücksichtigt werden, sofern eine Einbeziehung der GPU geplant ist.

4.3 Einsatz von GPUs in anderen Simulationsprogrammen

Neben dem vorgestellten Einsatz von GPUs in den Programmen ELEGANT und PIConGPU gibt es in diesem Forschungsumfeld viele aktuelle Ansätze zur Parallelisierung von Teilchensimulationen. Im Folgenden werden beispielhaft kurz weitere Ansätze vorgestellt, die eher genereller Natur sind und somit eine Grundlage für die eigenen Überlegungen im praktischen Teil der Arbeit bilden. Begonnen wird mit einem Ansatz zur Parallelisierung des Teilchentransports mittels externer Kräfte, danach folgen Ansätze zur Parallelisierung der PIC-Simulation zur Betrachtung der Kräfte der Teilchen untereinander.

Zur Parallelisierung des Teilchentransports, also der Veränderung der Teilchenkoordinaten durch externe Kräfte repräsentiert durch Matrizen, bietet [ABHS08] ein gutes Beispiel. Betrachtet wird hierbei eine selbstentwickelte GPU-Version des Teilchentransports mit linearer Strahloptik im Vergleich zu dem bekannten (seriellen) Teilchensimulationsprogramm MAD (Methodical Accelerator Design). Gezeigt wird die allgemeine Anwendbarkeit der GPU für den Teilchentransport. Da bei der reinen Betrachtung externer Kräfte die Teilchen unabhängig voneinander fortbewegt werden können, konnte im Rahmen einer Simulation des Teilchentransports eine Beschleunigung von 4,3-fach erzielt werden, obwohl deutlich ältere Programmierschnittstellen und GPUs zum Einsatz kamen. Das GPU-Programm ist ab 10.000 Teilchen schneller als das serielle Vergleichsprogramm. Interessant ist der bereits hier beobachtete lineare Anstieg der GPU-Berechnungszeit bei Variation der Teilchenanzahl. Dieser wird auch im praktischen Teil der vorliegenden Arbeit zu beobachten sein.

Zur Parallelisierung der PIC-Simulation zur Einbeziehung der Raumladungskräfte werden in einem Grundlagenpapier [CC97] aus dem Jahr 1997 allgemeine Strategien dargestellt. Festgestellt wird, dass der PIC-Algorithmus schwierig effizient zu parallelisieren ist, einerseits bedingt durch Abhängigkeiten zwischen benachbarten Datenpunkten und andererseits, weil die Komplexität des Systems räumlich und auch zeitlich im Verlauf der Simulation variiert [vgl. CC97, S. 1378]. Die statischen Gitter und die dynamischen Teilchen stellen dabei jeweils andere Anforderungen an die Parallelisierung. Eine wesentliche Aussage ist, dass es keine Vorschrift gibt, wie man PIC-Simulationen parallelisiert, nur allgemeine Strategien, die je nach konkretem Anwendungsfall ausgewählt werden müssen. Bei der Wahl einzelner Optionen befin-

det man sich im Spannungsfeld zwischen Datenlokalität und Lastbalancierung [vgl. CC97, S. 1378]. Für die einzelnen Schritte der Simulation werden folgende Strategien aufgezeigt:

1. Interpolation der Teilchen auf die Gitter (Scatter): skaliert mit der Anzahl der Teilchen. Ausgegangen wird von einer Aufteilung sowohl der Gitter als auch der Teilchen auf die Prozessoren. Liegen räumlich benachbarte Teilchen auf demselben Prozessor, kann zunächst für die lokalen Gitterpunkte die Verteilung interpoliert werden. Danach müssen Informationen für Gitterpunkte am Rand, die eigentlich von anderen Prozessoren verarbeitet werden, an diese kommuniziert werden. Für eine Datenlokalität ist eine möglichst hohe räumliche Übereinstimmung von Gitterpunkten und Teilchen pro Prozessor wünschenswert. Dies führt aber zu einer negativen Lastbalancierung, wenn die Teilchen sich an bestimmten Stellen im Raum konzentrieren, oder zu erhöhtem Kommunikationsaufwand, wenn man zur Aufteilung verschieden große Regionen verwendet.

2. Bestimmung des Raumladungsfelds (Field Solver): skaliert mit der Anzahl der Gitterpunkte. Zur Lastbalancierung sollte die Aufteilung der Gitterpunkte auf Prozessoren möglichst regelmäßig sein.

3. Interpolation der Felder auf dem Gitter zurück auf die Teilchen (Gather): skaliert mit der Anzahl der Teilchen, Betrachtung identisch zu Schritt 1.

4. Ermittlung der neuen Teilchenpositionen (Particle Push): skaliert mit der Anzahl der Teilchen. Zur Lastbalancierung sollten die Teilchen gleichmäßig auf die Prozessoren verteilt sein. Für eine Datenlokalität sollten hingegen räumlich benachbarte Teilchen auf demselben Prozessor bearbeitet werden, da dann ein effizienter Zugriff auf dieselben Gitterinformationen in den Schritten 1 und 3 (Scatter/Gather) erzielt werden kann. Allerdings entsteht so erhöhter Aufwand bei der Zuordnung und ggf. Neuzuordnung der Teilchen.

Ausgehend von den Betrachtungen der Einzelschritte wird in [CC97] deutlich, dass die Wahl der optimalen Datenzerlegung in den Schritten 1 und 3 (möglichst gute Übereinstimmung der Teilchen und Gitterpunkte pro Prozessor) konträr ist zu der optimalen Datenzerlegung in den Schritten 2 und 4 (Gitterpunkte und Teilchen möglichst gleichmäßgig auf alle Prozessoren verteilt). Es wird darüber hinaus das Problem thematisiert, dass selbst eine einmal gefundene gute Zerlegung auf Grund des dynamischen Charakters der Simulation nach ein paar Zeitschritten wieder obsolet ist [vgl. CC97, S. 1388]. Die in [CC97] beschriebenen Parallelisierungsstrategien und Spannungsfelder bei der Auswahl der geeigneten Strategie für das eigene Problem bieten eine gute Grundlage für eigene Überlegungen im Rahmen des praktischen Teils.

Zur Parallelisierung der PIC-Simulation speziell unter Einbeziehung von MPI wird das Problem der Verteilung der Teilchen und Gitter auf die einzelnen Berechnungsknoten in [WGW06] abstrahiert und allgemein als Optimierungsproblem beschrieben. Aus der Lösung des Problems werden Vorschläge für eine voneinander abhängige oder unabhängige Verteilung von Teilchen und Gitterpunkten auf die Prozessoren abgeleitet. Ein

solcher Ansatz sollte bei grundlegenden Entscheidungen für die Datenverteilung bei Einsatz von MPI herangezogen werden.

Zur Parallelisierung der PIC-Simulation speziell unter Einbeziehung der GPU bietet [DS11] ein gutes Beispiel. Ausgehend von der Feststellung, dass typischerweise die Schritte zur Interpolation der Teilchen auf die Gitter und umgekehrt die meiste Laufzeit in Anspruch nehmen, wird gefolgert, dass speziell diese Schritte optimal parallelisiert werden müssen. Deshalb wird bei diesem Ansatz zur Erreichung von Datenlokalität bei der Interpolation eine Sortierung der Teilchen gewählt [vgl. DS11, S. 642]. Dabei werden Überlegungen angestellt, wie Teilchen zwischen Threads und Thread-Blöcken wandern können, und jeweils Implementierungen vorgeschlagen. Die schnellste Laufzeit wurde erzielt, wenn tatsächlich bei jedem Durchlauf des Simulationszyklus die Teilchen neu sortiert wurden [vgl. DS11, S. 648].

Aufbauend darauf implementieren [KHRD11] eine bereits bei [SDG08] vorgeschlagene performante Sortierung für die Teilchen auf der GPU, die nur die sich wegbewegenden Teilchen neu einsortiert. Vorgeschlagen wird eine Zuordnung von einem Teilchen zu jeweils einem Thread, um die GPU möglichst gut auszulasten. Eine Sortierung der Teilchen ist hier auch deshalb nötig, weil die Gitter nicht vollständig in den gemeinsamen Speicher passen. So wird es möglich, dass alle Teilchen eines Thread-Blocks nur den Teil des Gitters aktualisieren, der tatsächlich in den gemeinsamen Speicher passt.

[AFPS11] gehen bei der Einbeziehung der GPU genau den umgekehrten Weg: Teilchen werden auf die Gitter mit Hilfe atomarer Operationen interpoliert. Um dabei die mit den Schreibkonflikten verbundenen Performanzeinbußen zu minimieren, werden die Teilchen möglichst gut durchmischt (wobei wiederum ein Sortieralgorithmus genutzt wird, um den Teilchenindex zu ermitteln, bevor gemischt wird). So wird erreicht, dass nebeneinander liegende Threads, die jeweils ein anderes Teilchen bearbeiten, sich möglichst beim Aktualisieren der Gitter nicht in den Weg kommen.

Die in den vorangegangenen Ausführungen vorgestellten Veröffentlichungen bieten einen Einblick in Untersuchungen zum vorliegenden Thema. Für die Einbeziehung externer Kräfte in Form von Matrizen in die Simulation gibt es bereits GPU-Implementierungen, die einen Laufzeitvorteil gegenüber der seriellen Variante bieten. Es wird aber deutlich, dass die Einbeziehung der GPU zur Darstellung von Raumladungskräften in Form von PIC-Simulationen ein offenes Forschungsgebiet ist. Zur Parallelisierung des PIC-Algorithmus gibt es eine Reihe Ansätze, die versuchen, je nach konkretem Problem Strategien für die Parallelisierung vorzuschlagen. Die dargestellten – teilweise gegensätzlichen – Strategien dienen als Grundlage für die Überlegungen im praktischen Teil dieser Arbeit.

5 Modifikation des Particle-in-cell Codes

Im praktischen Teil wird die im Rahmen der Arbeit mittels GPU-Programmierung realisierte Parallelisierung der bestehenden Programme PATRIC und LOBO beschrieben.

Dazu wird als erstes kurz die genutzte Entwicklungs- und Laufzeitumgebung vorgestellt. Als Basis für die Überlegungen zur Parallelisierung dienen Messungen mit der Originalversion von PATRIC, zunächst unter Beibehaltung der verteilten Berechnung mittels MPI. Hierbei wird jedoch zu erkennen sein, dass diese Messungen auf einem einzelnen Rechner wenig aussagekräftig sind. Stattdessen führt das genutzte MPI-Framework in Zusammenspiel mit dem lokalen Betriebssystem zu Aussagen, die so nicht auf ein verteiltes System übertragbar sind. Aus diesem Grund wird der Aspekt der verteilten Berechnung komplett ausgeklammert und erst am Ende des praktischen Teils in Form eines Ausblicks wieder aufgegriffen.

Zur Vorbereitung der eigentlichen Parallelisierung werden danach allgemeine Vorüberlegungen zur Einbeziehung der GPU angestellt. Hierbei wird einerseits auf die genutzten Datenstrukturen eingegangen und andererseits die Frage erörtert, welche Speicher auf der GPU für eine Speicherung der statischen und dynamischen Simulationsdaten in Frage kommen.

Nach diesen Vorüberlegungen werden verschiedene Ansätze zur Parallelisierung umgesetzt. Mit Hilfe des Programms PATRIC wird sich dabei zunächst auf den reinen Teilchentransport, d. h. die transversale Teilchenbewegung, beschränkt. Hierbei wird eine Herangehensweise gewählt, bei der große Teile des Originalprogramms bestehen bleiben können und nur einzelne Aspekte auf die GPU portiert werden. Zunächst wird dazu der einzelne Transportschritt auf die GPU gebracht. Um möglichst viele Berechnungen mit den Daten durchführen zu können, werden darauf aufbauend die Teilchen über mehrere Transportschritte hinweg auf der GPU belassen. Hierbei kann im Vergleich zum Originalprogramm eine Beschleunigung erzielt werden. Da aber die Teilchen für Ausgaben zurück auf den Host kopiert werden müssen, wird der Fragestellung nachgegangen, wie die Ausgabe von Zwischenergebnissen ohne zu große Laufzeiteinbußen durchgeführt werden kann. Dazu wird untersucht, inwiefern es möglich ist, Zwischenergebnisse direkt auf der GPU zu berechnen und nur diese statt der kompletten Teilchendaten zum Host zurück zu kopieren. Hierbei wird jedoch bereits die Problematik deutlich, dass auf der Grafikkarte Reduzieroperationen über alle Teilchen hohe Laufzeiten aufweisen. Anhand der GPU-Implementierung des Teilchentransports werden im Anschluss Nebenthemen wie der Vergleich der Laufzeit CUDA gegenüber Thrust für die Strahlgrößenberechnung und einfache gegenüber doppelter Genauigkeit zur Speicherung der Teilchenkoordinaten behandelt.

Mit Hilfe des Programms LOBO wird danach die longitudinale Teilchenbewegung betrachtet; speziell liegt hierbei der Fokus auf den kollektiven Effekten. Im Gegensatz zu den vorangegangen Modifikationen wird bei LOBO der Ansatz verfolgt, zentrale Teile des Programms auf die GPU zu portieren. Die im Rahmen der PIC-Simulation auftretenden Schritte werden dabei einzeln betrachtet. Da die Interpolation der Teilchen auf das Gitter und umgekehrt die meiste Zeit in Anspruch nimmt, konzentrieren sich die Untersuchungen auf diese Schritte. Zwei unterschiedliche Varianten zur Parallelisierung werden dabei verfolgt und miteinander verglichen.

Aufbauend auf den Ergebnissen der Arbeit wird anschließend eine Überlegung zur Integration der vorgeschlagenen Parallelisierungen im Zusammenhang mit der Betrachtung der gesamten Teilchenbewegung angestellt. Ein Ausblick zur Einbeziehung einer verteilte Berechnung mittels MPI beschließt den praktischen Teil der Arbeit.

5.1 Entwicklungsumgebung

Für die Entwicklung im Rahmen der Diplomarbeit kam ein Desktopsystem mit einem 2.67 GHz Intel Xeon X5650 Prozessor mit sechs Kernen und einer NVIDIA Tesla C2075 Grafikkarte zum Einsatz. Weitere Informationen zur genutzten Grafikkarte sind in Tab. 5.1 aufgelistet. Die Entwicklung wurde mit NVIDIA CUDA Version 5.0 unter Ubuntu Linux 12.04 durchgeführt, als Entwicklungsumgebung kamen Eclipse und nsight, die NVIDIA eigene Eclipse Version, zum Einsatz.

Für spezielle Fragestellungen stand zudem eine einfachere NVIDIA Quadro 1000M Grafikkarte mit nur zwei Multiprozessoren (statt 14) auf einem Laptopsystem mit einem 2.50 GHz Intel Core i5-2520M Prozessor mit zwei physikalischen bzw. vier logischen Kernen zur Verfügung. Die Entwicklungs- und Laufzeitumgebung waren ebenfalls die oben genannten.

Tabelle 5.1: Spezifikation der verwendeten Grafikkarte

	Tesla C2075
CUDA-Berechnungsfähigkeit:	2.0
Globaler Speicher:	5375 MBytes
CUDA-Kerne:	448 Kerne (14 Multiprozessoren x 32 Kerne)
GPU-Taktrate:	1.15 GHz
L2-Cache Größe:	786432 bytes
Konstanter Speicher:	65536 bytes
Gemeinsamer Speicher pro Block:	49152 bytes
Register pro Block:	32768
Größe eines Warps:	32 Threads
Max. Anzahl Threads pro Multiprozessor:	1536
Max. Anzahl Threads pro Block:	1024

5.2 Unverändertes PATRIC Programm

Als Grundlage für spätere Überlegungen zur Parallelisierung wurden zunächst Messungen mit dem Originalprogramm durchgeführt. Hierbei stand vor allem die Frage im Vordergrund, ob das Programm in unveränderter Form für die Einbeziehung der GPU verwendet werden kann.

5.2.1 Messungen mit MPI

Bei den Messungen wurde eine konstant fokussierende Optik mit 16 Transportmatrizen und 100.000 Teilchen gewählt. Gemessen wurde auf dem Desktopsystem; als Vergleich mit einem langsameren Rechner diente das Laptopsystem. Die Anzahl der MPI-Knoten wurde im Bereich von 1 bis 8 variiert, was der tatsächlichen Nutzung entspricht. Für die Zeitmessung kam der Unix-Befehl `clock_gettime()` zur Ermittlung Prozessorzeit und `MPI_Wtime()` zur Ermittung der Ausführungszeit – also der tatsächlich verstrichenen Zeit (wallclock time) – zum Einsatz. Für alle im Folgenden dargestellten Messungen wurden jeweils Mittelwerte über 20 Läufe gebildet. Da sich bei den Ausführungen nur geringe Schwankungen um wenige Zehntelsekunden ergaben, sind bei den Messwerten keine Fehlerbalken eingezeichnet. Abb. 5.1 und 5.2 zeigen die Ergebnisse der durchgeführten Messungen mit dem Originalprogramm.

Wie in Abb. 5.1 zu sehen, nimmt die Ausführungszeit unter Verwendung einer höheren Anzahl an MPI-Knoten zunächst ab, dann jedoch wieder zu. Wie beim Desktopsystem zu erkennen ist, nimmt die Zeit mit der Anzahl der verfügbaren Prozessorkerne stark ab. Dass dies jedoch nicht bis zu den vollen sechs physikalischen Kernen möglich ist, kann nur so erklärt werden, dass vom Betriebssystem nicht alle sechs Kerne zur Verfügung gestellt wurden. Beim Laptopsystem nimmt die Zeit mit der Anzahl der zwei physikalischen Prozessorkerne stark ab. Mit den zwei weiteren logischen Kernen ist darüber hinaus nur noch eine leichte Abnahme zu beobachten. Danach steigt die Zeit zunächst bei beiden Systemen wieder an.

Beim Aufruf des MPI-Programms wird pro angegebenem Knoten ein eigener Prozess gestartet, welcher vom Betriebssystem jeweils einem Prozessorkern zugewiesen werden kann. Nutzt man eine größere Anzahl MPI-Knoten als verfügbare Prozessorkerne, steigt die Ausführungszeit an. Im Fall eines Einzelplatzsystems ist es also für die Gesamtausführungszeit günstig, genau so viele MPI-Knoten wie verfügbare Prozessorkerne zu wählen. Dies setzt voraus, dass diese auch zur Verfügung stehen und dass das Betriebssystem eine entsprechende Aufteilung auf freie Prozessorkerne vornimmt. Da der Aufruf des Simulationsprogramms über ein Pythonscript erfolgt, könnte beim Aufruf als Argument die per `grep -c processor /proc/cpuinfo` ermittelte Anzahl an Prozessorkernen mitgegeben werden.

Allerdings zeigt sich in Abb. 5.2, dass mit steigender Anzahl MPI-Knoten die Prozessorzeit deutlich anwächst. Das Programm ist so gestaltet, dass bei einer größeren Anzahl MPI-Knoten der einzelne Knoten weniger Daten bearbeitet. Aus diesem Grund nimmt

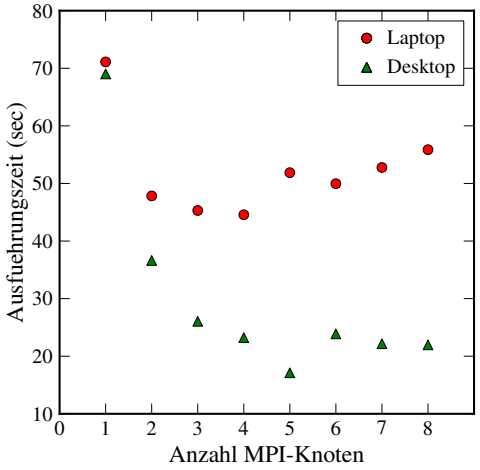

Abbildung 5.1: Ausführungszeit des Originalprogramms PATRIC (wallclock time).

