Optimierung paralleler Particle-In-Cell-Verfahren für Grafikprozessoren

Philipp Raithel

Bayreuth Reports on Parallel and Distributed Systems

No. 11, January 2019

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

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

UNIVERSITÄT BAYREUTH

BACHELOR ARBEIT

Optimierung paralleler Particle-In-Cell-Verfahren für Grafikprozessoren

---

Author:
Philipp Raithel

1. Gutachter
PD Dr. Matthias Korch
2. Gutachter
Prof. Dr. Gerhard Rein

Bachelorarbeit zur Erlangung des akademischen Grades Bachelor of Science

Angewandte Informatik 2
Fakultät für Mathematik, Physik und Informatik der Universität Bayreuth

14. Januar 2019

UNIVERSITÄT BAYREUTH

Zusammenfassung

Bachelor of Science

Optimierung paralleler Particle-In-Cell-Verfahren für Grafikprozessoren

von Philipp Raithel

Diese Ausarbeitung behandelt die Implementierung von Particle-In-Cell-Verfahren für GPUs ausgehend von bereits vorhandenen parallelen Implementierungen für CPUs. Diese werden mit dem CUDA-Framework für Nvidia-GPUs entwickelt und im Verlauf der Arbeit weiter analysiert und optimiert.

Zu Beginn wird auf die Unterschiede zwischen CPU- und GPU-Implementierung eingegangen und die nötigen Schritte erläutert, wie GPU-Implementierungen ausgehend von CPU-Codes entwickelt werden können. Anschließend wird eine erste konkrete Version der Particle-In-Cell-Verfahren implementiert, welche während der fortlaufenden Arbeit als Referenz für weitere Optimierungen dient. Mit Profiling durch den Visual-Profiler von Nvidia werden danach Bottlenecks identifiziert und mit unterschiedlichen Ansätzen behandelt. Um die Effizienz der einzelnen Ansätze vergleichen zu können, wird jede optimierte Variante in Bezug auf die jeweils anderen Varianten mithilfe von Laufzeitmessungen und Profiling-Ergebnissen bewertet. Zum Schluss wird der Einfluss der Genauigkeit von Fließkommazahlen auf die Simulationsergebnisse und die Laufzeit der Simulationen untersucht.

UNIVERSITY OF BAYREUTH

Abstract

Bachelor of Science

Optimization of parallel particle-in-cell methods for graphics processors

from Philipp Raithel

This thesis covers the implementation of Particle-In-Cell methods for GPUs, which are based on existing parallel implementations for CPUs. These are developed with the CUDA framework for Nvidia-GPUs and are further analyzed and optimized in the course of the thesis.

At the beginning, the differences between CPU and GPU implementations are discussed and the necessary steps to develop GPU implementations based on CPU codes are explained. Subsequently, a first concrete version of the Particle-In-Cell methods is implemented, which serves as a reference for further optimizations during the ongoing work. Bottlenecks are identified with profiling by the Visual-Profiler of Nvidia and treated with different approaches. In order to be able to compare the efficiency of the individual approaches, each optimized variant is evaluated in relation to the other variants using runtime measurements and profiling results. Finally, the influence of the accuracy of floating point numbers on the simulation results and the runtime of the simulations is investigated.

Inhaltsverzeichnis

1. Motivation

Particle-In-Cell-Verfahren (PIC-Verfahren) dienen der Simulation von Partikeln unter physikalischen Einflüssen, z. B. der Gravitation, welche auch in dieser Arbeit betrachtet wird. Der große Rechenaufwand für diese Simulationen fordert eine gute parallele Implementierung, um eine möglichst große Zahl an Partikeln und Zeitschritten in gegebener Zeit zu simulieren. Bisher wurden sequenzielle und parallele CPU-Implementierungen für die Vlasov-Poisson- und die Vlasov-Einstein-Simulation mithilfe von MPI und Pthreads entwickelt, welche als Ausgangspunkt dieser Arbeit anzusehen sind [10]. Das Ziel dieser Arbeit ist die Implementierung und Optimierung ebendieser Simulationen mit dem CUDA-Framework für Nvidia-GPUs. Um eine möglichst effiziente GPU-Implementierung zu erreichen, ist die Ausnutzung des massiv-parallelen Programmiermodells von GPUs vonnöten.