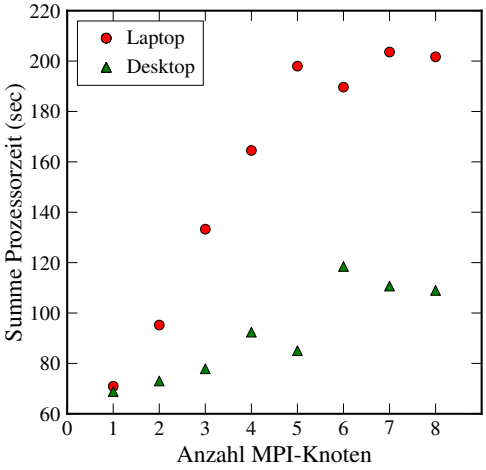

Abbildung 5.2: Prozessorzeit des Originalprogramms PATRIC als Summe über alle MPI-Knoten.

die Kommunikation der einzelnen Knoten untereinander zu, was zusätzlich Prozessorzeit kostet. Dennoch könnte man annehmen, dass dies die Prozessorzeit in geringerem Maße ansteigen lässt, als in der Messung zu sehen. Da die Abarbeitung auf einem Rechner stattfindet, sind nur geringe Kommunikationszeiten zu erwarten. Darüber hinaus fällt auf, dass der Anstieg der Prozessorzeit bis zur Anzahl der verfügbaren Prozessorkerne gleichmäßiger erfolgt, danach jedoch schwankt. Eine zentrale Rolle spielt das verwendete MPI-Framework MPICH und dessen Umgang mit der Situation, dass mehrere MPI-Knoten auf *einem* Rechner gestartet werden. Wie auf den Webseiten des Herstellers [MPI12] zu lesen, steigen gerade im Fall einer sog. „oversubscription“ (d. h. mehrere MPI-Prozesse werden auf demselben Prozessorkern eingeplant) die Prozessorzeiten stark an, da im Fall einer Kommunikation aktiv gewartet wird (busy waiting). Dies würde die stark ansteigende Prozessorzeit erklären. Der aktuell laufende Prozess bekommt so keine Antwort vom nächsten MPI-Prozess, der seinerseits auf das Freiwerden des Prozessorkerns wartet. Hierbei spielt das Betriebssystem eine wichtige Rolle, da es vorgibt, wie lange Prozesse warten, bis sie suspendiert werden. Dazu existieren Vorhersagemechanismen, die es erlauben, aus Vergangenheitswerten Einplanungsstrategien für zukünftige Prozessausführungen abzuleiten und somit auch Wartezeiten für die nächste Ausführung festzulegen [vgl. Tan09, S. 631ff.].

Da jedoch die Interna des verwendeten MPI-Frameworks und das Scheduling des Linux Betriebssystems an dieser Stelle nicht näher beleuchtet werden können und aus diesem Grund die Höhe des Anstiegs der Prozessorzeit im Fall eines Einzelplatzsystems nicht zuverlässig bestimmt werden kann, wird vorgeschlagen, die folgenden Messungen auf den Fall mit einem MPI-Knoten zu beschränken. Dies wird als ausreichend erachtet, da der Fokus dieser Arbeit auf den Einbeziehungsmöglichkeiten einer lokal vorhandenen GPU liegt. Erste Überlegungen zur Einbeziehung mehrerer MPI-Knoten erfolgt im Anschluss als Ausblick in Kap. 5.7.

5.3 Vorüberlegungen zum Einbezug der GPU

In den hier betrachteten Simulationen ist das Verhältnis der Anzahl Berechnungen zur Anzahl globaler Speicherzugriffe sehr klein, so beträgt es z. B. drei für den Teilchentransport. In der Literatur wird hingegen für die verwendete Grafikkartengeneration ein Verhältnis von mehr als 30 empfohlen [vgl. KH10, S. 97], um einen deutlichen Laufzeitvorteil durch den Einsatz der GPU zu erzielen. Da deshalb die Speicherzugriffe bei den Optimierungen im Vordergrund stehen, werden zunächst allgemeine Überlegungen zu Datenstrukturen und Speichern angestellt.

5.3.1 Verwendete Datenstrukturen

Damit sie sowohl vom Host als auch vom Device aus zugänglich sind und über Kernelaufrufe hinweg bestehen bleiben, müssen die Simulationsdaten im globalen Speicher der Grafikkarte gehalten werden. Beim Zugriff auf den globalen Speicher werden jeweils 128-Byte große Zeilen in den darüberliegenden Level-2-Cache und Level-1-Cache eingelesen. Die Bandbreite beim Speicherzugriff wird am besten ausgenutzt, wenn die gesamte gelesene Speicherzeile auch von den Threads benötigt wird. Den schnellsten Zugriff erzielt man dabei, wenn aufeinanderfolgende Threads hintereinander im Speicher liegende Daten auslesen [vgl. NVI13b].

Um einen solchen Zugriff zu ermöglichen, ist deshalb es nötig, die Datenstrukturen entsprechend zu gestalten. In der Programmiersprache C werden häufig Arrays aus Strukturen (AoS - Array of Structures) verwendet, so auch im Programm PATRIC, in dem die Teilchen als Vektor dargestellt sind. Jedes einzelne Teilchen innerhalb dieses Vektors wird dabei als Struktur mit seinen Koordinaten repräsentiert. Im Kernel auf der GPU wird jedoch von allen Threads zunächst z. B. auf die x-Koordinate der Teilchen zugegriffen, dann auf die y-Koordinate usw. Um einen zusammenhängenden Speicherzugriff zu erzielen, müssen die Datenstrukturen auf der GPU diesem Zugriffsschema angepasst werden. Empfohlen wird in der Literatur eine Struktur aus Einzelarrays (SoA - Structure of Arrays) [vgl. Far11, S. 6], wie in Abb. 5.3 dargestellt. Für die Repräsentation der Teilchen bedeutet dies, dass auf der GPU jeweils ein Einzelarray pro Koordinate genutzt wird. Beim Transfer der Daten muss eine entsprechende Umsetzung erfolgen.

P_1_x		P_n_x	x
P_1_xs		P_n_xs	xs
P_1_y	...	P_n_y	y
P_1_ys		P_n_ys	s
P_1_z		P_n_z	z
P_1_dp		P_n_dp	dp

Abbildung 5.3: Darstellung der Teilchendaten entweder als ein Array aus Strukturen (blau, vertikal) oder als Struktur aus mehreren Einzelarrays (horizontal, grün).

Obwohl diese Datenrepräsentation auf der GPU einfach möglich ist, geht so jedoch der Zusammenhang der Koordinaten zu einem einzelnen Teilchen verloren – im Rahmen der Programmierung muss hierauf besonders geachtet werden. (Im Zusammenhang mit Thrust gibt es wiederum den zip-Iterator, mit dem es möglich ist, über mehrere Arrays gleichzeitig zu iterieren und ein logisch zusammenhängendes Objekt bestehend aus den Einzelinformationen der Arrays zu bearbeiten. Dieser Iterator gleicht so den Nachteil der einzelnen Arrays wieder aus.)

5.3.2 Verwendeter Speicher für statische Daten

Wie in Kap. 2.3.3 beschrieben, stehen bei der Verwendung der GPU verschiedene Speicher zur Verfügung. Wie in den CUDA Programmierrichtlinien zur Optimierung der Speicherzugriffe vorgeschlagen, sollte nach Möglichkeit Speicher auf dem Chip oder gecachter Speicher verwendet werden. Datentransfers zum globalen Speicher der Grafikkarte sollten möglichst vermieden werden [vgl. NVI12, S. 68].

Für die Betrachtung kleinerer Probleme mit wenigen Transportmatrizen kommt zur Speicherung der Matrizen der konstante Speicher in Frage. Dieser ist gecacht und ermöglicht im Vergleich zum globalen Speicher einen schnelleren Zugriff. Allerdings ist der konstante Speicher bei der eingesetzten Grafikkarte nur 64 kB groß. Eine Transportmatrix besteht aus 6×6 Elementen. Im Fall von double-Werten (8 Byte pro Wert) würden also

$$\frac{64 \text{ kB konstanter Speicher}}{36 \text{ Matrixelemente} \cdot 8 \text{ Byte}} \approx 227 \text{ Transportmatrizen}$$

in den konstanten Speicher passen. Im einfachen betrachteten Fall von 16 Transportmatrizen stellt dies natürlich kein Problem dar. Würde man jedoch das zukünftige Synchrotron SIS100 mit seinen 272 Dipolen und Quadrupolen [vgl. FAI08, S. 10] einzeln modellieren wollen, könnten bereits nicht mehr alle Transportmatrizen gleichzeitig im konstanten Speicher gehalten werden.

Eine Möglichkeit wäre, sich den symmetrischen Aufbau von Synchrotrons zunutze zu machen. Der Ring des SIS100 besteht aus sechs gleichartigen Teilstücken, in einem Teilstück gibt es 18 Dipole und 14 Quadrupole. Diese Anzahl passt gut in den konstanten Speicher. Auch bei Anwendung einer Simulationstechnik, bei der Magnete in zwei Teile aufgeteilt werden, um dazwischen weitere Kicks für Multipolkomponenten oder Raumladung einbeziehen zu können, sind diese beiden Teile ebenfalls wieder symmetrisch und müssten nur einmal vorgehalten werden. Bei einem Ansatz, der die vorhandenen Symmetrien ausnutzt, würde zusätzlich eine Zuordnungstabelle benötigt, mit Hilfe derer der Index des Elements im Beschleuniger auf die gespeicherte Matrix abgebildet wird. Eine solche Zuordnungstabelle sollte dann ebenfalls im konstanten Speicher gehalten werden. Sobald jedoch Feldfehler der einzelnen Magnete in die Simulation einbezogen werden sollen, wäre auch eine solche vereinfachte Betrachtungsweise nicht mehr möglich, da die Fehler pro Element wiederum die Größe des konstanten Speichers überschreiten würden.

Neben der Problematik der geringen Größe des konstanten Speichers ist zudem das Programm PATRIC so aufgebaut, dass die Elemente mit ihren Matrizen Teil der Daten sind, die beim Start des Programms dynamisch aus Eingabedateien eingelesen werden. Da der konstante Speicher nur statisch zum Kompilierzeitpunkt mit einer festen Größe allokiert werden kann, müsste man den gesamten konstanten Speicher statisch allokieren und zur Laufzeit die Matrizen blockweise dort hineinladen und sie ggf. mehrmals pro Umlauf austauschen. Dies und die o. g. Gründe, warum bei der aktuellen Größe der konstante Speicher für die Simulationen i. A. nicht ausreichen würde, führen dazu, dass die Transportmatrizen in den vorgeschlagenen Programmodifikationen im globalen Speicher gehalten werden. Sie werden allerdings mit dem Schlüsselwort `const` versehen, um erweiterte Cachemöglichkeiten zu nutzen.

Ein dennoch durchgeführter Vergleich zu den Zugriffszeiten des konstanten und globalen Speichers für den einfachen betrachteten Fall mit nur 16 Transportmatrizen (die natürlich in den konstanten Speicher passen) ergab keinen wesentlichen Laufzeitunterschied, vgl. Anhang A.3.2. Dies wird auch auf die verbesserten Cachemöglichkeiten der genutzten Generation von Grafikkarten zurückgeführt (im Vergleich zu Grafikkarten älteren Typs). Daneben war, vermutlich bedingt durch den genutzten Grafikkartentreiber in Verbindung mit der CUDA Berechnungsfähigkeit der Grafikkarte, die Speicherung im konstanten Speicher nur von `float` Werten möglich. Im Folgenden wurde deshalb von einer Nutzung des konstanten Speichers abgesehen.

5.3.3 Verwendeter Speicher für dynamische Daten

Daten, die im Rahmen der Simulation sehr häufig geändert werden, sind die Teilchenkoordinaten und die Informationen auf den Gitterpunkten. Beide müssen mit dem Host ausgetauscht werden und über Kernelaufrufe hinweg gespeichert werden. Deshalb müssen sie, wie bereits vorher bemerkt, im globalen Speicher der Grafikkarte gehalten werden. Auch die Anzahl der Teilchen (min. 100.000, entspricht 3,43MB) erzwingt eine Speicherung der Teilchendaten im globalen Speicher.

Wegen der hohen Zugriffszeiten auf den globalen Speicher wird jedoch in den CUDA Programmierrichtlinien vorgeschlagen, Daten ggf. dennoch in den gemeinsamen Speicher auf dem Chip zu laden, um bei einer großen Menge von Berechnungen von dessen reduzierten Zugriffszeiten zu profitieren. Für die Teilchen kann also nicht von vorn herein festgelegt werden, in welchem Speicher sie während der Berechnungen gehalten werden, vielmehr bietet diese Frage Ansatz für Optimierungspotential. Dasselbe gilt auch für Informationen auf Gitterpunkten. Die Anzahl der Gitterpunkte ist im Verhältnis zur Teilchenzahl kleiner. Gerade da Informationen auf Gittern von vielen Threads verändert werden, könnte hierbei ein Ansatz günstig sein, Kopien der Gitter pro Threadblock zunächst in den gemeinsamen Speicher zu laden und dort zu verändern, bevor sie in den globalen Speicher zurückgeschrieben werden müssen.

Hauptsächlich wirft also die Speicherung der dynamischen Daten bzgl. des verwendeten Speichers auf. Sie muss auf Grund der Zugriffe auf die Daten beantwortet und im Rahmen der Implementierungsvarianten berücksichtigt werden.

5.4 Tracking mit PATRIC

Im Fokus der Untersuchungen stand zunächst die Einbeziehung der GPU für das reine Tracking der Teilchen. Hierunter versteht man die Verfolgung der Teilchen entlang ihres Wegs im Beschleuniger und dabei die Veränderung der Teilchenkoordinaten durch die Transportmatrizen. Dies entspricht einer rein tranversalen Betrachtung der Teilchenbewegung. Verwendet wurde eine auf genau diesen Teilchentransport zugeschnittene Variante des Programms PATRIC. Auf Grund der Überlegungen in Kap. 5.2.1 wurde von der Verwendung von MPI abgesehen und ausschließlich ein Knoten betrachtet, auf dem die GPU einbezogen werden soll. Da auch technisch die Unterstützung für das Debugging und Profiling für hybride MPI/GPU-Anwendungen in der verwendeten Entwicklungsumgebung nsight noch nicht verfügbar war (angekündigt für eine der folgenden Versionen), wurden die entsprechenden Stellen im Quelltext auskommentiert. Diese modifizierte Version von PATRIC ohne MPI wurde im Folgenden für die Untersuchungen zur Einbeziehung der GPU genutzt.

Für die Messungen wurden die Standardparameter mit einer einfachen konstant fokussierenden Optik mit 16 Transportmatrizen und 100.000 Teilchen gewählt. Messungen mit einem Tracking rein auf der CPU zeigen, dass das Originalprogramm fast zwei drittel der Zeit (63.7%) mit dem eigentlichen Transport verbringt (`SectorMap::transport` und `Pic::transport`). Der Grund ist, dass hier die eigentliche Berechnung erfolgt, d. h. in einer Schleife über alle Teilchen werden mit Hilfe der Transportmatrizen die Teilchenkoordinaten verändert. An zweiter Stelle folgt eine Funktion, die Teilchen, die sich mehr als einen halben Umlauf im Ring entfernen, auf der anderen Seite wieder in die Betrachtung wandern lässt (`Pic::periodic_bc` – periodic boundary condition). Hier ist ebenfalls eine Schleife über alle Teilchen enthalten, in der die Koordinaten ggf. korrigiert werden. Danach folgen Methoden zur Auswertung von Teilchengrößen und zur Ausgabe der Informationen. Deren Anteil an der Laufzeit des Gesamtprogramms hängt davon ab, wie häufig Ausgaben geschrieben werden sollen. In den Messungen wurden zunächst Zwischenergebnisse einmal pro Umlauf der Teilchen durch den Beschleuniger ausgegeben.

```
 %   cumulative  self              self    total
time seconds seconds    calls  ns/call ns/call name
38.97   4.27   4.27 204900000   20.84   20.84  SectorMap::transport(..)
24.73   6.98   2.71                           Pic::transport(..)
15.97   8.73   1.75                           Pic::periodic_bc(..)
 5.66   9.35   0.62   4300000  144.21  144.21  Pic::get_phaseadvance_h(..)
 4.56   9.85   0.50                           Pic::print(..)\
 4.38  10.33   0.48   4300000  111.65  111.65  Pic::get_phaseadvance_v(..)
 1.10  10.45   0.12                           Pic::rms_emittance_x()
 ...
```

Abbildung 5.4: Ausgabe des Profilers gprof (verkürzt): fast zwei Drittel der Zeit verbraucht das Originalprogramm im Transportschritt.

5.4.1 Einfacher Ansatz, Transportschritt auf der GPU

Da in der verwendeten Version von PATRIC die meiste Zeit im Transportschritt verbracht wird, ist dieser Schritt Ziel der Überlegungen zur Parallelisierung. Die Daten, mit denen die entsprechende Funktion operiert, sind die dynamischen Teilchendaten und die statischen Transportmatrizen. Da die Transportmatrizen von allen Threads gelesen werden müssen, werden sie nach den Vorüberlegungen in Kap. 5.3.2 zwar im globalen Speicher der GPU gehalten, jedoch mit dem Schlüsselwort `const` versehen. Eine geometrische Datenzerlegung kann an Hand der Teilchendaten erfolgen. Dies stellt im Vergleich zum Originalprogramm ein Schleifenparallelismus dar: die ursprüngliche Schleife über die Teilchen wird parallelisiert. Da durch die Matrix-Vektor-Multiplikation einzelne Teilchenkoordinaten verändert werden, kann die Datenzerlegung dabei entweder bis auf das einzelne Teilchen oder sogar bis auf die einzelne Koordinate erfolgen. Durch die Organisation in Threads und Blöcken und deren asynchroner Ausführung stehen für beide Arten der Zerlegung genügend Ausführungseinheiten zur Verfügung.

Da nur der Transportschritt parallelisiert werden sollte, blieb das restliche Programm unverändert. Als einzige Änderung werden die Teilchendaten vor jedem Transportschritt auf die GPU kopiert und danach wieder zurück. Dabei muss jeweils die Struktur der Daten von einem Array aus Strukturen auf dem Host auf die auf der GPU benötigten Einzelarrays umgesetzt werden. Zur Vereinfachung wurde in dieser Variante noch ein einziges großes Array verwendet, die eigentlichen Einzelarrays kamen erst in den folgenden Programmodifikationen zum Einsatz. Die Funktionen zur Korrektur der Teilchenkoordinaten und zum Entfernen verlorengegangener Teilchen verblieben auf der CPU. Erwartet wird durch das Kopieren der Teilchen und die serielle Umsetzung der Datenstruktur ein zusätzlicher Aufwand und zusätzliche Programmlaufzeit. Die Berechnung selbst kann stark beschleunigt werden, da die Laufzeit des Algorithmus von $O(n)$ auf $O(\frac{n}{p})$ zurückgeht, vgl. den in Quelltext 5.1 dargestellten Pseudocode.

Im Rahmen der Implementierung wurden beide Varianten umgesetzt: ein Thread behandelt entweder ein einzelnes Teilchen oder nur eine einzelne Teilchenkoordinate. Der Quelltext ist in Anhang A.3.1 dargestellt. Die Zeitmessung wurde zum einen mit dem Unix-Befehl `clock_gettime()` für die Prozessorzeit und zum anderen mit CUDA Events und den Befehlen `cudaEventCreate(&start)` und `cudaEventRecord(start, 0)` für die Grafikprozessorzeit durchgeführt, vgl. Anhang A.2. Zu beachten ist hierbei, dass die Zeitmessung selbst ca. 1 s in Anspruch nimmt, für echte Simulationen sollte sie also ausgeschaltet werden.

Quelltext 5.1: PRAM Pseudocode: Transportschritt auf der GPU (Matrix-Vektor-Multiplikation).

```
1 for i=1 to n pardo
2   for i=m to 6 do
3     value := 0
4     for i=n to 6 do
5       value += transportMatrix[m,n] * particleVectors[n]
6     particleVectors[m] := value
```

Die Variante, bei der pro Thread eine Teilchenkoordinate berechnet wird, nimmt die Ressourcen der GPU stärker in Anspruch, da hier sechs mal so viele Threads ausgeführt werden. Im Vergleich zur Originalversion ergibt sich eine leicht kürzere Laufzeit von 17,95 *s*, die Einbeziehung der GPU für die Berechnung bringt hier leichte Vorteile. Zur Analyse, ob die in dieser Version notwendige Synchronisation der Threads (vgl. den Quelltext ist in Anhang A.3.1) einen Flaschenhals darstellt, wurde `syncthreads()` für eine Ausführung bewusst auskommentiert, obwohl sich dadurch falsche Ergebnisse ergeben. Die Laufzeit veränderte sich jedoch kaum (17,99 *s*), sodass die Synchronisation selbst keinen negativen Einfluss auf die Laufzeit der GPU hat. Um zu ermitteln, welchen Anteil die Berechnungen auf der GPU einnehmen, wurden diese für einen Vergleich auskommentiert, wodurch sich der Laufzeitanteil auf der GPU reduziert und eine Gesamtzeit von 15,59 *s* ermittelt wurde. Die Menge der Berechnungen auf der GPU ergeben also einen Laufzeitunterschied, was zu erwarten war. Zu erkennen ist, dass auch ohne Berechnungen allein der Aufruf des Kernels incl. der anschließenden Synchronisation der GPU mit dem Host 0,43 *s* in Anspruch nimmt.

Die Variante, bei der ein Teilchen pro Thread berechnet wird, zeigt eine Beschleunigung von 1,18 im Vergleich zur Originalversion. Gegenüber der Variante mit einem Thread pro Koordinate ist immerhin noch eine Beschleunigung von 1,11 zu beobachten. Es kann also geschlossen werden, dass bei letzterer der Aufwand durch das Erzeugen der sechs mal mehr Threads im Verhältnis zu den verbleibenden sehr wenigen Berechnungsschritten pro Kernel zu groß ist. Aus diesem Grund wird als Grundlage für weitere Entwicklungen auf der etwas schnelleren Version aufgesetzt, bei der pro Thread ein Teilchen bearbeitet wird. Da in den folgenden Quelltextänderungen Aspekte auf der GPU behandelt werden sollen, die sich nicht nur auf die Einzelkoordinate beziehen, sondern meist auf das gesamte Teilchen, erscheint es generell besser, pro Thread auf der GPU jeweils ein Teilchen zu behandeln. Dies entspricht auch den Abläufen der Simulationen, da die betrachteten Effekte jeweils auf ein ganzes Teilchen wirken. Insgesamt wirkt der Quelltext dadurch wartbarer.

In einer weiteren Messung wurden bewusst die Kopierschritte weggelassen, die vor jedem Transportschritt die Daten vom Host zur Grafikkarte und danach wieder zurück kopieren. Es ist deutlich sichtbar, dass sich diese negativ auf die Laufzeit auswirken.

Tabelle 5.2: Transportschritt auf der GPU: Vergleich, wenn pro Thread eine Koordinate oder ein Teilchen berechnet wird. Die Variante mit einem Teilchen pro Thread zeigt eine Beschleunigung von 1,18 gegenüber der Originalversion.

Variante	CPU-Zeit	GPU-Zeit	Summe
CPU: Originalversion	19,04 *s*	—	19,04 *s*
GPU: Transportschritt, Thread: Koordinate	15,21 *s*	2,74 *s*	17,95 *s*
(Version ohne Synchronisation)	15,26 *s*	2,73 *s*	17,99 *s*
(Version ohne Berechnungen)	15,16 *s*	0,43 *s*	15,59 *s*
GPU: Transportschritt, Thread: Teilchen	15,30 *s*	0,82 *s*	16,12 *s*
(Version ohne Daten zu kopieren)	10,83 *s*	0,99 *s*	11,82 *s*

5.4.2 Teilchen auf der GPU halten

Kopiert man wie im vorangegangenen Beispiel die Teilchen für jeden einzelnen Transportschritt auf die GPU, dominiert dieser Kopieraufwand die eigentliche Berechnung. Aus diesem Grund erscheint es sinnvoll, die Teilchen nur dann wieder auf den Host zurück zu kopieren, wenn sie zur Ermittlung von Zwischenergebnissen benötigt werden oder die gesamte Berechnung beendet ist. Dies entspricht auch der Empfehlung in den CUDA Programmierrichtlinien, die Daten auf die GPU zu bringen und sie möglichst lang dort zu halten [vgl. NVI13b].

Das Programm wurde entsprechend um die Routinen `moveParticlesToHost()` und `moveParticlesToDevice()` erweitert, die während des Ablaufs aufgerufen werden können. Da nun mehrere Transportschritte hintereinander auf der GPU ausgeführt werden, ohne die Teilchen zum Host zurück zu kopieren, müssen die Funktionen zur Korrektur der Teilchenkoordinaten und zur Behandlung eines Teilchenverlusts ebenfalls auf die GPU verlagert werden. Da bei der GPU-Programmierung mit Arrays fester Größe gearbeitet wird, wurde ein weiteres `boolean`-Array `valid` vorgesehen, das angibt, ob ein Teilchen i noch existiert ($\texttt{valid}_i$ = `true`). Diese Information wird beim Zurückkopieren der Teilchen dazu verwendet, nur die noch gültigen Einträge in den Teichenvektor auf dem Host zu übernehmen. Die genutzte Datenstruktur für die Teilchen ist in Anhang A.3.3 dargestellt. Um eine Anpassung an eine sich verändernde Teilchenzahl zu erzielen, wäre es möglich, die Datenstrukturen auf der GPU während des Simulationslaufs in ihrer Größe anzupassen oder zumindest die noch gültigen Teilchen in den vorderen Teil des Teilchenarrays zu bringen, um ein Divergieren der Threads zu verhindern. Da jedoch während einer normalen Simulation mit keinem großen Teilchenverlust zu rechnen ist, wurde hierauf verzichtet.

Bevor mit den eigentlichen Messungen begonnen wurde, musste zunächst die Anzahl Threads pro Block festgelegt werden. Wie in Kap. 2.3.3.2 besprochen, sollte diese ein Vielfaches von 32 sein, in der Literatur wird 256 als typisch genannt [vgl. Far11]. Laut des auf der NVIDIA Webseite bereitgestellten Rechners zur Ermittlung der optimalen Blockgröße für eine gute Auslastung der GPU sind auf Grund der verwendeten Register und des belegten gemeinsamen Speichers mehrere Konfigurationen möglich [vgl. NVI13c]. Tab. 5.3 stellt die so ermittelten möglichen Blockgrößen zusammen mit den gemessenen Laufzeiten dar. Da sich die Zeiten nicht wesentlich unterscheiden und erst ab 512 ein leichter Laufzeitanstieg zu beobachten ist, wird im Folgenden die in der Literatur empfohlene Anzahl von 256 Threads pro Block gewählt.

Tabelle 5.3: Teilchen auf der GPU: Vergleich der Anzahl Threads pro Block.

Threads pro Block	CPU-Zeit	GPU-Zeit	Summe
64	2,57 s	1,47 s	4,04 s
128	2,61 s	1,48 s	4,09 s
256	2,60 s	1,50 s	4,10 s
512	2,64 s	1,61 s	4,25 s

An dieser Stelle sei angemerkt, dass es sich bei den dargestellten Zeiten um Prozessorzeiten der CPU und GPU handelt. In den durchgeführten Messungen entsprachen diese durchaus der tatsächlich verstrichenen Zeit (wallclock time), da das Rechnersystem für Messungen exklusiv zur Verfügung stand. Wird der Rechner unter normalen Bedingungen gleichzeitig von Kollegen für Simulationen genutzt, stieg die Ausführungszeit in einer beispielhaft durchgeführte Messung um ca. 15% an. Da dies jedoch vom Ressourcenbedarf der anderen Programme abhängt und auch auf die reine CPU-Variante zutrifft, werden im Folgenden weiterhin nur die Prozessorzeiten betrachtet.

Um das generelle Verhalten des parallelen Programms bei wachsender Problemgröße zu untersuchen, wurden alle Parameter festgehalten und nur die Teilchenzahl zwischen 10.000 und 1.000.000 variiert. Die Laufzeiten wurden mit denen des unveränderten Programms verglichen. Beim unveränderten Programm wird ein linearer Anstieg der CPU-Zeit mit der Variation der Teilchenzahl erwartet, da die in Kap. 3.1 beschriebenen zentralen Schleifen des Programms über die Teilchen eine Komplexität von O(n) haben. Dies spiegelt sich in den Messungen gut wieder. Bei der GPU-Variante reduziert sich der Aufwand von O(n) auf O($\frac{n}{p}$). Als zusätzlicher Aufwand kommt das Kopieren der Daten vom Host zum Device und umgekehrt hinzu. Hier ist zunächst nicht bekannt, wie sich dieser Aufwand bei wachsender Problemgröße verhält.

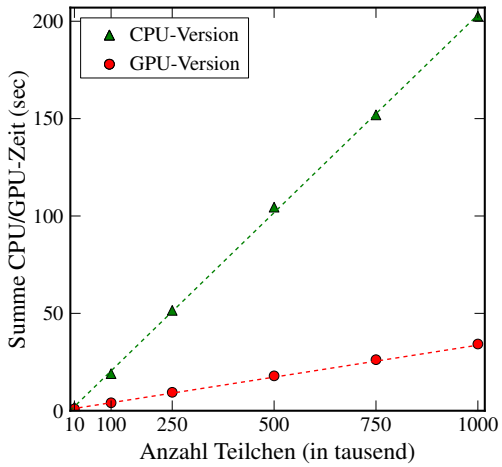

Abbildung 5.5: Messung der modifizierten PATRIC Version, in der die Teilchen so lang wie möglich auf der GPU gehalten werden, im Vergleich zur Originalversion bei Variierung der Teilchenzahl. Zu beobachten ist eine 6-fache Beschleunigung.

Deutlich zu sehen in Abb. 5.5 ist der lineare Anstieg sowohl der Originalversion von PATRIC als auch der Version, bei der die Teilchen auf der GPU gehalten werden. Dies ist interessant, da es bedeutet, dass sich eine Variation der Teilchenzahl auch in der parallelen Version nur linear auswirkt. Die parallele Variante verzeichnet dabei eine 6-fache Beschleunigung gegenüber der seriellen, wobei ab ca. 8150 Teilchen die GPU-Variante schneller ist. Es ist festzuhalten, dass der Einsatz von GPUs für das Tracking der Teilchen eine gute Beschleunigung erzielt, wenn die Teilchen so lang wie möglich auf der GPU gehalten werden können.

Obwohl eine 6-fache Beschleunigung zu verzeichnen war, ist wichtig festzuhalten, dass dies vor allem durch die eingesetzte Grafikkarte mit 14 Multiprozessoren ermöglicht wird. Bei einer Messung mit dem ebenfalls zur Verfügung stehenden Laptopsystem mit nur zwei Multiprozessoren konnte eine solche Beschleunigung nicht erzielt werden, hier ergab sich sogar eine Verlangsamung der Ausführung um den Faktor zwei. Für zukünftige Projekte ist also wichtig zu verzeichnen, dass – wie auch erwartet – die Anzahl der parallelen Ausführungseinheiten auf der Grafikkarte den Haupteinfluss auf die parallele Laufzeit hat. Ihre Anzahl bestimmt, wie viele Berechnungen parallel abgearbeitet werden können.

Wie gezeigt, ist bei gleichbleibender Anzahl paralleler Ausführungseinheiten zu empfehlen, die Teilchen möglichst lang auf der GPU zu halten. Hält man die Teilchen jedoch lang auf der GPU und ergibt sich ein größerer Teilchenverlust im Verlauf der Simulation, kommt es zu einem Divergieren der Threads. Threads, welche die noch gültigen Teilchen bearbeiten, führen dabei noch Berechnungen durch, andere nicht mehr. Um abschätzen zu können, welchen Einfluss diese Tatsache auf die Laufzeit hat, wurde die GPU-Version des Programms einmal mit und einmal ohne Teilchenverlust verglichen. Im Programm wurde hierbei die ungünstigste Variante eines 50%-igen Teilchenverlusts simuliert, indem das `boolean`-Array, dass die noch gültigen Teilchen anzeigt, an jeder zweiten Stelle den Wert `false` enthält.

Wie die Messungen zeigen, ist allerdings die Auswirkung der Thread-Divergenz auf die GPU-Zeit sehr gering, die Zeit steigt nur um knapp 1,5% an, vgl. Tab. 5.4. Die geringfügig längere Laufzeit der GPU wird durch die deutlich geringere CPU-Zeit ausgeglichen, denn hier wird beim Zurücklesen der Daten nur noch mit den tatsächlich existierenden Teilchen weiter gearbeitet. Dass die Thread-Divergenz einen so geringen negativen Einfluss hat, kann darauf zurückgeführt werden, dass die eine Hälfte der Threads bei einem verloren gegangenen Teilchen tatsächlich keine Berechnungen durchführen muss, sondern einfach nur nicht an den Berechnungen teilnimmt. Auf Grund dieser Ergebnisse kann im Folgenden davon abgesehen werden, für Teilchenverluste eine spezielle Behandlung auf der GPU vorzusehen. Es muss trotz Teilchenverlusten keine Neusortierung o. ä. vorgenommen werden.

Tabelle 5.4: Teilchen auf der GPU: Vergleich der Laufzeit mit und ohne Teilchenverlust (50% Teilchenverlust, maximal divergierende Threads auf der GPU).

Variante	CPU-Zeit	GPU-Zeit	Summe
Teilchen auf der GPU, kein Teilchenverlust	2,60 *s*	1,49 *s*	4,09 *s*
Teilchen auf der GPU, 50% Teilchenverlust	1,63 *s*	1,51 *s*	3,14 *s*

Wie die vorangegangenen Messungen gezeigt haben, verbessert das Halten der Daten auf der GPU die Laufzeit deutlich. Die Divergenz der Threads auf Grund von Teilchenverlusten hat darüber hinaus keinen negativen Einfluss auf die Gesamtlaufzeit, so dass auch bei größeren Teilchenverlusten die Teilchen möglichst lang auf der GPU gehalten werden können. Diese auf die GPU portierte Variante von PATRIC diente im Folgenden aus Ausgangspunkt für weitere Untersuchungen.

5.4.3 Ausgabe von Zwischenergebnissen

Die Ausgabe von Zwischenergebnissen während der Programmausführung dient dazu, Effekte über den Verlauf der Simulation zu beobachten. Zwischenergebnisse sind ermittelte Strahlgrößen wie die Emittanz, aber auch die Teilchendaten selbst. Davon ausgehend, dass die Teilchen möglichst lang auf der GPU gehalten werden sollen, stellt sich die Frage, wie häufig Ausgaben dann noch möglich ist, da dazu die Teilchendaten auf den Host zurück kopiert werden müssen. Im Folgenden wurde untersucht, welchen Einfluss Ausgaben auf die Laufzeit haben. Dazu wird die Ausgabehäufigkeit variiert, um zu generellen Empfehlungen zu kommen. Verglichen werden typische Szenarien, bei denen Zwischenergebnisse nach einem Umlauf (entspricht 16 Transportmatrizen) oder mehreren Umläufen der Teilchen durch den Beschleuniger ermittelt werden sollen. Die Frage stand im Vordergrund, ob evtl. durch häufige Ausgaben der Laufzeitgewinn durch einen Einsatz der GPU zunichte gemacht wird. Erwartet wird generell, dass sich mit selteneren Ausgaben die Laufzeit des Programms verkürzt.

Abb. 5.6 zeigt, dass es auf jeden Fall gut möglich ist, auch beim Halten der Teilchen auf der GPU Zwischenergebnisse auszugeben. Die Laufzeit der GPU-Version bleibt immer unterhalb der der CPU-Version. Der zusätzliche Laufzeitgewinn stagniert jedoch rasch. Werden die Daten seltener als alle fünf Umläufe ausgegeben, ist kaum ein weiterer Performanzgewinn zu beobachten. Bei einer Ausgabe häufiger als einmal pro Umlauf steigt die Laufzeit stark an. Gibt man tatsächlich nach jedem Transportschritt durch jedes der 16 Elemente im Beschleuniger Zwischenwerte aus, steigt die Laufzeit der GPU-Variante auf ca. 92 s und die der CPU-Variante auf ca. 445 s an (nicht mehr in der Abbildung gezeigt)! Aus diesem Grund sollten häufigere Ausgaben möglichst vermieden und nur dann eingesetzt werden, wenn dies für die Simulation notwendig ist. Festgestellt werden kann also, dass vor allem eine häufige Ausgabe von Zwischenergebnissen einen negativen Einfluss auf die Laufzeit hat.

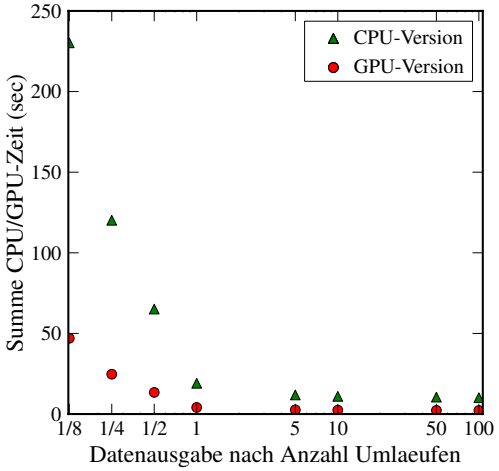

Abbildung 5.6: Messung der modifizierten PATRIC Version, in der die Teilchen auf der GPU gehalten werden im Vergleich zur Originalversion bei Variierung der Häufigkeit, in der Zwischenergebnisse ausgegeben werden.

Falls die Notwendigkeit besteht, Zwischenergebnisse sehr häufig auszugeben, kann die Möglichkeit interessant werden, die Schritte zur Datenausgabe soweit wie möglich mit der nächsten Berechnung zu überlappen. Wie in Kap. 2.3.4.3 beschrieben, bieten Streams diese Möglichkeit. Überlappt werden kann eine Kernelausführung mit jeweils einem Datentransfer vom bzw. zum Host. Im vorliegenden Fall wurde beispielhaft eine Variante implementiert, bei der das Kopieren der Daten zurück auf den Host zu Ausgabezwecken mit der Berechnung des nächsten Transportschritts auf der GPU durch die Nutzung verschiedener Streams überlagert wurde, siehe Abb. 5.7.

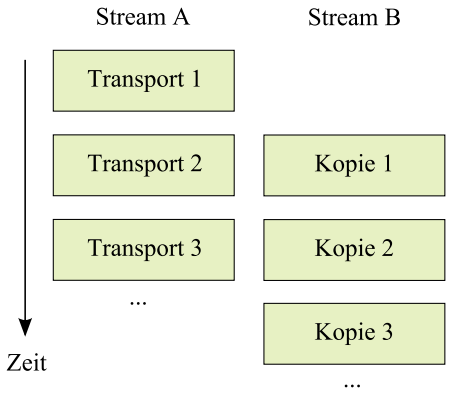

Abbildung 5.7: Schema der Überlappung des Kopierens der Daten zurück auf den Host mit der nächsten Berechnung.

Damit die Teilchendaten kopiert werden können, während bereits die nächste Berechnung auf ihnen durchgeführt wird, wurde mit zwei Mengen an Teilchenarrays gearbeitet. In jedem Transportschritt werden die einen als Ausgangswerte genommen und nur die Zielwerte verändert, im nächsten Transportschritt werden die vorherigen Zielwerte als neue Ausgangswerte interpretiert. Dies wird durch ein Austauschen von Zeigern auf die Daten erreicht (engl. pointer switching), vgl. Anhang A.3.4.

Da jedoch in den Simulationen die Speicherzugriffe die Berechnungen dominieren, ergibt sich durch die Nutzung der unterschiedlichen Streams für das Kopieren und die Berechnung nur ein Laufzeitgewinn von 5%, siehe Tab. 5.5. Verglichen wurde die Variante, bei der nach jedem Element im Beschleuniger Daten ausgegeben werden. Da bei normalen Simulationen die Daten seltener ausgegeben werden und dann weniger überlappt werden kann, wird der Gewinn durch die Nutzung von Streams i. A. nicht so groß sein. Aus diesem Grund ist festzustellen, dass sich die Möglichkeiten der Streams für die vorliegenden Simulationen nicht lohnen.