2. Stand der Forschung

Für parallele Implementierungen von Particle-In-Cell-Verfahren existieren viele wissenschaftliche Arbeiten, sowohl für CPUs als auch für GPUs. Interessant für diese Arbeit sind besonders diejenigen, die sich mit Implementierungen für GPUs beschäftigen. Im Folgenden werden zwei bekannte Artikel betrachtet und mit den Schlüsselaspekten dieser Arbeit verglichen.

Eine der bekanntesten Implementierungen von PIC-Verfahren für GPUs ist PIConGPU [1]. Aus diesem Grund dient der dazugehörige Artikel als Referenz für die nachfolgende Arbeit. PIConGPU simuliert mehrdimensionale PIC-Verfahren und verwendet hierfür eine verkettete Liste als Datenstruktur für die Partikel. In dieser Arbeit wird auf eine verkettete Liste verzichtet und die Minimierung der nicht koaleszenten Speicherzugriffe durch Sortierung mit Abstand an Zeitschritten versucht, welches hier bereits als mögliche Performanceoptimierung vorgeschlagen wurde. Ein weiterer Unterschied liegt in der Dimension. Wie bereits erwähnt simuliert PIConGPU mehrdimensionale PIC-Verfahren, wohingegen in den hier betrachteten Simulationen lediglich eindimensional simuliert wird. Auch werden keine expliziten Fusionierungen von Kernels erwähnt, welche in dieser Arbeit die besten Laufzeitverbesserungen erzielen werden.

Die zweite betrachtete wissenschaftliche Arbeit [8] behandelt ebenfalls eine Implementierung von PIC-Verfahren auf GPUs. Auch hier werden mehrdimensionale PIC-Verfahren simuliert, aber im Gegensatz zu PIConGPU werden die einzelnen Partikelkomponenten eindimensional abgespeichert und mit einer dem *Bucketsort* ähnlichen Art und Weise sortiert. Im Grunde wird diese Vorgehensweise in dieser Arbeit auch auf die eindimensionalen PIC-Verfahren angewendet. Ein weiterer Unterschied sind die untersuchten Architekturen. In dem Artikel wurden die Performancemessungen auf der Kepler- und der Maxwell-Architektur untersucht, wohingegen in dieser Arbeit die Performancemessungen zum einen auf der Maxwell-Architektur und zum anderen auf der Volta-Architektur durchgeführt werden.

3. Von der CPU zur GPU

Startpunkt dieser Arbeit ist die Analyse einer bestehenden, sequenziellen und mit MPI bzw. Pthreads parallelisierten Implementierung von unterschiedlichen Particle-In-Cell Codes für CPUs. In diesem Abschnitt werden die Grundlagen für die Implementierung von GPU-Codes basierend auf CPU-Codes behandelt. Insbesondere wird hier auf die nötigen Umformungen eingegangen, womit mathematische Standardoperationen parallel auf der GPU berechnet werden können.

3.1 Grundlagen der Particle In Cell Verfahren

In dieser Arbeit werden sowohl GPU-Versionen für das Vlasov-Poisson-System mit kartesischen Koordinaten (VPSS) als auch für das Vlasov-Einstein-System mit Eddison-Finkelstein (VEEF) und Schwarzschild Koordinaten (VESC) implementiert. Alle Varianten werden entsprechend dem nachfolgenden Algorithmus simuliert. Der Algorithmus der Particle-In-Cell-Simulation besteht im Allgemeinen aus fünf Schritten [7]:

1) Zuerst wird der Raum in Gitterzellen unterteilt und in jede Gitterzelle wird ein numerisches Teilchen gesetzt, welches die Masse bzw. Teilchenzahl in der jeweiligen Zelle repräsentiert. Dies wird nur zur Initialisierung der Simulation verwendet und in der Zeitschleife nicht ausgeführt.

2) Aus den numerischen Teilchen wird eine Approximation für die räumliche Massendichte und eventuell anderen benötigten räumlichen Dichten auf einem festen Gitter berechnet.

3) Aus den Approximationen für die räumlichen Dichten werden Approximationen für das Gravitationsfeld generiert.

4) Partikel werden entsprechend der Feldapproximationen auf dem Gitter bewegt.

5) Die Berechnung für den nächsten Zeitschritt beginnt bei Punkt 2.