Tabelle 5.5: Vergleich der Laufzeit bei einer Ausgabe nach jedem Element im Beschleuniger ohne und mit Nutzung von Streams.

Variante	**CPU-Zeit**	**GPU-Zeit**	**Summe**
GPU sequentiell, 1 Stream	89,69 *s*	1,62 *s*	91,31 *s*
GPU überlappend, 2 Streams	84,81 *s*	1,52 *s*	86,33 *s*

5.4.4 Berechnung von Strahlgrößen auf der GPU

Wie im vorangegangen Kapitel veranschaulicht, steigen die Programmlaufzeiten stark an, sobald häufig viele Daten ausgegeben werden sollen. Aus diesem Grund liegt die Idee nahe, sich bei der Ausgabe rein auf die Strahlgrößen zu beschränken und diese direkt auf der GPU zu ermitteln. So entfällt die Notwendigkeit, vor jeder Ausgabe die kompletten Teilchendaten zurück zum Host kopieren zu müssen. Auf der anderen Seite ist das Zusammenfassen von Werten auf der GPU prinzipiell laufzeitintensiv. Am Beispiel der Ermittlung der Emittanz soll deshalb untersucht werden, inwiefern es vorteilhaft ist, Strahlgrößen direkt auf der GPU zu berechnen. Dazu wird eine Eigenimplementierung vorgeschlagen und mit der bestehenden CPU-Variante verglichen. Diese in CUDA geschriebene Eigenimplementierung wird im Anschluss einer bereits in der Abteilung Strahlphysik vorhandenen Emittanzermittlung verglichen, die statt CUDA die Thrust Bibliothek verwendet. An Hand dieses Beispiels soll untersucht werden, wie sich die Verwendung der einfacheren Thrust Schnittstelle auf die Laufzeit des Programms auswirkt.

5.4.4.1 Eigenimplementierung der Emittanzberechnung

Zur Berechnung der Emittanz sind in zwei Schritten zuerst die durchschnittlichen Werte der Teilchenkoordinaten (hier z. B. horizontal x und xs) zu bilden und danach die Summen der Abweichungen, vgl. Kap. 2.1.2. Die nötigen Summen können mit Hilfe eines umgekehrten Binärbaums generell in $O(\log n)$ Zeit ermittelt werden. Da zwischen beiden Schritten eine globale Synchronisierung notwendig ist, wurden für die Eigenimplementierung der Emittanzberechnung auf der GPU die beiden Schritte jeweils als einzelner Kernel umgesetzt.

Nach den Vorüberlegungen im Kap. 2.4.2 ist es von Vorteil, die Daten blockweise zu summieren. Für die Eigenimplementierung wird aus diesem Grund eine Reduktion mittels binärer Summierung vorgeschlagen. Bereits NVIDIA selbst veröffentlicht Optimierungsstrategien für solche sog. Reduzieroperationen [vgl. Har07], die vor allem auch auf einen möglichst optimierten Speicherzugriff zugeschnitten sind und sich deshalb gut für eine GPU-Implementierung eignen. Es soll an dieser Stelle nicht auf dem schnellsten der dort genannten Algorithmen aufgesetzt werden. Statt dessen wird als Grundlage ein schnellerer, aber dennoch weiterhin gut les- und wartbarer Algorithmus gewählt (sog. „sequentielle Adressierung“). Der umgesetzte Algorithmus ist als Pseudocode 5.2 und in Abb. 5.8 schematisch dargestellt.

Quelltext 5.2: PRAM Pseudocode: Summierung und sequentieller Adressierung.

```
1 for i=1 to n pardo
2   i := blockDim.x/2
3   while i > 0
4     tempData[i] += tempData[i + j];
5     i := i / 2
```

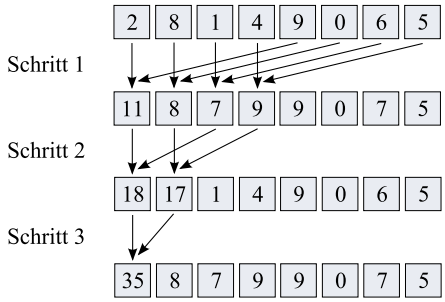

Abbildung 5.8: Schema der Reduktion mittels sequentieller Adressierung.

Im Verfahren der Reduktion werden für eine Summierung dabei von jedem Thread-Block zunächst die Daten in den gemeinsamen Speicher gelesen. Hierbei werden genau so viele Daten eingelesen, wie Threads existieren (z. B. 256 Werte). Diese werden dann mit Hilfe des Algorithmus reduziert und so eine Summe pro Thread-Block gebildet. Das Ergebnis wird von einem der Threads wieder in den globalen Speicher geschrieben, auch hier kommt wieder eine atomare Operation zum Einsatz, um zur globalen Summe zu gelangen. Der zugehörige Quelltext befindet sich in Anhang A.3.5. Bei den durchgeführten Messungen kamen jeweils Programmvarianten zum Einsatz, bei denen lediglich die Emittanz ausgegeben wurde und nicht mehr die kompletten Teilchendaten, weshalb sich insgesamt eine kürzere Laufzeit ergibt.

Obwohl die Reduzieroperation ausschließlich im gemeinsamen Speicher stattfindet, und somit schnell ausgeführt werden kann, ergibt sich insgesamt jedoch eine schlechtere Laufzeit, als wenn die Teilchendaten komplett zum Host kopiert werden und dort die Auswertung erfolgt, siehe Abb. 5.9. Dies war so nicht erwartet worden, da die Laufzeit pro Block der Summierung von $O(n)$ auf $O(\log n)$ sinkt. Die Messungen zeigen jedoch deutlich, dass Operationen über alle Daten, bei denen speziell die Zugriffe auf Speicherstellen zu synchronisieren und damit zu serialisieren sind, insgesamt aufwändiger sind. Zur Optimierung der Implementierung wurden dabei bereits schon die in beiden Schritten der Emittanzbestimmung nötigen Einzelsummen in jeweils zwei Kerneln zusammengefasst. Dieses Zusammenfassen führt jedoch zu speziell für einzelne Strahlgrößen entwickeltem Quelltext, der schlecht wiederverwendbar ist.

5.4.4.2 Emittanzberechnung mit Thrust

Wie bereits angedeutet, ist es eine wichtige Frage, was bei einer Portierung eines Simulationsprogramms auf die GPU mit bestehenden Auswerteverfahren geschieht. Allgemein ist ein hoher Aufwand damit verbunden, alle bestehenden Routinen zur Ermittlung von Strahlgrößen für die GPU neu zu entwickeln, was deshalb nicht erstrebenswert ist. Wie jedoch am Beispiel der Eigenimplementierung der Emittanzberechnung gesehen, ist eine Nachentwicklung auf der GPU nicht unbedingt schneller und erscheint somit für bestehende Auswertungen auch nicht notwendig.

Falls jedoch Auswertungen neu entwickelt werden, stellt sich natürlich die Frage, ob die GPU einfach eingebunden werden kann. Selbst wenn auf der GPU keiner oder nur ein geringer Laufzeitvorteil erzielt werden kann, kann es durchaus sinnvoll sein, eine solche Auswertung auch auf der GPU zur Verfügung zu haben. So könnten z. B. unabhängig von Ausgaben Strahlgrößen auf der GPU ermittelt werden, die dazu führen, ggf. beim Überschreiten von Schwellwerten die Simulation abzubrechen. Auch wenn die Laufzeitschnittstelle der GPU wie bisher gesehen recht gut les- und wartbar ist, entsteht bei ihrer Nutzung viel technischer Quelltext, der den Blick auf den eigentlichen Algorithmus verdeckt. Außerdem ist eine gewisse Einarbeitung nötig, um mit CUDA Programme für die GPU schreiben zu können.

Aus diesem Grund kam die Frage auf, inwiefern für solche zukünftigen Entwicklungen die Thrust Bibliothek genutzt werden kann, welche einen einfacheren Zugang zur GPU verspricht, vgl. Kap. 2.3.4.4. Im Rahmen der Arbeit sollte deshalb an Hand eines kurzen Beispiels die Handhabbarkeit der Bibliothek und die aus deren Nutzung resultierenden Laufzeiten beurteilt werden. Eine Emittanzberechnung mit Thrust, die im Rahmen erster Versuche mit der GPU in der Abteilung Strahlphysik entstanden ist, diente dabei als Grundlage für einen Vergleich; der Quelltext ist in Anhang A.3.6 aufgeführt.

Die Messungen mit der Thrust-Variante zeigen im Vergleich zur Eigenentwicklung eine deutlich schnellere Laufzeit, die mit der ursprünglichen CPU-Variante vergleichbar ist, siehe Abb. 5.9. Es wird davon ausgegangen, dass die Operationen in Thrust speziell optimiert sind und so im Vergleich zur Eigenentwicklung Vorteile haben. Allerdings ist anzumerken, dass hierbei nicht das `boolean`-Array ausgewertet wurde, dass die noch gültigen Teilchen repräsentiert, da es sich hier nur um einen ersten Vergleich handeln sollte. Würde man dies einbeziehen, ergäbe sich eine leicht längere Laufzeit. Bei der Verwendung von Thrust ist festzustellen, dass man nur mit den zur Verfügung gestellten Funktionen der Schnittstelle arbeiten kann, was teilweise etwas unhandlich erscheint. Funktionen müssen hierbei z. B. einzeln aufgerufen werden, die ggf. im Rahmen eines selbst implementierten Kernels zusammengefasst werden könnten.

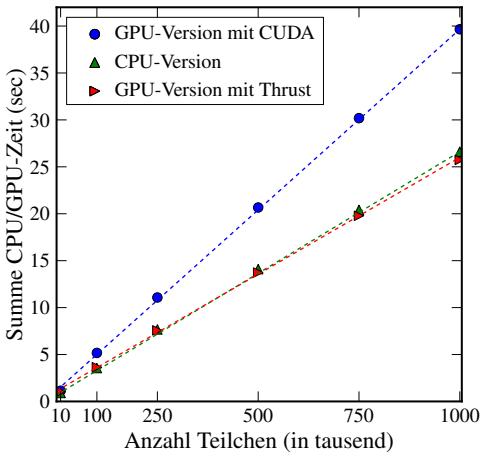

Abbildung 5.9: Messung der modifizierten PATRIC Version, in der die Emittanzermittlung auf der CPU oder in zwei Varianten (CUDA und Thrust) auf der GPU erfolgt.

Wie in beiden GPU-Varianten veranschaulicht, ist es durchaus möglich, Strahlgrößen auf der GPU zu berechnen und nur diese dem Host zur Verfügung zu stellen, ohne dafür die gesamten Teilchendaten zum Host kopieren zu müssen. Allerdings ergab sich mit der genutzten Eigenimplementierung eine Laufzeitverschlechterung, die bei der Abteilung bereits im Vorfeld entstandene Thrust-Variante wies ungefähr dieselbe Laufzeit wie die CPU-Variante auf. Insgesamt ist festzuhalten, dass Reduktionsoperationen generell teuer sind und hierbei die GPU bei der Ermittlung von Strahlgrößen zunächst keine Vorteile bietet. Dies spricht dafür, die Auswertungen zunächst auf dem Host zu belassen. Das deckt sich auch damit, dass bestehender Quelltext zur Auswertung der Teilcheninformationen existiert, und dieser so nicht zwingend auf die GPU portiert werden muss. Nichtsdestotrotz ist eine Ermittlung von Strahlgrößen auf der GPU und Einbeziehung in dortige Berechnungen möglich.

5.4.5 Gleitkommadarstellung und Genauigkeit

In einer letzten Untersuchung zum Programm PATRIC sollte das Thema der Genauigkeit der Gleitkommadarstellung und ihrer Auswirkung auf die Programmlaufzeit untersucht werden. Da im Originalprogramm und auch in den bisher vorgestellten GPU-Varianten nach den Vorüberlegungen in Kap. 2.4 mit doppelter Genauigkeit gearbeitet wurde, stellt sich die Frage, welchen Laufzeitvorteil ein Wechsel zu einfacher Genauigkeit bieten würde. Laut Spezifikation der eingesetzten Grafikkarte können Berechnungen mit einfacher Genauigkeit nahezu doppelt so schnell ausgeführt werden, vgl. [NVI09b]. Dieser Geschwindigkeitsvorteil bezieht sich zunächst nur auf die Berechnungen. Darüber hinaus müssen bei einfacherer Genauigkeit insgesamt weniger Daten geladen werden, was auch die Speicherzugriffe leicht verbesssern sollte.

Wie in Tab. 5.6 zu sehen, ergibt sich durch den Einsatz einfacher Genauigkeit insgesamt nur ein Geschwindigkeitsvorteil von weniger als 5%. So gering war dieser nicht erwartet worden; das Ergebnis unterstreicht aber wiederum die Tatsache, dass in den vorliegenden Simulationen die Speicherzugriffe die Berechnungen dominieren und somit durch nahezu doppelt so schnelle Berechnungen insgesamt kein wesentlicher Geschwindigkeitsvorteil erzielt werden kann. Man kann deshalb davon ausgehen, dass bereits mit der aktuellen Grafikkartengeneration eine Nutzung doppelter Genauigkeit keine besonderen Geschwindigkeitsnachteile mit sich bringt. Dies ist für die vorliegenden Simulationen ein erfreuliches Ergebnis, da die bestehenden Datenstrukturen auf der Basis von `double`-Werten beibehalten werden können und keine Umsetzung für die GPU incl. der nötigen Fehleranalysen notwendig werden.

Tabelle 5.6: Teilchen auf der GPU: Vergleich der Laufzeit mit einfacher und doppelter Genauigkeit.

Variante	CPU-Zeit	GPU-Zeit	Summe
Berechnungen mit doppelter Genauigkeit	2,60 *s*	1,49 *s*	4,09 *s*
Berechnungen mit einfacher Genauigkeit	2,54 *s*	1,36 *s*	3,90 *s*

5.5 Kollektive Effekte

5.5.1 Messungen mit dem Originalprogramm LOBO

Nachdem bisher der Fokus auf dem reinen Teilchentransport und der Ermittlung von Strahlgrößen lag, werden im Folgenden kollektive Effekte betrachtet. Im Vordergrund steht dabei der 1D-Fall; dazu wird das in Kap. 3.2 beschriebene Simulationsprogramm LOBO verwendet, da es speziell auf die Betrachtung des longitudinalen Falls zugeschnitten ist. Da das Programm nach dem im Kap. 2.1.2 vorgestellten PIC-Simulationszyklus arbeitet, liegt der Fokus hier auf der Möglichkeit der Parallelisierung der einzelnen Schritte dieses Ablaufs.

Um zu ermitteln, welcher Schritt wieviel Laufzeit beansprucht, wurden zunächst Messungen mit dem Originalprogramm durchgeführt. Dabei wurde ein Gitter der Größe 512 und eine Teilchenzahl von 250.000 gewählt; die Ausgabe des Profilers zum Originalprogramm ist in Abb. 5.10 dargestellt. Deutlich erkennbar ist, dass ein Großteil der Laufzeit des Programms, ca. 85%, auf die Interpolation der Teilchen auf die Gitter entfällt. Dieser Schritt ist deshalb sehr aufwändig, weil hier wieder in Schleifen über die gesamten Teilchen iteriert werden muss, um die Werte auf den Gittern zu aktualisieren. Der `push`-Schritt, der die Teilchen selbst auf Grund des errechneten Raumladungsfelds bewegt, nimmt demgegenüber nur ca. 4% der Laufzeit ein. Interessant ist, dass die hier im `propagate`-Schritt enthaltene Bestimmung des Raumladungsfelds selbst mit 1% nur einen verschwindend geringen Teil der Laufzeit beansprucht.

Aus den in Kap. 5.4.3 vorgenommenen Untersuchungen ist bekannt, dass die Ausgabe von Zwischenergebnissen einen großen Einfluss auf die Laufzeit hat. Dies soll deshalb im Folgenden nicht Gegenstand der Betrachtungen sein. Der Fokus liegt auf den einzelnen Schritten der PIC-Simulation. Das Originalprogramm wurde so abgeändert, dass die Teilchen selbst nicht ausgegeben werden, sondern nur die Gitterinformationen. Weiterhin werden insgesamt selten Werte ausgegeben. Diese Version wurde als Grundlage für die folgenden Modifikationen verwendet.

```
 %   cumulative  self              self    total
time  seconds  seconds    calls  ns/call ns/call name
37.84   14.61   14.61 577200000    0.00    0.00  Grid1D::Pic2Field(..)
34.65   27.98   13.37 192400000    0.00    0.00  Grid2D::Pic2Grid(..)
12.12   32.66    4.68 192300000    0.00    0.00  Grid1D::get_grid_lin(..)
 6.79   35.28    2.62     1924    1.36    1.36  Grid2D::reset()
 2.13   36.10    0.82     1923    0.43    0.43  Beam::push_z(..)
 2.10   36.91    0.81                           Beam::push_dp(..)
 1.04   37.31    0.40                           Beam::propagate(..)
 ...
```

Abbildung 5.10: Ausgabe des Profilers gprof (verkürzt): mehr als ein Drittel der Zeit verbraucht das Originalprogramm LOBO jeweils im Schritt zum Berechnen der Dichteverteilungen und zur Interpolation der Teilchen auf dem Gitter.

Zur Untersuchung des Verhaltens bei unterschiedlicher Datenmenge wurde danach die Laufzeit der CPU-Version bei Variation der Teilchenanzahl gemessen. Wie erwartet ergibt sich hier wieder ein lineares Verhalten, da die Schleifen über die Teilchen eine Laufzeit von O(n) besitzen.

Wie bereits in Kap. 4 bei der Vorstellung der Ansätze in anderen Simulationsprogrammen gesehen, gibt es bei der Einbindung der GPU im Rahmen des PIC-Algorithmus im Wesentlichen zwei Ansätze. Ein Ansatz ist es, die Gitterinformationen direkt von den Teilchen aus zu aktualisieren. Hierbei sind ebenso viele Speicherzugriffe nötig, wie Teilchen bearbeitet werden. Da der Zugriff auf die Gitterzellen konkurrierend erfolgt, müssen diese Speicherzugriffe synchronisiert werden, was in diesem Ansatz den Hauptaufwand erzeugt. In einem zweiten Ansatz werden die Informationen auf den Gittern jeweils blockweise mit vorsortierten Teilchen ermittelt, weshalb hierbei nur so viele konkurrierende Speicherzugriffe entstehen, wie es Gitterpunkte gibt. Bei diesem Ansatz liegt der Aufwand in der nötigen Sortierung der Teilchen. Im Folgenden werden diese beiden unterschiedlichen Ansätze bei der Portierung des Programms LOBO auf die GPU verfolgt und gegenübergestellt.

5.5.2 Interpolation mit atomaren Operationen

Wie in der Vorstellung des Programms LOBO in Kap. 3.2 zu sehen, werden in der zentralen Schleife des Programms die einzelnen Schritte der PIC-Simulation in Form von Funktionen aufgerufen. Entsprechend ist bereits die Struktur für eine Funktionszerlegung vorhanden. Als Daten, mit denen die einzelnen Funktionen operieren, existieren die Teilchendaten sowie die Gitterdaten. Typischerweise ist hierbei die Anzahl der Teilchen um 1-2 Größenordnungen höher, als die Anzahl der Gitterzellen, weshalb später entsprechend pro Teilschritt der Simulation ggf. unterschiedlich viele Ausführungseinheiten benötigt werden. Da, wie in der Ausgabe des Profilers deutlich geworden, vor allem die Interpolation der Teilcheninformationen auf die Gitter in der Originalversion die meiste Laufzeit beansprucht, wurde der Fokus auf die Portierung dieser Schritte auf die GPU gelegt; die Bestimmung des Raumladungsfeldes wurde auf der CPU belassen.

Nach den Überlegungen in Kap. 5.4 wird wieder der Ansatz gewählt, bei der Bearbeitung der Teilchen jeweils mit einem Thread pro Teilchen zu arbeiten. D. h. bei der Interpolation der Teilchen auf die Gitter und umgekehrt werden jeweils so viele Threads wie Teilchen gestartet. Bei einer erneuten Interpolation müssen jeweils die Gitterinformationen zunächst wieder auf null gesetzt werden. Da danach jeweils eine globale Synchronisation nötig ist, wird dieser Initialisierungsschritt mit Hilfe eines eigenen Kernels durchgeführt. Die konkurrierenden Speicherzugriffe von allen Threads aus werden mit Hilfe atomarer Operationen synchronisiert. Da für den Datentyp `double` keine entsprechende Funktion in den CUDA-Spracherweiterungen vorhanden ist, wurde wieder die von NVIDIA in den CUDA Programmierrichtlinien vorgeschlagene Funktion `atomicAdd(double* address, double val)` verwendet [vgl. NVI13b]. Der Quelltext dieser Variante ist in Anhang A.3.7 gezeigt.

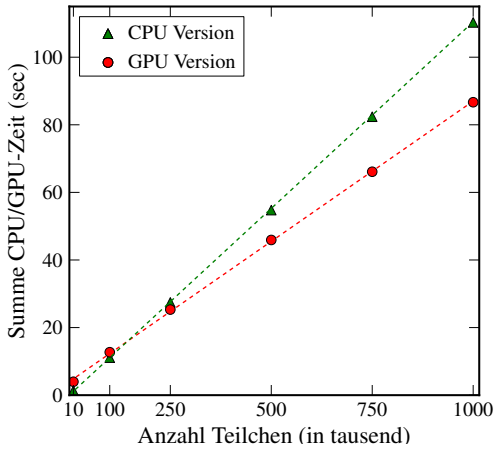

Abbildung 5.11: Messung der modifizierten LOBO Version, in der die Interpolationsschritte auf die GPU portiert wurden. Die Aktualisierung der Gitterinformationen erfolgt mit atomaren Operationen.

Analog zu den Messungen mit dem Programm PATRIC wird zumindest für die GPU-Variante ebenfalls wieder ein lineares Verhalten mit Variierung der Teilchenzahl erwartet. Da die Speicherzugriffe auf die Gitterdaten von allen Teilchen aus erfolgen und synchronisiert werden müssen, wird in diesem Schritt auf der GPU der Flaschenhals erwartet. Zunächst kann die Laufzeit gegenüber der CPU-Variante vorher schlecht abgeschätzt werden, da vor allem die Synchronisierung zeitaufwändig ist.

Wie in Abb. 5.11 zu sehen, ist die GPU-Variante erst ab ca. 170.000 Teilchen schneller als die CPU-Variante. Insgesamt weist die GPU-Version eine Beschleunigung von 1,19 auf. Dass die GPU-Variante bei einer größeren Teilchenanzahl leichte Vorteile bietet, ist vor allem auf die Möglichkeit der Grafikkarte zurückzuführen, dass andere Threads abgearbeitet werden können, während auf einen Speicherzugriff gewartet werden muss. Um genauer zu identifizieren, wo bei der GPU-Variante die meiste Zeit verbraucht wird, wurde für den Fall mit 250.000 Teilchen die Laufzeit genauer analysiert und den einzelnen Schritten im PIC-Simulationszyklus zugeordnet. Wie Tab. 5.7 zeigt, ist bei der GPU-Implementierung vor allem der Schritt zur Interpolation der Teilchen auf die Gitter der, der mit ca. 67% den größten Anteil der Laufzeit auf der GPU beansprucht. Dass hier vor allem der konkurrierende Schreibzugriff den Flaschenhals darstellt, wird im Vergleich mit dem Lesezugriff auf die Gitter bei der Interpolation auf die Teilchen und deren Fortbewegung deutlich.

Tabelle 5.7: Teilchen und Gitter auf der GPU: Laufzeiten der einzelnen Schritte. Die GPU-Zeit ist aufgeteilt in den Schritt zur Interpolation auf die Gitter, Interpolation auf die Teilchen incl. deren Fortbewegung sowie in Berechnungen zu Ausgabezwecken.

Variante	CPU-Zeit	GPU-Zeit	Gitter	Teilchen	Ausgabe
GPU mit atomaren Operationen	3,57 *s*	21,24 *s*	14,83 *s*	2,65 *s*	3,76 *s*

Um herauszufinden, in wiefern dabei die Gitter selbst eine Rolle spielen, wurde die Teilchenanzahl bei 250.000 festgehalten und die Gittergröße variiert. Gewählt wurden dabei in den Simulationen typische Gittergrößen; das Ergebnis ist in Tab. 5.8 dargestellt. Deutlich zu erkennen ist der lineare Zusammenhang zwischen Gittergröße und Laufzeit bei der CPU-Variante. Bei der GPU-Variante scheint dieser für größere Gitter ebenfalls gegeben zu sein. Dass die Laufzeit für eine Gittergröße von 256 nicht weiter sinkt, liegt daran, dass bei kleinen Gittern die Anzahl der konkurrierenden Zugriffe sehr hoch wird.

Tabelle 5.8: Teilchen und Gitter auf der GPU: Vergleich der Laufzeit mit unterschiedlichen Gittergrößen.

Variante / Gittergröße	**256**	**512**	**1024**
CPU-Variante	$13,49\ s$	$27,44\ s$	$56,74\ s$
GPU-Variante	$22,38\ s$	$24,81\ s$	$53,16\ s$

Zusammenfassend ist für die Variante, bei der die Interpolation auf die Gitter mit Hilfe atomarer Operationen erfolgt, festzustellen, dass eine solche Implementierung nur leichte Vorteile gegenüber der ursprünglichen CPU-Variante bietet. Zudem hängt es auch von der Teilchenzahl und Gittergröße ab; hierbei ist die GPU-Variante erst ab 170.000 Teilchen und ab einer Gittergröße von 512 zu empfehlen.

5.5.3 Interpolation mit Vorsortierung

Wie in Kap. 5.5.2 gezeigt, ist vor allem der Schritt der Interpolation der Teilchen auf die Gitter durch die konkurrierenden Schreibzugriffe laufzeitintensiv. Als Alternative existiert wie bereits in Kap. 4 gesehen die Strategie, die Teilchen bei jedem Fortbewegen im push-Schritt nach ihrer neuen Zuordnung zu Gitterzellen neu zu sortieren. Damit erreicht man, dass die einzelnen Gitterzellen bei der Interpolation der Teilchen unabhängig voneinander bearbeitet werden können. Jeder Thread ist in dieser Variante für eine Gitterzelle zuständig und bearbeitet nun eine Menge von Teilchen, die dieser Gitterzelle zugeordnet sind.

Bei der Implementierung wurde zunächst bei den Teilcheninformationen ein zusätzliches Array geschaffen, das nach jedem Fortbewegen der Teilchen aktualisiert wird und pro Teilchen den Index der zugeordneten Gitterzelle enthält. Zwei weitere neue Arrays in der Größe des Gitters geben an, ab welchem Teilchenindex sich Teilchen für die jeweilige Gitterzelle befinden und wie viele Teilchen dieser Gitterzelle zugeordnet sind. Beide Informationen können nun im modifizierten Kernel `kernel_line_current_density_block` dazu verwendet werden, alle dieser Gitterzelle zugeordneten Teilchen zu bearbeiten. Pro Gitterzelle werden am Ende des Kernels die ermittelten Werte für den linken und rechten Gitterpunkt mit Hilfe von atomaren Operation aktualisiert. Insgesamt sind so $2 \cdot k$ atomare Operationen nötig, wobei k die Größe des Gitters darstellt. Die Sortierung selbst und das Ermitteln der Indizes wurde ähnlich wie in einer GPU-Variante des Programms ELEGANT mit

den Thrust-Funktionen `thrust::sort_by_key` und `thrust::lower_bound` realisiert, vgl. auch Abb. 4.1. Der Quelltext ist in Anhang A.3.8 dargestellt. Bei dieser Variante wird erwartet, dass die Interpolation der Teilchen auf die Gitter selbst schneller abläuft, jedoch hierbei die als Teil der Teilchenbewegung realisierte Sortierung der Teilchendaten wiederum Laufzeit kostet.

Wie in Tab. 5.9 zu sehen, steigt in dieser Variante die Laufzeit sehr stark an und ist für 100.000 Teilchen ca. 8-mal langsamer als die CPU-Variante. Dies wird vor allem darauf zurückgeführt, dass in der vorgenommenen Implementierung pro Gitterzelle ein Thread gestartet wurde, also insgesamt mit 512 Threads die gesamten Teilchen bearbeitet wurden, was einer optimalen Auslastung der GPU entgegenwirkt. Dies, zusammen mit der Sortierung nach jedem Schritt führt insgesamt zu einer deutlich höheren Laufzeit. Würde man diesen Ansatz weiter verfolgen wollen, müsste man Maßnahmen ergreifen, um eine weitere Unterteilung der Teilchen zu erreichen. Dies würde aber im Gegenzug wieder zu einer nötigen Synchronisierung der Threads führen. Es ist also festzustellen, dass dieser Ansatz mit Vorsortierung für das vorliegende Programm und die gewählten Teilchen- und Gittergrößen keine Vorteile bietet.

Tabelle 5.9: Teilchen und Gitter auf der GPU: Vergleich der Laufzeit mit einer Sortierung der Teilchendaten nach jeder Teilchenbewegung.

Variante	CPU-Zeit	GPU-Zeit	Summe
CPU-Variante (zum Vergleich)	$11,10\ s$	$0,00\ s$	$11,10\ s$
GPU-Variante mit Sortierung	$9,07\ s$	$80,14\ s$	$89,21\ s$

5.6 Integration der vorgenommenen Parallelisierungen

In den bisherigen Betrachtungen zur Einbeziehung der GPU erfolgte zuerst in Kap. 5.4 eine Konzentration auf die transversale und danach in Kap. 5.5 auf die longitudinale Teilchenbewegung und die jeweiligen Möglichkeiten zur Parallelisierung. Die Ergebnisse haben gezeigt, dass sich das reine Tracking der Teilchen gut parallelisieren lässt, da die Teilchen in der Berechnung unabhängig voneinander betrachtet werden können. Die Betrachtung der kollektiven Effekte hat jedoch gezeigt, dass diese schwer zu parallelisieren sind, da die Berechnung der Kräfte der Teilchen untereinander und in der Verbindung mit dem umgebenden Beschleuniger es bedingen, dass viele parallele Threads schreibend und lesend auf die selben Gitterdaten zugreifen müssen. Hierzu sind entweder geeignete Synchronisierungsmechanismen beim Speicherzugriff oder eine entsprechende Vorsortierung der Teilchen nötig, um mit den entstehenden Zugriffskonflikten umzugehen oder diese zu vermeiden, was einer Parallelisierung entgegensteht.

Die Simulationen in der Abteilung Strahlphysik konzentrieren sich entweder analog zu den Betrachtungen bisher speziell auf die transversale oder longitudinale Bewegung, oder es wird die gesamte Teilchenbewegung betrachtet. In einer umfangreicheren Version von PATRIC sind beide Betrachtungen enthalten. Hierbei wechseln sich in der

Berechnung im Rahmen der Simulation die Bewegung der Teilchen auf Grund von externen Kräften mit der Bewegung auf Grund von kollektiven Effekten ab. Es muss also abwechselnd ein Tracking mit Hilfe der Matrizen durchgeführt werden und danach jeweils die Bestimmung der Raumladung mit Rückwirkung auf die Teilchen. Eine solche integrierte Simulation kann aufbauend auf den im Rahmen der Arbeit entwickelten Mechanismen durchgeführt werden. Dazu müssen beide GPU-Implementierungen in einem Programm zusammengefasst werden, wozu ggf. die Datenstrukturen entsprechend anzupassen sind. Für eine Realisierung des Schritts der Bestimmung des Raumladungsfelds selbst existiert mit cuFFT eine mächtige und auch performante Bibliothek, die in einer solchen Variante komplett auf der GPU zum Einsatz kommen sollte. Aus Messungen anderer Arbeiten entspricht hierbei die Laufzeit in etwa der einer FFT auf der CPU. Erwartet wird insgesamt für eine Implementierung der gesamten Teilchenbewegung auf der GPU eine leichte Laufzeitverbesserung, die durch die 6-fache Beschleunigung des Trackings herrührt.

Da die angesprochene erweiterte Version von PATRIC mit 2D-Gittern für das Raumladungsfeld arbeitet, muss entsprechend die Interpolation der Teilchen auf die Gitter und umgekehrt mit einer zweidimensionalen Datenstruktur zur Repräsentation der Gitter arbeiten. Obwohl einem Teilchen dadurch vier Gitterpunkte im 2D-Raum zugeordnet sind und somit doppelt so viele Berechnungen bei der Interpolation nötig sind, war bisher zu verzeichnen, dass die Berechnungen hier nicht der Flaschenhals sind und deshalb kaum ins Gewicht fallen. Durch die 2D-Gitter wird statt dessen ein Vorteil für die Variante mit atomaren Operationen erwartet, da auf die einzelnen Gitterpunkte insgesamt weniger Teilchen entfallen als bei einer 1D-Projektion und somit weniger Konflikte beim Speicherzugriff zu erwarten sind. Für die Variante mit einer Vorsortierung der Teilchen ist hingegen eine längere Laufzeit als im 1D-Fall zu erwarten, da die Sortierung selbst mit entsprechend einer quadratischen Anzahl an Gitterzellen arbeiten muss. Da zu erwarten ist, dass die atomaren Operationen in zukünftigen Grafikkartengenerationen schneller durchführbar sind, spricht dies tendenziell für eine weitere Verfolgung dieser Programmvariante.

Eine Umsetzung der gesamten Teilchenbewegung auf der GPU war nicht mehr Teil dieser Arbeit, da hier die grundsätzlichen Möglichkeiten und Grenzen der Einbeziehung der GPU untersucht werden sollten. Als Ausblick auf zukünftige Entwicklungen kann insgesamt jedoch festgestellt werden, dass sich mit den erarbeiteten Mitteln eine Simulation der gesamten Teilchenbewegung auf der GPU realisieren lässt und dies voraussichtlich einen Laufzeitvorteil bieten wird.

5.7 Einbeziehung mehrerer MPI-Knoten

Im Rahmen der Arbeit wurde die Möglichkeit der Einbeziehung einer lokal vorhandenen GPU in die Simulationsprogramme untersucht. Um sich auf dieses Thema zu konzentrieren wurde für das Programm PATRIC die bereits im Programm vorhandene verteilte Ausführung mit Hilfe von MPI ausgeklammert und nur mit einem Berechnungsknoten gearbeitet. Im Folgenden soll dieses Thema kurz wieder aufgegriffen

werden, obwohl im Rahmen der Arbeit kein System für die verteilte Rechnung mit MPI und lokalen GPUs zur Verfügung stand. Aktuell werden jedoch in der Helmholtz-Gemeinschaft Deutscher Forschungszentren mehrere Kompetenzzentren für die GPU-Entwicklung aufgebaut und die Rechenzentren sollen in Zukunft eine solche hybride Ausführung von Programmen unterstützen. Aus diesem Grund sollen an dieser Stelle erste Hinweise zur Einbeziehung von Grafikprozessoren und MPI gegeben werden.

Speziell bei der verteilten Berechnung auf mehrere MPI-Knoten ist zunächst wichtig, dass bei der Datenzerlegung (vgl. Kapitel 2.2.4) beachtet wird, dass die Arbeit auf den einzelnen MPI-Knoten den Kommunikationsaufwand übersteigen muss, um effizient zu sein. Entsprechend dürfen die Anzahl der Knoten nicht zu groß bzw. der Anteil an den Teilchendaten, den ein Knoten zu bearbeiten hat, nicht zu klein werden. Da das Verhältnis Rechenzeit zu Kommunikationszeit (vgl. 2.2.6) von dem verwendeten System, dessen Knoten und Netzwerken abhängt, kann zunächst nur betont werden, dass das Verhältnis bei den verwendeten Größen von Teilchenmengen und Knotenanzahlen größer eins sein sollte. Wie in Kap. 5.4.2 zu sehen, war selbst bei dem gut parallelisierbaren Problem des Teilchentransports eine Einbeziehung der GPU erst ab einer Größenordnung von 10.000 Teilchen interessant, bei den kollektiven Effekten sogar erst ab 200.000. Diese Zahlen sollten bei einer Datenzerlegung für die Berechnungsknoten beachtet werden. Hier muss kritisch hinterfragt werden, ob auf Grund der Teilchenzahlen wirklich mehr als nur ein paar wenige Berechnungsknoten eingebunden werden können und ob sich vor diesem Hintergrund der Programmieraufwand für eine hybride Implementierung tatsächlich lohnt.

Möchte man MPI einbeziehen, erscheinen Teile des Algorithmus unkritisch, bei denen nur die Gitterdaten kommuniziert werden müssen, da hier die zu verteilende Datenmenge im Vergleich zu den gesamten Teilchendaten gering ist. Bei MPI gibt es effiziente globale Reduzieroperationen, die dabei zum Einsatz kommen können, so dass ein solcher Ansatz bei den hier vorliegenden Gittergrößen empfehlenswert ist.

Problematisch erscheinen Auswertungen, die sich nicht aufteilen lassen und bei denen die gesamten Teilchendaten benötigt werden, oder Konfigurationen, bei denen eine häufige Kommunikation zwischen den Knoten nötig wird. In erstere Kategorie fallen Berechnungen der Strahleigenschaften. Eine Kommunikation zwischen den Knoten wird nötig, wenn Teilchen am Rande einer Scheibe eines Teilchenpakets sich zwischen zwei benachbarten Scheiben hin- und herbewegen: diese Teilchen müssen dann zwischen den beiden MPI-Knoten, die für die benachbarten Teilchenpaketscheiben zuständig sind, ausgetauscht werden.

Generell ist der zusätzliche Aufwand bei der Einbeziehung der lokalen GPU vor dem Hintergrund zu betrachten, dass Daten nicht nur kommuniziert, sondern auch mit der lokalen GPU ausgetauscht werden müssen. In diesem Zusammenhang sollte man, sobald ein solches System zur Verfügung steht, aktuelle MPI-Frameworks evaluieren, die es ermöglichen, Daten direkt zwischen den GPUs der verteilten Knoten auszutauschen. Dies geschieht, indem direkt Zeiger auf die Daten auf der GPU in den MPI-Befehlen verwendet werden können. Aktuell gehören dazu die MPI-Frameworks MVAPICH2, OpenMPI und IBM Platform MPI.

6 Diskussion

6.1 Zusammenfassung und Bewertung der Ergebnisse

Im Rahmen der Arbeit wurden die Möglichkeiten einer Parallelisierung der vorhandenen Simulationsprogramme PATRIC und LOBO mittels GPU-Programmierung untersucht. Auf Grund der modularisierten Ausgangsprogramme, die bereits eine Funktionszerlegung aufwiesen, war es dabei gut möglich, einzelne Funktionen schrittweise auf die GPU zu portieren.

Basierend auf einer vereinfachten Version des Programms PATRIC erfolgte zunächst eine Konzentration auf den Transport der Teilchen und die Ermittlung von Strahlgrößen. Durch die Untersuchungen konnte gezeigt werden, dass der Einsatz von GPUs für einzelne kleine Teile des Algorithmus (hier: der einzelne Transportschritt) kaum Vorteile gegenüber einer reinen CPU-Variante bietet, da der Aufwand zur Einbeziehung der GPU der wesentlich höheren Berechnungsperformanz der Grafikkarte entgegenwirkt. Sobald die Daten jedoch länger auf der GPU gehalten werden können und viele Berechnungen auf ihnen durchgeführt werden können, zeigt sich deutlich ein Laufzeitvorteil. Bei der Variante mit einem Teilchen pro Thread und dem Halten der Daten möglichst lange auf der GPU konnte so eine 6-fache Beschleunigung erzielt werden. Die Divergenz der Threads auf Grund von Teilchenverlusten hat darüber hinaus keinen negativen Einfluss auf die Laufzeit.

Das bei einem Halten der Teilchen auf der GPU auftretende Problem, dass die Daten zum Ausgeben von Zwischenergebnissen wieder in regelmäßigen Abständen zum Host zurückkopiert werden müssen, wurde in verschiedenen Messungen quantifiziert. Ab einer Ausgabehäufigkeit von 16 Transportmatrizen (entsprach hier einem Umlauf durch den Beschleuniger) war eine Ausgabe von Zwischenergebnissen gut möglich; werden Daten seltener als alle 80 Transportmatrizen (bzw. hier fünf Umläufe) ausgegeben, ergab sich durch seltenere Ausgaben kein Laufzeitvorteil mehr. Durch eine Überlappung des Kopiervorgangs mit der nächsten Berechnung mit Hilfe von Streams konnte nur ein geringer Laufzeitvorteil erzielt werden, da in den vorliegenden Simulationen der Anteil der Berechnungen im Vergleich zu den Speicherzugriffen sehr gering ist.

Hält man die Teilchen auf der GPU, stellt sich die Frage, ob dort auch Strahlgrößen ermittelt werden können. Eine solche Ermittlung kann zur Reduktion der Datenmenge beim Zurückkopieren auf den Host oder als Teil von Berechnungen oder Abbruchbedingungen auf der GPU selbst dienen. Im Rahmen der Arbeit wurde gezeigt, dass diese Art der Berechnungen auf der GPU prinzipiell möglich sind, davon allerdings

aus mehreren Gründen abzuraten ist. Trotz der Reduktion der Datenmenge beim Kopieren wird durch die Emittanzermittlung auf der GPU kein Laufzeitvorteil erreicht, da dieser durch die nötigen Berechnungen auf der GPU wieder aufgezehrt wird. Darüber hinaus führt eine Ermittlung solcher Größen zu spezialisiertem Code, der i. A. schlecht wartbar erscheint. Muss dennoch als Teil einer Berechnung eine solche Größe auf der GPU ermittelt werden, sollte von Eigenimplementierungen abgesehen werden, da Reduktionsoperationen über die gesamten Teilchendaten laufzeitintensiv sind. Stattdessen sollten die stark optimierten Thrust-Funktionen für Operationen über alle Teilchen eingesetzt werden. Allerdings führen auch diese insgesamt zu keinem Laufzeitvorteil gegenüber der CPU. Es ist deshalb davon auszugehen, dass die bestehenden Auswerteroutinen auf der CPU weiterhin genutzt werden können.

Im Rahmen der Untersuchungen zu PATRIC wurde als Nebenthema bei einem Laufzeitvergleich mit einfacher und doppelter Genauigkeit für die einfache Genauigkeit lediglich eine um 5% schnellere Laufzeit festgestellt. Dies ist ein gutes Ergebnis, da die vorhandenen mit doppelter Genauigkeit arbeitenden Routinen nicht abgeändert werden müssen, was weitere Fragen nach möglichen Fehlern und deren Fortpflanzung aufgeworfen hätte.

Basierend auf einer Version des Programms LOBO wurden die Parallelisierungsmöglichkeiten bei der Berechnung der kollektiven Effekte untersucht. Hierbei ist festzustellen, dass von den Schritten im Rahmen der PIC-Simulation vor allem die Interpolation der Teilchen auf die Gitter schwierig zu parallelisieren ist, da die konkurrierenden Schreibzugriffe auf die Gitter einen Flaschenhals darstellen. Die beiden in anderen Forschungsarbeiten zum Thema vorgeschlagenen Implementierungsvarianten wurden für den vorliegenden Fall umgesetzt und miteinander verglichen. Da die Variante, bei der die Teilchen nicht sortiert werden und die Gitterdaten mit atomaren Operationen aktualisiert werden, einen leichten Vorteil gegenüber der CPU-Variante bieten, sollte dieser Ansatz für zukünftige Entwicklungen weiter verfolgt werden. Dafür spricht auch, dass atomare Operationen mit neueren Grafikkartengenerationen besser unterstützt werden und zu erwarten ist, dass diese in Zukunft immer schneller durchführbar sind. Die alternative Variante mit einer Vorsortierung der Teilchen war im Vergleich zu langsam, da hier die Parallelisierungsmöglichkeiten der GPU nicht gut genug ausgenutzt werden konnten.

Zusammenfassend lässt sich feststellen, dass für gut parallelisierbare Probleme durch den Einsatz der GPU eine gute Laufzeitverbesserung zu erzielen ist. In der vorliegenden Implementierung konnte für das Tracking der Teilchen eine 6-fache Beschleunigung erzielt werden. Bei Berechnungen, bei denen Werte über die gesamten Teilchen gebildet werden müssen oder eine Synchronisation vieler Threads nötig wird, ist der Nutzen der GPU beschränkt. Da in den schnellsten vorgeschlagenen Implementierungen die GPU-Variante jedoch auch hierbei einen leichten Vorteil verspricht, steht insgesamt einem Einsatz für die Simulation der kompletten Teilchenbewegung nichts im Weg. Für eine solche Umsetzung können die im Rahmen der Diplomarbeit entstandenen Bausteine als Grundlage dienen.

6.2 Ausblick

Wie im Rahmen der Diplomarbeit gesehen, können Grafikkarten auf Grund ihrer heute vorhandenen universellen Programmierschnittstellen bereits sehr gut zur Parallelisierung eingesetzt werden. Da solche Grafikkarten leicht einsetzbar sind, bestehen heutzutage wesentlich bessere Parallelisierungsmöglichkeiten, da man nicht mehr auf die Existenz von dedizierten Parallelrechnern angewiesen ist. Der Nachteil hierbei ist, dass die Programmierschnittstellen noch nicht standardisiert sind, und somit in Zukunft ggf. Anpassungsaufwand entstehen wird. Im Allgemeinen kann hierbei geraten werden, den Quelltext für die GPU trotz eventueller Performanzeinbußen leicht lesbar zu halten und im Idealfall einer bereits vorhandenen Funktionszerlegung zu folgen. Zu sehr auf die jeweilige GPU angepasster und optimierter Quelltext kann dabei leicht zu einer deutlich schlechteren Wartbarkeit führen.

Bei den anderen betrachteten Programmen wie ELEGANT oder PIConGPU wird in zunehmendem Maße auf Bibliotheken aufgesetzt, um von der Komplexität der Grafikkarte zu abstrahieren. Dies ist für kleinere Projekte so kein gangbarer Weg, da die Ressourcen zum Aufbau solcher Bibliotheken nicht vorhanden sind. Dennoch kann hier erwartet werden, dass es zukünftig evtl. in diesem Umfeld frei verfügbare Bibliotheken geben wird, auf die dann statt dessen aufgesetzt werden kann. Es ist davon auszugehen, dass mit Hilfe solcher Bibliotheken die Einbeziehung der GPU einfacher und schneller möglich wird.

Als zukünftigen Trend werden in der Literatur hybride Modelle favorisiert, bei denen die vorhandenen Rechenressourcen – sei es nun die CPU, GPU oder evtl. vorhandene Parallelrechnersysteme – gemischt eingesetzt werden [vgl. SK10, S. 8]. Hier ist allerdings fraglich, ob die dadurch entstehende Komplexität den zu erwartenden Performanzgewinn tatsächlich rechtfertigt. Gerade vor dem Hintergrund der hier in den Simulationen verwendeten eher geringeren Datenmengen stellt sich die Frage, ob evtl. die Konzentration auf ein oder zwei Parallelisierungstechniken nicht insgesamt einen guten Kompromiss aus Wartbarkeit und Laufzeitgewinn darstellt. Vor diesem Hintergrund müssen in näherer Zukunft zu erwartende hybride Ansätze mit MPI und GPUs entsprechend bewertet werden. Die mit einer hybriden Lösung verbundenen Problematiken wurden im Rahmen dieser Arbeit nur in Form eines kurzen Ausblicks angesprochen. An dieser Stelle können in Zukunft weitere Untersuchungen ansetzten, sobald solche Systeme zur Verfügung stehen. Es ist zu erwarten, dass hier auch in Zukunft noch mehr Unterstützung durch die Hersteller erfolgt; so bietet die Entwicklungsumgebung nsight von NVIDIA ab CUDA 5.5, das aktuell als Vorabversion vorliegt, das Debugging im gemischten MPI-GPU-Umfeld. Das Thema der Verwendung aller vorhandener Hardware zur Parallelisierung und damit ein Übergang hin zu Mischlösungen wird also zukünftig das Hauptthema sein.

Speziell für die Einbeziehung der GPU zeichnet sich ein Trend zu einer stärkeren Integration mit der CPU ab, obwohl die Hersteller hierbei unterschiedliche Wege gehen. AMD integriert mit der im Rahmen des Projekts Fusion entstandenen APU (Accelerated Processing Unit) die GPU auf dem Prozessorchip der CPU. Beide Prozessoren

haben so Zugriff auf denselben Hauptspeicher [vgl. Adv13]. Ein ähnlicher Weg wird aktuell bei Intel verfolgt, auch hier werden GPUs mit der CPU integriert. NVIDIA demgegenüber geht mit dem Projekt Denver in die Richtung, leistungsfähige ARM-Prozessoren direkt auf der GPU zu integrieren, wobei auch in Zukunft größere und leistungsfähigere GPUs als Grundlage dienen können. Auf diesen Prozessoren kann wiederum direkt das Betriebssystem laufen [vgl. Koe13]. Unabhängig von der konkreten Lösung wird also das Bestreben deutlich, in Zukunft eine Zusammenführung der CPU und GPU zu erreichen. Dies wird zum einen eine Vereinfachung der Programmierung bedeuten, da die Daten nicht mehr explizit zwischen Host und Device umkopiert werden müssen, zum anderen sind durch den Wegfall der Datentransfers Laufzeitverkürzungen zu erwarten. Davon können voraussichtlich die vorliegenden Simulationen profitieren, da bei diesen vor allem der Speicherzugriff den Flaschenhals darstellt. Aus diesem Grund sollte der Markt in Zukunft genau beobachtet werden.

Anhang

A Programmierung

A.1 Übersicht über die CUDA C Spracherweiterungen

Zum Verständnis der im Rahmen der Arbeit gezeigten Quelltextauszüge erfolgt hier eine kurze Übersicht der Schlüsselwörter der CUDA C Spracherweiterung, die zum Nachschlagen der Bedeutung einzelner Befehle verwendet werden kann. Eine vollständige Liste aller Spracherweiterungen findet sich in [NVI13b].

Kennzeichnung für Funktionen:

`__global__`	ausgeführt auf dem Device, vom Host aus aufrufbar
`__device__`	ausgeführt auf dem Device, vom Device aufrufbar
`__host__`	ausgeführt auf dem Host, vom Host aufrufbar
`__device__` und `__host__`	jeweils eine Funktion für Host und Device

Kennzeichnung für Variablen (spezifiziert den Speicherort):

`__device__`	im globalen Speicher, nicht gecacht, Zugriff: Host, alle Threads
`__constant__`	im konstanten Speicher, gecacht, Zugriff: Host, alle Threads
`__shared__`	im gemeinsamen Speicher, Zugriff: alle Threads eines Blocks

Speicherallokation, Zugriffe:

- `cudaMalloc(void ** pointer, size_tnbytes)`
- `cudaFree(void* pointer)`
- `cudaMemcpy(void *dst, void *src, size_tnbytes, direction)`
  Mögliche Richtungen: `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`

Synchronisation aller Threads in einem Block:

```
void __syncthreads();
```

Aufruf eines Kernels, Start einer GPU-Prozedur vom Host aus:

```
kernel<<<dim3 grid, dim3 block>>>(...)
```

In spitzen Klammern wird die „Ausführungskonfiguration“ angegeben. Die Dimension des Grids und Thread-Blocks erfolgt dabei jeweils in 3d, d. h. x, y, z.

A.2 Zeitmessung

Für die Zeitmessung wurde auf der CPU mittels `clock_gettime` die Prozessorzeit ermittelt. Für die GPU wurde mit CUDA Events gearbeitet, dazu kamen die Befehle `cudaEventCreate(&start)` und `cudaEventRecord(start, 0)` zum Einsatz. Als Konfiguration wurde hierbei die blockierende Synchronisation mit dem Host gewählt. Anstatt aktiv zu Warten kann bei dieser Konfiguration die CPU in der Zwischenzeit anderweitig genutzt werden, während die Berechnung auf der Grafikkarte läuft. Außerdem lassen sich so die gemessenen Zeiten der CPU und GPU besser voneinander trennen, da bei aktiven Warten ja auch die CPU-Zeit hochgezählt würde. Deshalb wird diese Konfiguration für Zeitmessungen empfohlen, obwohl sich hierdurch eine leicht erhöhte Laufzeit ergibt (in einer durchgeführten Vergleichsmessung wurde eine um ca. 2.53% verlängerte Laufzeit gemessen, 15.73 *s* zu 16.13 *s*). Die beiden folgenden Quelltextauszüge zeigen eine Zeitmessung für die CPU und für die GPU.

Quelltext A.1: Zeitmessung der CPU-Programmausführung.

```
timespec start, stop;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
// Calculation here..
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &stop);
timespec temp;
if ((stop.tv_nsec - start.tv_nsec) < 0) {
  temp.tv_sec = stop.tv_sec - start.tv_sec - 1;
  temp.tv_nsec = 1000000000 + stop.tv_nsec - start.tv_nsec;
} else {
  temp.tv_sec = stop.tv_sec - start.tv_sec;
  temp.tv_nsec = stop.tv_nsec - start.tv_nsec;
}
double recordedTime = temp.tv_sec + temp.tv_nsec / 1000000000.0;
fprintf(stdout, "CPU time: %.8lf", recordedTime);
```

Quelltext A.2: Zeitmessung der GPU-Programmausführung.

```
cudaEvent_t start, stop;
cudaEventCreateWithFlags(&start, cudaEventBlockingSync);
cudaEventCreateWithFlags(&stop, cudaEventBlockingSync);
cudaEventRecord(start, 0);
kernel_singleParticle<<<dimGrid, dimBlock>>>(d_T, d_particleVectors,
    numberOfParticles);
cudasafe(cudaPeekAtLastError(), "Error in multiplication kernel");
cudasafe(cudaDeviceSynchronize(), "Error in multiplication kernel");
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float gpuTimeInSecs = 0;
cudaEventElapsedTime(&gpuTimeInSecs, start, stop);
gpuTimeInSecs = gpuTimeInSecs / 1000;
  fprintf(stdout, "GPU time: %.8lf", recordedTime);
```

A.3 Programmauszüge

Im Folgenden sind einige Quelltextauszüge wiedergegeben, die die beschriebenen Programmmodifikationen verdeutlichen sollen. Auch hierbei handelt es sich wiederum nur um (ggf. gekürzte) Quelltextauszüge. Die vollständigen Programmversionen aller implementierten Varianten sind der Arbeit auf CD beigefügt.

A.3.1 PATRIC: Transportschritt auf der GPU

Bei der Portierung des Transportschritts in PATRIC auf die GPU wurden zwei Varianten implementiert. Bei der ersten Variante wird pro Thread eine Koordinate eines Teilchens berechnet. Bei der zweiten Variante wird pro Thread auf der GPU genau ein Teilchen mit allen Koordinaten bearbeitet. Die Teilchendaten selbst werden in einem großen Array gehalten. Gezeigt ist hier beispielhaft die erste Variante der Implementierung. In der zweiten Variante werden entsprechend alle Koordinaten nacheinander berechnet.

Quelltext A.3: PATRIC: Transportschritt auf der GPU, Berechnung der Einzelkoordinaten.

```
1  __global__ void kernel_singleCoordinate(const double *transportMatrix,
       double *particleVectors, int numberOfParticles) {
2
3    // We work on one particle vector element (one coordinate)
4    int coordinateToCalculate = threadIdx.x + blockDim.x * blockIdx.x;
5
6    // Calculate the new coordinate value
7    double value = 0.0;
8    if (coordinateToCalculate < (numberOfParticles * 6)) {
9      for (int i = 0; i < 6; i++) {
10       value += transportMatrix[(coordinateToCalculate % 6) * 6 + i]
11                 * particleVectors[coordinateToCalculate - (
                       coordinateToCalculate % 6) + i];
12     }
13   }
14   __syncthreads();
15
16   // Update the coordinate with the new value
17   if (coordinateToCalculate < (numberOfParticles * 6)) {
18     particleVectors[coordinateToCalculate] = value;
19   }
20 }
```

A.3.2 PATRIC: Transportschritt auf der GPU, Variante mit konstantem Speicher

Wenn die Transportmatrizen nicht im globalen Speicher gehalten werden, sondern im etwas schnelleren konstanten Speicher, muss entsprechend die Deklaration und das Befüllen des Speichers abgeändert werden. Der konstante Speicher kann nur zum Kompilierzeitpunkt allokiert werden, weshalb bereits im Quelltext die Größe festgelegt werden muss. Beispielhaft wurde dies für eine Transportmatrix durchgeführt, hier sind die entsprechend nötigen Änderungen im Quelltext dargestellt. Auf Grund der Version der Grafikkarte und des verwendeten Treibers war es hierbei nur möglich, die Matrizen in einfacher Genauigkeit im konstanten Speicher abzulegen. Diese Messung war zudem nur auf dem genutzten Laptopsystem möglich, da es mit der Treiberversion auf dem eigentlichen Rechnersystem in Zusammenhang mit dem konstanten Speicher Probleme gab. Auf dem Laptopsystem ergab sich kein signifikanter Unterschied zwischen der Nutzung des konstanten Speichers und der Nutzung des globalen Speichers mit dem zusätzlichen Schlüsselwort `const` (37,64 *s* zu 37,67 *s*). Es wird vermutet, dass dies auf die verbesserten Cachemöglichkeiten der aktuellen Grafikkarten zurückzuführen ist.

Auf Grund der beschränkten Größe des konstanten Speichers und dem kaum feststellbaren Geschwindigkeitsvorteil wurde diese Variante jedoch lediglich für den einmaligen Laufzeitvergleich verwendet. In allen anderen Modifikationen wurden die Transportmatrizen im globalen Speicher gehalten (versehen mit dem Schlüsselwort `const`).

Quelltext A.4: PATRIC: Transportschritt auf der GPU, Nutzung des konstanten Speichers.

```
1
2     // Anlegen des konstanten Speichers, hier fuer eine Transportmatrix
3     __constant__  float d_TC[36]; // transport matrix on the device,
4                                   // constant memory
5
6     // Kopieren einer Transportmatrix in den konstanten Speicher
7     cudasafe(cudaMemcpyToSymbol(d_TC, h_matrix,
8                                 36 * sizeof(float)),
9       "Could not copy the transport matrix to the device");
10
11    // Beispiel: Ansprechen im Rahmen der Berechnung
12    value += d_TC[matrixIndex++] * old_x;
```

A.3.3 PATRIC: Teilchendaten auf der GPU halten

Um die Teilchendaten während mehrerer Berechnungen auf der GPU zu halten, wurden diese im globalen Speicher abgelegt. Als Datenstruktur wurden Einzelarrays pro Koordinate verwendet (sog. structure of arrays), um einen möglichst performanten Zugriff der Threads auf die Daten zu erzielen.

In dem Array mit Namen `valid` wird gespeichert, ob ein Teilchen noch gültig, d. h. während der Simulation noch nicht verloren gegangen ist. Beim Kopieren der Daten zurück auf den Host wird diese Information ausgewertet, um dort nur noch die gültigen Teilchen zu erzeugen.

Quelltext A.5: PATRIC: Teilchendaten auf der GPU, Darstellung der genutzten Datenstruktur.

```
1
2 // collection of particles, structure-of-arrays
3 struct ParticleSoA {
4     int maxNumberOfParticles;
5
6     // arrays for particle coordinates
7     double* x;
8     double* xs;
9     double* y;
10    double* ys;
11    double* z;
12    double* dp;
13
14    // array for indicating, if a particle is still valid
15    bool* valid;
16 };
```

A.3.4 PATRIC: Nutzung von Streams

Um die laufzeitintensive Datenausgabe nach Möglichkeit mit der nächsten Berechnung zu überlappen, wurden in dieser Variante unterschiedliche Abarbeitungsströme, sog. Streams, genutzt. Auf der genutzten Grafikkarte kann dabei ein Kernelaufruf mit jeweils einer Kopieroperation vom bzw. zum Host überlappt werden. Die Programmvariante, bei der die Teilchen auf der GPU gehalten werden, wurde um die Definition der Streams erweitert. Beim Kopieren der Daten zu Ausgabezwecken zurück auf den Host wird ein anderer Stream genutzt, als für den nächsten Kernelaufruf zur Berechnung des nächsten Transportschritts. Für den Teilchentransport wurden die Arrays zur Darstellung der Teilchen dupliziert und nun mit Zeigern auf die Teilchenarrays gearbeitet. Nachdem eine Berechnung abgeschlossen ist, erfolgt eine Synchronisation der beiden Streams, bevor die Zeiger auf die Teilchen für den nächsten Kopier- bzw. Berechnungsschritt ausgetauscht werden.

Quelltext A.6: Nutzung von Streams: Quelltextauszüge zu Definition und Verwendung

```
1
2  // initialize streams
3  cudaStream_t streamCalc, streamPrint;
4  cudaStreamCreate(&streamCalc);
5  cudaStreamCreate(&streamPrint);
6
7  // reserve pinned memory on the host
8  cudasafe(cudaMallocHost((void**) &h_particles.x, size *sizeof(double)),
9      "Could not allocate particle coordinate array for x on the host");
10
11 // copy particles back up from the device to the host
12 cudasafe(cudaMemcpyAsync(h_particles.x, d_particles_source->x, size *
       sizeof(double), cudaMemcpyDeviceToHost, streamPrint),
13     "Could not copy x-Array to the host");
14
15 // pointer switching before each call to the transport kernel
16 tempPointer = particles_source;
17 particles_source = particles_target;
18 particles_target = tempPointer;
19
20 // kernel call including the stream
21 transport_kernel<<<dimGrid, dimBlock, 0, streamCalc>>>(d_T,
       elementLength, particlesSource, particlesTarget, accInfos,
       synParticle);
```

A.3.5 PATRIC: Strahlgrößen auf der GPU berechnen, Version mit CUDA

Als Beispiel für die Berechnung von Strahlgrößen auf der GPU dient die Berechnung der Emittanz, vgl. Formel 2.1.5. Da zunächst die Mittelwerte über die Koordinaten x und x' berechnet werden müssen, bevor die eigentliche Emittanz berechnet wird, muss zwischen diesen beiden Schritten ein globaler Synchronisationsschritt eingefügt werden. Wie in Kap. 2.3.4.2 beschrieben, setzt man eine solche globale Barriere mit einem erneuten Kernelaufruf um. Aus diesem Grund besteht die Emittanzberechnung aus zwei Kerneln, s. u. Im ersten Schritt werden die Mittelwerte über die Koordinaten x und x' berechnet, die im zweiten Schritt genutzt werden. Bei beiden Schritten handelt es sich um eine Reduktion mit O(log n) Laufzeit pro Block der zu summierenden Daten.

Quelltext A.7: Berechnung der Emittanz auf der GPU, Version mit CUDA.

```
1  __global__ void rms_emittance_kernel_part1(double* x, double* xs, bool*
       valid, int maxNumberOfParticles, double* result ){
2    __shared__ double tempData1[numberOfThreadsPerBlock];
3    __shared__ double tempData2[numberOfThreadsPerBlock];
4    __shared__ double numberOfParticles;
5    int threadId = threadIdx.x;
6    int globalId = blockIdx.x * blockDim.x + threadIdx.x;
7
8    // initialize data
9    if (threadId == 0)
10     numberOfParticles = 0;
11   __syncthreads();
12
13   // load data from global memory
14   if (globalId < maxNumberOfParticles && valid[globalId] == true) {
15     tempData1[threadId] = x[globalId];
16     tempData2[threadId] = xs[globalId];
17     atomicAdd(&numberOfParticles, 1);
18   } else {
19     tempData1[threadId] = 0;
20     tempData2[threadId] = 0;
21   }
22   __syncthreads();
23
24   // do the reduction (with sequential addressing)
25   for (int i = blockDim.x/2; i > 0; i = i/2) {
26     if (threadId < i) {
27       tempData1[threadId] += tempData1[threadId + i];
28       tempData2[threadId] += tempData2[threadId + i];
29     }
30     __syncthreads();
31   }
32
33   // thread 0 writes back the result to global memory
34   if (threadId == 0) {
35      atomicAdd(&result[0], tempData1[threadId]);
36      atomicAdd(&result[1], tempData2[threadId]);
37      atomicAdd(&result[2], numberOfParticles);
38   }
39 }
```

```
40 __global__ void rms_emittance_kernel_part2(double* x, double* xs, bool*
     valid, int maxNumberOfParticles, double* result ){
41   __shared__ double tempData1[numberOfThreadsPerBlock];
42   __shared__ double tempData2[numberOfThreadsPerBlock];
43   __shared__ double tempData3[numberOfThreadsPerBlock];
44   __shared__ double xMean;
45   __shared__ double xsMean;
46   __shared__ int n;
47   double temp_x, temp_xs;
48
49   // load data from global memory
50   int threadId = threadIdx.x;
51   int globalId = blockIdx.x * blockDim.x + threadIdx.x;
52   if (threadId == 0) {
53     double nAsDouble = result[2];
54     n = abs(nAsDouble);
55     xMean = result[0] / nAsDouble;
56     xsMean = result[1] / nAsDouble;
57   }
58   __syncthreads();
59
60   if (globalId < maxNumberOfParticles && valid[globalId] == true) {
61     temp_x = x[globalId];
62     temp_xs = xs[globalId];
63     tempData1[threadId] = pow(temp_x - xMean, 2);
64     tempData2[threadId] = pow(temp_xs -xsMean, 2);
65     tempData3[threadId] = (temp_x - xMean) * (temp_xs - xsMean);
66   } else {
67     tempData1[threadId] = 0;
68     tempData2[threadId] = 0;
69     tempData3[threadId] = 0;
70   }
71   __syncthreads();
72
73   // do the reduction (with sequential addressing)
74   for (int i = blockDim.x/2; i > 0; i = i/2) {
75     if (threadId < i) {
76       tempData1[threadId] += tempData1[threadId + i];
77       tempData2[threadId] += tempData2[threadId + i];
78       tempData3[threadId] += tempData3[threadId + i];
79     }
80     __syncthreads();
81   }
82
83   // thread 0 writes back the result to global memory
84   if (threadId == 0) {
85      atomicAdd(&result[5], tempData1[threadId]);
86      atomicAdd(&result[6], tempData2[threadId]);
87      atomicAdd(&result[7], tempData3[threadId]);
88   }
89   if (threadId == 0 && blockIdx.x == 0){
90     result[3] = xMean;
91     result[4] = xsMean;
92   }
93 }
```

A.3.6 PATRIC: Strahlgrößen auf der GPU berechnen, Version mit Thrust

Als Beispiel zum Einsatz von Thrust in den Routinen zur Strahlgrößenermittlung kam eine Variante der Emittanzermittlung zum Einsatz,die bei der Abteilung Strahlphysik entstanden war und dem unten gezeigten Programm als Vorlage diente. Der Quelltext wurde gegenüber der Originalvariante leicht abgewandelt, um eine Anpassung an die genutzten Datenstrukturen zu erzielen. Diese Thrust-Variante der Emittanzermittlung wurde bei den Laufzeitvergleichen mit der in Anhang A.3.5 vorgestellten eigenentwickelten Emittanzermittlung mit CUDA gegenübergestellt.

Quelltext A.8: Berechnung der Emittanz auf der GPU, Version mit Thrust.

```
1 extern "C" float rms_emittance_gpu_thrust_internal(ParticleSoA particles,
      double *result) {
2
3    int Npic = particles.maxNumberOfParticles;
4    float tem1, tem2, tem3, tem4, tem5;
5
6    cudaEvent_t start, stop;
7      cudaEventCreateWithFlags(&start, cudaEventBlockingSync);
8      cudaEventCreateWithFlags(&stop, cudaEventBlockingSync);
9      cudaEventRecord(start, 0);
10
11   thrust::device_ptr<double> dev_ptr_x(particles.x);
12   thrust::device_ptr<double> dev_ptr_xs(particles.xs);
13   thrust::device_vector<double> temp(Npic);
14
15   tem4 = thrust::reduce(dev_ptr_x, dev_ptr_x + Npic)/Npic;
16   tem5 = thrust::reduce(dev_ptr_xs, dev_ptr_xs + Npic)/Npic;
17
18   tem1 = thrust::transform_reduce(dev_ptr_x, dev_ptr_x + Npic,
         emittance_helper(tem4), 0.0, thrust::plus<double>());
19   tem2 = thrust::transform_reduce(dev_ptr_xs, dev_ptr_xs + Npic,
         emittance_helper(tem5), 0.0, thrust::plus<double>());
20
21   thrust::transform(dev_ptr_x, dev_ptr_x + Npic, dev_ptr_xs,temp.begin()
         , emittance_helper2(tem4,tem5));
22   tem3 = reduce(temp.begin(), temp.end());
23
24     cudasafe(cudaDeviceSynchronize(), "Error while calculating the
          Emittance using Thrust");
25     cudaEventRecord(stop, 0);
26     cudaEventSynchronize(stop);
27     float recordedTime = 0;
28     cudaEventElapsedTime(&recordedTime, start, stop);
29
30   *result = sqrt(tem1 * tem2 / pow((float) Npic, 2) - pow(tem3 / (float)
       Npic, 2));
31   return recordedTime;
32
33 }
```

A.3.7 LOBO: Teilchen und Gitter auf der GPU, Interpolation mit atomaren Operationen

Im Rahmen der Parallelisierung des Programms LOBO wurden die Schritte der Simulation zur Interpolation der Teilchen auf die Gitter und umgekehrt, sowie die Fortbewegung der Teilchen auf die GPU portiert. Dazu wurde zunächst die bestehende Aufteilung in Unterschritte beibehalten, um eine einfache Wartbarkeit zu erreichen. Die folgenden Programmausschnitte zeigen beispielhaft die Kernel für das Bewegen der Teilchen (`kernel_push_z` und `kernel_push_dp`) sowie den Kernel zur Bestimmung der Stromdichte (`kernel_line_current_density`) auf dem Gitter. Bei der Interpolation der Teilchen auf das Gitter wird dabei mit atomaren Operationen gearbeitet.

Quelltext A.9: LOBO: Kernel zur Bewegung der Teilchen.

```
1  __global__ void kernel_push_z(StructBeam beam, double begin, double end,
       double dt, SynParticle sp) {
2
3    int threadNumber = blockIdx.x * blockDim.x + threadIdx.x;
4    if (threadNumber < beam.numberOfParticles) {
5      beam.particles_z[threadNumber] -= sp.eta0 * beam.particles_dp[
           threadNumber] * clight * sp.beta0 * dt;
6      if (beam.particles_z[threadNumber] < begin)
7        beam.particles_z[threadNumber] += end - begin;
8      else if (beam.particles_z[threadNumber] > end)
9        beam.particles_z[threadNumber] -= end - begin;
10   }
11 }
12
13 __global__ void kernel_push_dp(StructBeam beam, StructGrid1D efield,
       double dt, SynParticle sp) {
14
15   int threadNumber = blockIdx.x * blockDim.x + threadIdx.x;
16   if (threadNumber < beam.numberOfParticles) {
17     double charge = qe * sp.Z;
18     double mass = mp * sp.A;
19     double temp = charge / (sp.beta0 * clight * sp.gamma0 * mass) * dt;
20
21     double z0 = beam.particles_z[threadNumber];
22       int j1, j2;
23     double f1, f2;
24     double dist0 = z0 - efield.begin;
25     j1 = (int) floor(dist0 / efield.dz - 0.5);
26     j2 = j1 + 1;
27     f1 = ((j2 + 0.5) * efield.dz - dist0) / efield.dz;
28     f2 = (dist0 - (j1 + 0.5) * efield.dz) / efield.dz;
29
30     double pic_efield = efield.gridValues[j1] * f1 + efield.gridValues[j2
           ] * f2;
31
32     beam.particles_dp[threadNumber] += temp * pic_efield;
33   }
34 }
```

Quelltext A.10: LOBO: Interpolation, Variante mit atomaren Operationen.

```
__global__ void kernel_line_current_density( StructGrid1D ldy, StructBeam
    beam, SynParticle sp ) {

  int threadNumber = blockIdx.x * blockDim.x + threadIdx.x;

  // pic to field
  double pic_quantity;
  if (threadNumber < beam.numberOfParticles) {
    pic_quantity = beam.charge * sp.beta0 * clight / ldy.dz;

    int j1, j2;
    double f1, f2;
    double dist0 = beam.particles_z[threadNumber] - ldy.begin;
    j1 = (int) floor(dist0 / ldy.dz - 0.5);
    j2 = j1 + 1;
    f1 = ((j2 + 0.5) * ldy.dz - dist0) / ldy.dz;
    f2 = (dist0 - (j1 + 0.5) * ldy.dz) / ldy.dz;

    // periodic boundary conditions
    j1 = periodic_boundary_condition(ldy.n, j1);
    j2 = periodic_boundary_condition(ldy.n, j2);

    // add values to the grid
    atomicAdd(&ldy.gridValues[j1], pic_quantity * f1);
    atomicAdd(&ldy.gridValues[j2], pic_quantity * f2);
  }
}
```

A.3.8 LOBO: Teilchen und Gitter auf der GPU, Interpolation mit vorheriger Sortierung der Teilchen

In einer zweiten Variante wurden die Teilchen nach jedem Durchlaufen des PIC-Simulationszyklus neu nach der Gitterzelle sortiert, in der sie sich dann aktuell befinden. Diese Sortierung ist im Folgenden dargestellt. Bei der Interpolation kann nun mit einem Thread pro Gitterzelle gearbeitet werden. Der einzelne Thread muss dann alle Teilchen bearbeiten, die dieser Gitterzelle zugeordnet sind. Diese Interpolation ist ebenfalls dargestellt.

Quelltext A.11: LOBO: Variante mit Sortierung, Sortiermechanismus.

```
1 extern "C" float sort_particles( StructBeam beam, StructBucketBounds
      bucketBounds ) {
2
3   cudaEvent_t start, stop;
4   cudaEventCreateWithFlags(&start, cudaEventBlockingSync);
5   cudaEventCreateWithFlags(&stop, cudaEventBlockingSync);
6   cudaEventRecord(start, 0);
7
8   // sort particles
9   thrust::device_ptr<int> dev_ptr_particles_bucket(beam.particles_bucket)
        ;
10  thrust::device_ptr<double> dev_ptr_particles_z(beam.particles_z);
11  thrust::device_ptr<double> dev_ptr_particles_dp(beam.particles_dp);
12
13  thrust::device_ptr<int> dev_ptr_bucket_sequence_numbers(bucketBounds.
        sequence_numbers);
14  thrust::device_ptr<int> dev_ptr_bucket_lower_bounds(bucketBounds.
        bucket_lower_bounds);
15  thrust::device_ptr<int> dev_ptr_bucket_amount(bucketBounds.
        bucket_amount);
16
17  thrust::sort_by_key(  dev_ptr_particles_bucket,
        dev_ptr_particles_bucket + beam.numberOfParticles, thrust::
        make_zip_iterator(thrust::make_tuple(  dev_ptr_particles_z,
        dev_ptr_particles_dp)));
18
19  thrust::lower_bound(dev_ptr_particles_bucket, dev_ptr_particles_bucket
        + beam.numberOfParticles, dev_ptr_bucket_sequence_numbers,
        dev_ptr_bucket_sequence_numbers + bucketBounds.n,
        dev_ptr_bucket_lower_bounds);
20
21  thrust::transform(dev_ptr_bucket_lower_bounds +1,
        dev_ptr_bucket_lower_bounds + (bucketBounds.n + 1) ,
        dev_ptr_bucket_lower_bounds , dev_ptr_bucket_amount, thrust::minus<
        int>());
22
23  cudaEventRecord(stop, 0);
24  cudaEventSynchronize(stop);
25  float recordedTime = 0;
26  cudaEventElapsedTime(&recordedTime, start, stop);
27  return recordedTime;
28 }
```

Quelltext A.12: LOBO: Variante mit Sortierung, Interpolation.

```
1
2  __global__ void kernel_line_current_density_block(StructGrid1D ldy,
       StructBeam beam, StructBucketBounds bucketBounds, double pic_quantity
       , int maxParticleNumber ) {
3
4      // there are as many threads as buckets
5    int threadNumber = blockIdx.x * blockDim.x + threadIdx.x;
6
7    // pic to field
8    if (threadNumber < ldy.n) {
9
10     // get particle start index to work on
11     int particleStartIndex = bucketBounds.bucket_lower_bounds[
           threadNumber];
12     // get number of particles to work on
13     int amount = bucketBounds.bucket_amount[threadNumber];
14
15     int particleIndex;
16     int j1, j2;
17     double dist0;
18
19     if (amount > 0 ) {
20       double leftValue = 0;
21       double rightValue = 0;
22
23       for (int i = 0; i < maxParticleNumber; i++) {
24         particleIndex = particleStartIndex + i;
25         if (i < amount && particleIndex < beam.numberOfParticles) {
26           dist0 = beam.particles_z[threadNumber] - ldy.begin;
27           j1 = (int) floor(dist0 / ldy.dz - 0.5);
28           j2 = j1 + 1;
29
30           leftValue += pic_quantity * (((j2 + 0.5) * ldy.dz - dist0) /
                 ldy.dz);
31           rightValue += pic_quantity * ((dist0 - (j1 + 0.5) * ldy.dz) /
                 ldy.dz);
32         }
33       }
34
35       // add values to the grid
36       atomicAdd(&ldy.gridValues[threadNumber], leftValue);
37       atomicAdd(&ldy.gridValues[(threadNumber + 1) % ldy.n], rightValue);
38     }
39   }
40 }
```

B Formeln

B.1 Transportmatrizen

Die im Rahmen der linearen Strahloptik verwendeten Transportmatrizen sind im Folgenden als Referenz aufgelistet. Die Matrizen mit samt ihrer Herleitung finden sich bei [Hin08, S. 135ff.].

Driftstrecke der Länge L:

$$M_{\text{Drift}} = \begin{pmatrix} 1 & L & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & L & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\gamma^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.1}$$

Dipol:

$$M_{\text{Dipol}} = \begin{pmatrix} \cos\alpha & \rho_0 \sin\alpha & 0 & 0 & 0 & \rho_0(1-\cos\alpha) \\ -\frac{\sin\alpha}{\rho_0} & \cos\alpha & 0 & 0 & 0 & \sin\alpha \\ 0 & 0 & 1 & \rho_0\alpha & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -\sin\alpha & -\rho_0(1-\cos\alpha) & 0 & 0 & 1 & \rho_0\frac{\alpha}{\gamma^2} - \rho_0(\alpha - \sin\alpha) \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.2}$$

Quadrupol (horizontal fokussierend, vertikal defokussierend),
k als normalisierte Quadrupolstärke:

$$M_{\text{QF}} = \begin{pmatrix} \cos\sqrt{k}L & \frac{\sin\sqrt{k}L}{\sqrt{k}} & 0 & 0 & 0 & 0 \\ -\sqrt{k}\sin\sqrt{k}L & \cos\sqrt{k}L & 0 & 0 & 0 & 0 \\ 0 & 0 & \cosh\sqrt{k}L & \frac{\sinh\sqrt{k}L}{\sqrt{k}} & 0 & 0 \\ 0 & 0 & \sqrt{k}\sinh\sqrt{k}L & \cosh\sqrt{k}L & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\gamma^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.3}$$

Quadrupol (horizontal defokussierend, vertikal fokussierend):

$$M_{\text{QD}} = \begin{pmatrix} \cos\sqrt{k}L & \frac{\sinh\sqrt{k}L}{\sqrt{k}} & 0 & 0 & 0 & 0 \\ -\sqrt{k}\sinh\sqrt{k}L & \cosh\sqrt{k}L & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos\sqrt{k}L & \frac{\sin\sqrt{k}L}{\sqrt{k}} & 0 & 0 \\ 0 & 0 & -\sqrt{k}\sin\sqrt{k}L & \cos\sqrt{k}L & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{L}{\gamma^2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.4}$$

B.2 Fehlerformeln

Im Folgenden sind die Formeln zur Fehlerabschätzung bei der Summierung von n Zahlen dargestellt. Die Herleitungen finden sich bei [Übe95].

Serielle Summierung:

$$|\tilde{s} - s| \leq \frac{k \cdot \epsilon}{1 - k \cdot \epsilon} \sum_{i=1}^{n} |x_i| \tag{B.5}$$

Paarweise (binäre) Summierung, wobei $n = 2^k$:

$$|\tilde{s} - s| \leq |x_1| R_{n-1} + \sum_{i=2}^{n} |x_i| \cdot R_{n-i+1} \quad \text{mit } R_m := \frac{m \cdot \epsilon}{1 - m \cdot \epsilon} \tag{B.6}$$

C Simulationsprogramme

C.1 Die Konfigurationsdatei von PATRIC

Die folgende Konfigurationsdatei zeigt die verwendeten Einstellungen für das Simulationsprogramm PATRIC. Im Rahmen der Arbeit wurde vor allem die Einstellung für die Anzahl der Teilchen variiert. Die Häufigkeit, mit der Zwischenergebnisse während des Simulationslaufs ausgegeben werden, wurde darüber hinaus direkt im Quelltext des Programms verändert.

```
patric_dict=OrderedDict()
patric_dict['NPIC']=100000
patric_dict['NX']=128
patric_dict['NY']=128
patric_dict['cells']=128
patric_dict['e_kin']=3000.0
patric_dict['Z']=18.0
patric_dict['A']=40.0
patric_dict['Np']=2.0e11
patric_dict['piperadius']=0.075
patric_dict['circum']=54.0
patric_dict['gamma_t']=5.4
patric_dict['CF_advance_h']=129.3*pi/180.0
patric_dict['CF_advance_v']=129.3*pi/180.0
patric_dict['CF_R']=0.0
patric_dict['CF_length']=18.0
patric_dict['NCF']=16
patric_dict['koct']=8.0
patric_dict['pic_subset']=10000
patric_dict['init_pic_xy']=0  # 0 (WB), 1 (KV), 2 (SG), 3 (GS)
patric_dict['momentum_spread']=1.0e-2
patric_dict['rms_emittance_x0']=12.5
patric_dict['rms_emittance_y0']=12.5
patric_dict['mismatch_x']=1.0
patric_dict['mismatch_y']=1.0
patric_dict['offcenter']=0.0
patric_dict['madx_input_file']=0
patric_dict['octupole_kick']=0
patric_dict['chroma']=1
```

C.2 Die Konfigurationsdatei von LOBO

Die im Rahmen der Arbeit verwendete Konfiguration des Simulationsprogramms LOBO ist im Folgenden dargestellt. Es handelt sich dabei um eine Konfigurationsdatei, welche zu Anfang einmal eingelesen wird. Sie enthält die für die Simulation verwendeten Einstellungen. Im Rahmen der Arbeit wurden die Teilchenzahl und die Gittergrößen variiert.

```
Points_in_z_direction(NZ):
512
Points_in_dp_direction(NV):
512
Particles(NI):
8.0e8
Macroparticles((int)NP):
100000
Charges(Z):
18
Mass(A):
40
cut-off(h_c):
10
broadband-impedanz(BRI=0,1):
0
shunt-impedanz(RS):
0
Often_print(print_step):
25
endTime(t_max):
1.e-5
dp/p_max(dp):
3.7e-3
n_mal_dp(vmax):
10
Scheme(elliptic=0;gauss=1;):
0
bunchlengh(zlm):
0.2
```

D Inhalt der beigefügten DVD

Auf der beigefügten DVD befindet sich die Diplomarbeit im pdf-Format, der Quelltext der verschiedenen Programmmodifikationen sowie Messergebnisse und Auswertescripte. Die folgende Übersicht zeigt den Inhalt der DVD:

- INFO.txt: Überblick über die Inhalte der DVD, außerdem Informationen zur Laufzeitumgebung (Betriebssystem, Treiberversion, etc.)
- Quelltext: Pro Quelltextmodifikation existiert ein Unterverzeichnis, geringfügige Modifikationen sind teilweise in einem Unterverzeichnis zusammengefasst
  - patric_ original_ meas:
    Originalversion von PATRIC mit Zeitmessung, die als Grundlage für die Einbeziehung der GPU diente
  - patric_ gpu_ v1_ transport:
    Transportschritt auf der GPU, enthält die Varianten, die ein Teilchen oder eine Teilchenkoordinate auf der GPU berechnen und den Vergleich konstanter vs. globaler Speicher
  - patric_ gpu_ v2_ particles:
    Hauptversion von PATRIC auf der GPU: hier werden die Teilchen so lange wie möglich auf der GPU gehalten. Enthält verschiedene Varianten, z. B. zur Berechnung von Strahlgrößen auf der CPU oder GPU und eine Variante mit 50% Teilchenverlust
  - patric_ gpu_ v3_ single_ double:
    Vergleich der Nutzung einfacher oder doppelter Genauigkeit bei der Berechnung
  - patric_ gpu_ v4_ streams:
    Überlappung des Datenkopierens und der Berechnungen mittels Streams
  - lobo_ original_ meas:
    Originalversion von LOBO mit Zeitmessung, die als Grundlage für die Einbeziehung der GPU diente
  - lobo_ gpu_ v1_ atomics:
    LOBO auf der GPU, Interpolation mit atomaren Operationen
  - lobo_ gpu_ v2_ sort:
    LOBO auf der GPU, Sortierung der Teilchen
- Messungen: Messergebnisse incl. Auswertescripte
- Diplomarbeit: Diplomarbeit im pdf-Format
- Literatur: Überblick über das Literaturverzeichnis

Literaturverzeichnis

[ABHS08] Appleby, R; Bailey, D; Higham, J ; Salt, M: *High performance stream computing for particle beam transport simulations*. In: *Journal of Physics: Conference Series* 119 (2008), Nr. 4, S. 042001. http://dx.doi.org/10.1088/1742-6596/119/4/042001

[ABP+12a] Amyx, K.; Balasalle, J.; Pogorelov, I.; Borland, M.; Soliday, R. ; Wang, Y.: *Accelerating Particle-Tracking Based Beam Dynamics Simulations with GPUs*. In: *Proc. of the GPU Technology Conference (GTC) 2012 - Poster*. San Jose, CA, USA : Tech-X Corporation, Argonne National Laboratory, 2012

[ABP+12b] Amyx, K.; Balasalle, J.; Pogorelov, I.; Borland, M.; Soliday, R. ; Wang, Y.: *CUDA Kernel Design for GPU-Based Beam Dymanics Simulations*. In: *Proc. of the International Particle Accelerator Conference (IPAC) 2012*. New Orleans, LO, USA, Juli 2012, S. 343–345

[ABP+13] Amyx, K.; Balasalle, J.; Pogorelov, I.; Borland, M.; Soliday, R. ; Wang, Y.: *GPU-Accelerated Beam Dynamics Simulations with ELEGANT*. In: *Proc. of the GPU Technology Conference (GTC) 2013 - Poster*, 2013

[Adv13] Advanced Micro Devices Inc.: *AMD Accelerated Processing Units*. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx#1, besucht am 04.09.2013

[AFPS11] Abreu, Paulo; Fonseca, Ricardo A.; Pereira, João M. ; Silva, Luís O.: *PIC Codes in New Processors: A Full Relativistic PIC Code in CUDA-Enabled Hardware With Direct Visualization*. In: *IEEE Transactions on Plasma Science* 39 (2011), Nr. 2, S. 675–685. http://dx.doi.org/10.1109/TPS.2010.2090905

[AHU74] Aho, Alfred V.; Hopcroft, John E. ; Ullman, Jeffrey D.: *The design and analysis of computer algorithms*. Reading, MA, USA : Addison-Wesley, 1974

[Akl89] Akl, Selim G.: *Design and analysis of parallel algorithms.* Englewood Cliffy, NJ, USA : Prentice Hall, 1989

[Amy] Amyx, Keegan M.: *Antwort von K.M. Amyx, Tech-X Corp., auf eine Anfrage vom 13.11.2012*

[App11] Appel, Sabrina: *Simulation und Messung longitudinaler Raumladungseffekte in intensiven Ionenstrahlen im SIS18 Synchrotron*, Technische Universität Darmstadt, Diss., Mai 2011. http://tuprints.ulb.tu-darmstadt.de/2594/1/TUDthesis_sappel.pdf

[BE95] Borland, M.; Emery, L.: *The self-describing data sets file protocol and Toolkit*. In: *Proc. of the International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS) 1995*. Chicago, IL, USA : Argonne National Laboratory, Dezember 1995

[BFH00] Boine-Frankenheim, O.; Hofmann, I.: *Vlasov simulation of the microwave instability in space charge dominated coasting ion beams*. In: *Physical Review Special Topics - Accelerators and Beams* 3 (2000), Nr. 10, S. 104202. http://dx.doi.org/10.1103/PhysRevSTAB.3.104202

[BFK06] Boine-Frankenheim, O.; Kornilov, V.: *Implementation and Validation of Space Charge and Impedance Kicks in the Code Patric for Studies of Transverse Coherent Instabilities in the Fair Rings*. In: *Proc. of the International Computational Accelerator Physics Conference (ICAP) 2006*. Chamonix, France, 2006, S. 267–270

[BL05] Birdsall, C.K.; Langdon, A.B.: *Plasma Physics via Computer Simulation*. New York, NY, USA : Taylor and Francis Group, 2005

[Bor10] Borland, M.: *Overview of elegant and SDDS*. http://www.aps.anl.gov/Accelerator_Systems_Division/Accelerator_Operations_Physics/elegant.html, besucht am 28.03.2013

[Bor12] Borland, Michael: *User's Manual for elegant, Program Version 25.1.0* / Argonne National Laboratory. Version: 2012. http://aps.anl.gov/Accelerator_Systems_Division/Accelerator_Operations_Physics/manuals/elegant_latest/elegant.pdf. Argonne, IL, USA, 2012

[BSS+09] Borland, M.; Sajaev, V.; Shang, H.; Soliday, R.; Wang, Y. ; Xiao, A.: *Recent Progress and Plans for the Code ELEGANT*. In: *Proc. of the Computational Accelerator Physics Conference (ICAP) 2009*. San Francisco, CA, USA, 2009, S. 111–116

[BWH+10] Burau, Heiko; Widera, Renée; Honig, Wolfgang; Juckeland, Guido; Debus, Alexander; Kluge, Thomas; Schramm, Ulrich; Cowan, Tomas E.; Sauerbrey, Roland ; Bussmann, Michael: *PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster*. In: *IEEE Transactions on Plasma Science* 38 (2010), Nr. 10, S. 2831–2839. http://dx.doi.org/10.1109/TPS.2010.2064310

[CC97] Carmona, Edward A.; Chandler, Leon J.: *On parallel PIC versatility and the structure of parallel PIC approaches*. In: *Concurrency: Practice and Experience* 9 (1997), Nr. 12, S. 1377–1405. http://dx.doi.org/10.1002/(SICI)1096-9128(199712)9:12<1377::AID-CPE284>3.0.CO;2-Q

[Dep11] Department of Energy: *Accelerating Large-Scale Beam Dynamics Simulations with GPUs*. http://www.sbir.gov/sbirsearch/detail/373586, besucht am 01.04.2013

[DS11] Decyk, Viktor K.; Singh, Tajendra V.: *Adaptable Particle-in-Cell algorithms for graphical processing units*. In: *Computer Physics Communications* 182 (2011), Nr. 3, S. 641–648. http://dx.doi.org/10.1016/j.cpc.2010.11.009

[FAI08] FAIR: *FAIR Technical Design Report SIS100* / Gesellschaft für Schwerionenforschung. Version: 2008. http://www-win.gsi.de/fair-eoi/PDF/TDR_PDF/TDR_SIS100_19-03-08-dk-1.pdf. Darmstadt, 2008

[Far11] Farber, Rob: *CUDA Application Design and Development*. Amsterdam : Morgan Kaufmann, 2011

[GLS99] Gropp, William; Lusk, Ewing ; Skjellum, Anthony: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. 2. Aufl. Cambridge, MA, USA : MIT Press, 1999

[Har07] Harris, Mark: *Optimizing Parallel Reduction in CUDA* / NVIDIA Corporation. Version: 2007. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. 2007

[HB12] Hoberock, Jared; Bell, Nathan: *Thrust, open source parallel algorithms library*. http://thrust.github.io/, besucht am 16.06.2013

[Hel13] Helmholtz-Zentrum Dresden-Rossendorf: *PIConGPU Projekt Webseite*. http://www.hzdr.de/db/Cms?pOid=31887&pNid=0, besucht am 10.07.2013

[Hig02] Higham, Nicholas J.: *Accuracy and Stability of Numerical Algorithms*. 2. Aufl. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2002

[Hin08] Hinterberger, Frank: *Physik der Teilchenbeschleuniger und Ionenoptik*. 2. Aufl. Berlin : Springer-Verlag, 2008

[HSW+10] Hönig, W.; Schmitt, F.; Widera, R.; Burau, H.; Juckeland, G.; Müller, M. S. ; Bussmann, M.: *A Generic Approach for Developing Highly Scalable Particle-Mesh Codes for GPUs*. In: *Proc. of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC) 2010*. Knoxville, TN, USA, 2010

[IEE08] IEEE: *754-2008 - IEEE Standard for Floating-Point Arithmetic*. (2008). http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[JáJ92] JáJá, Joseph: *An Introduction to Parallel Algorithms*. Reading, MA, USA : Addison-Wesley, 1992

[JB10] Juckeland, Guido; Bussmann, Michael: *Presentation on GTC2010: Developing Highly Scalable Particle-Mesh Codes for GPUs: A Generic Approach*. http://nvidia.fullviewmedia.com/gtc2010/0921-n-2090.html

[Kal94] Kalisch, G.: *Erzeugung und Untersuchung gepulster Schwerionen-Strahlen höchster Phasenraumdichte im Experimentier-Speicherring der GSI*, Technische Hochschule Darmstadt, Diss., 1994

[KH10] Kirk, David B.; Hwu, Wen-mei W.: *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010

[KHRD11] Kong, Xianglong; Huang, Michael C.; Ren, Chuang ; Decyk, Viktor K.: *Particle-in-cell simulations with charge-conserving current deposition on graphic processing units*. In: *Journal of Computational Physics* 230 (2011), Nr. 4, S. 1676–1685. http://dx.doi.org/10.1016/j.jcp.2010.11.032

[Koe13] Koehler, Axel: *NVIDIA Roadmap of future GPU computing - Presentation*. In: *Proceedings of the Graphics Processing Units (GPUs) in High Energy Physics Workshop*, 2013

[Mes12] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 3.0* / University of Tennessee. Version: 2012. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Knoxville, TN, USA, 2012

[MMG08] Messmer, Peter; Mullowney, Paul J. ; Granger, Brian E.: *GPULib: GPU Computing in High-Level Languages*. In: *Computing in Science & Engineering* 10 (2008), Nr. 5, S. 70–73. http://dx.doi.org/10.1109/MCSE.2008.135

[MPI12] MPICH: *MPICH 3.0: An Implementation of the MPI Standard*. http://www.mpich.org, besucht am 04.03.2013

[MSM05] Mattson, Timothy G.; Sanders, Beverly A. ; Massingill, Berna L.: *Patterns for Parallel Programming*. Boston, MA, USA : Addison-Wesley, 2005

[NVI07] NVIDIA Corporation: *CUDA C Programming Guide, Version 1.1*. 2007

[NVI09a] NVIDIA Corporation: *Fermi Compute Architecture Whitepaper*. Version: 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 2009

[NVI09b] NVIDIA Corporation: *NVIDIA Tesla C2075 Companion Processor*. Version: 2009. http://www.nvidia.com/content/PDF/data-sheet/NV_DS_Tesla_C2075_Sept11_US_HR.pdf. 2009

[NVI12] NVIDIA Corporation: *CUDA Technology*. http://www.nvidia.com/CUDA, besucht am 31.08.2012

[NVI13a] NVIDIA Corporation: *CUDA C Best Practices Guide*. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, besucht am 16.06.2013

[NVI13b] NVIDIA Corporation: *CUDA C Programming Guide, Version 5.0*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[NVI13c] NVIDIA Corporation: *CUDA Occupancy Calculator*. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls, besucht am 01.03.2013

[NVI13d] NVIDIA Corporation: *CUDA Toolkit Documentation: Thrust*. http://docs.nvidia.com/cuda/thrust/index.html, besucht am 16.06.2013

[Rei08] Reiser, Martin: *Theory and design of charged particle beams*. 2. Aufl. Weinheim : Wiley-VCH, 2008

[RPMA10] Ranjbar, V.; Pogorelov, I.; Messmer, P. ; Amyx, K.: *Accelerated Particle Tracking using GPULib*. In: *Computational Challenges in High-Intensity Linacs, Rings incl. FFAGs, Cyclotrons*. Boulder, CO, USA : Tech-X Corporation, 2010, S. 286–289

[Sch00] Schüle, Josef: *Parallel Computing with Emphasis on Distributed Systems*. Aachen : Shaker Verlag, 2000

[SDG08] Stantchev, George; Dorland, William ; Gumerov, Nail: *Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU*. In: *Journal of Parallel and Distributed Computing* 68 (2008), Nr. 10, S. 1339–1349. http://dx.doi.org/10.1016/j.jpdc.2008.05.009

[SHFP+08] Saa-Hernandez, A.; Franchetti, G.; Pyka, N.; Ratzinger, U. ; Spiller, P.: *Slow Extraction Simulations for SIS300* / Gesellschaft für Schwerionenforschung. Version: 2008. http://web-docs.gsi.de/$\sim$giuliano/publications/reports_not_on_web/gsirep08-saa.pdf. Darmstadt, 2008

[SK10] Sanders, Jason; Kandrot, Edward: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ, USA : Addison Wesley, 2010

[Tan05] Tanenbaum, Andrew S.: *Computerarchitektur*. 5. Aufl. München : Pearson Studium, 2005

[Tan09] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. 3. akt. A. München : Pearson Studium, 2009

[TOP12] TOP500: *Top500 Supercomputer Rangliste*. http://s.top500.org/static/lists/2012/11/PressRelease201211.pdf, besucht am 08.02.2013

[Übe95] Überhuber, Christoph: *Computer-Numerik 1*. Berlin : Springer-Verlag, 1995

[Val90] Valiant, Leslie G.: *A bridging model for parallel computation*. In: *Communications of the ACM* 33 (1990), Nr. 8, S. 103–111. http://dx.doi.org/10.1145/79173.79181

[WA04] Wilkinson, Barry; Allen, Michael: *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 2. Aufl. Upper Saddle River, NJ, USA : Prentice Hall, 2004

[WB06] Wang, Y.; Borland, M.: *Pelegant: A Parallel Accelerator Simulation Code for Electron Generation and Tracking*. In: *Proc. of the 12th Advanced Accelerator Concepts Workshop*. Argonne, IL, USA : Argonne National Laboratory, 2006

[WB07] Wang, Y; Borland, M.: *Implementation and performance of parallelized elegant*. In: *Proc. of the IEEE Particle Accelerator Conference (PAC) 2007*. Albuquerque, NM, USA : IEEE, 2007, S. 3444–3446

[WFF11] Whitehead, Nathan; Fit-Florea, Alex: *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs* / NVIDIA Corporation. Version: 2011. `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`. 2011

[WGW06] Wolfheimer, F.; Gjonaj, E. ; Weiland, T.: *Parallel Particle-In-Cell (OIC) Codes*. In: *Proc. of the Computational Accelerator Physics Conference (ICAP) 2006*. Chamonix, France, 2006, S. 290–295

[Wil96] Wille, Klaus: *Physik der Teilchenbeschleuniger und Synchrotronstrahlungsquellen*. 2. Aufl. Stuttgart : Teubner Verlag, 1996

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

_____

Groß-Umstadt, im September 2013