Um die einzelnen Teilberechnungen parallel auf einer GPU berechnen zu können, müssen zuerst einige mathematische Operationen entsprechend umgeformt werden. So können Summen, Präfixsummen und andere Berechnungen, welche auf ein

vorher bereits bekanntes Ergebnis basieren, nicht äquivalent zum sequenziellen Code implementiert werden. Für GPUs werden dafür Reduktionen verwendet, wobei Berechnungen Bottom-up in einer Baumstruktur berechnet werden. Reduktionen existieren ebenfalls auf CPUs, werden hier aber nicht durch eine Baumstruktur umgesetzt. Um die Funktion von Reduktionen zu verdeutlichen, wird im Folgenden $\sum_{i=1}^{8} i$ per Reduktion berechnet. Um die acht Zahlen miteinander zu addieren,

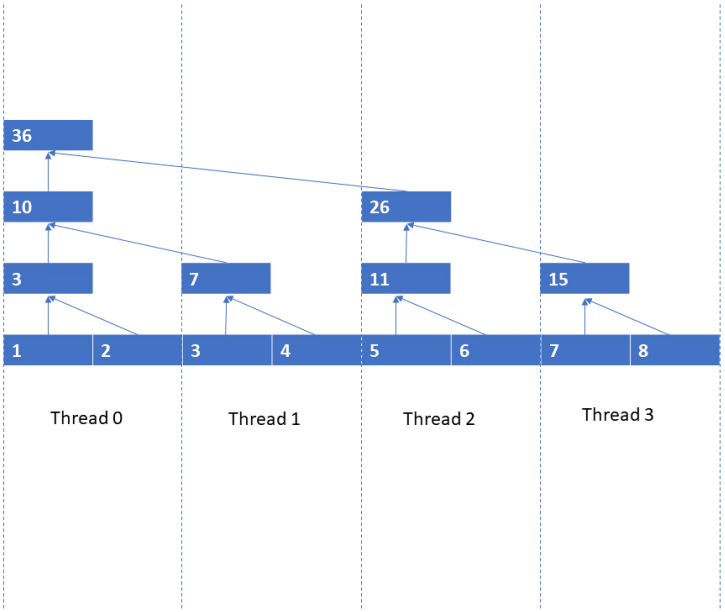


ABBILDUNG 3.1: Summenreduktion der Zahlen 1 bis 8

wird insgesamt ein Block mit 4 Threads benötigt. In jeder Ebene werden zwei Zahlen miteinander addiert, wodurch sich die Zahl der benötigten Threads halbiert. Auf diese Art und Weise berechnet Thread 0 des Blocks am Ende das Endergebnis. Bei größeren Summenberechnungen werden mehr Blöcke verwendet, um die GPU auslasten zu können. Dies hat zur Folge, dass zum Schluss pro Block ein Teilergebnis vorliegt. Diese Teilergebnisse müssen im Anschluss mittels erneuter Reduktion zu einer Gesamtsumme aufsummiert werden.

Für parallele Standardoperationen auf GPUs stehen zwei Bibliotheken zur Verfügung: Thrust [4] und CUB [5]. Um entscheiden zu können, welche man hiervon in den GPU-Implementierungen verwendet, wird die Laufzeit der Präfixsummenberechnung für beide Bibliotheken auf der Nvidia GTX 970 gemessen. Die technischen Daten der GPU sind im nachfolgenden Teilkapitel dargestellt.

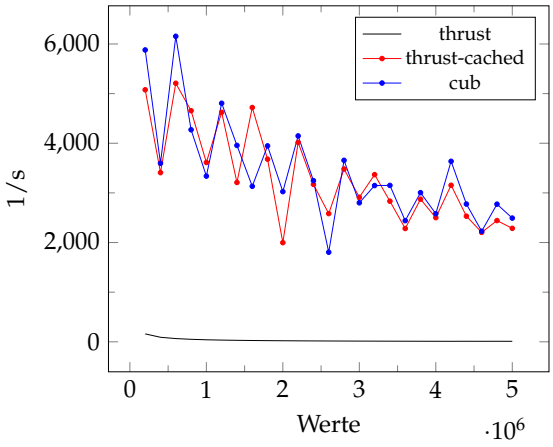


ABBILDUNG 3.2: CUB vs. Thrust

Die Messergebnisse sind im Diagramm 3.2 dargestellt. Es zeigt sich, dass die Standard-Präfixsummenberechnung der Thrust-Bibliothek die mit Abstand langsamste Variante zur Berechnung der Präfixsumme ist. Dies liegt an der Allokierung von temporärem Speicher bei jeder Präfixsummenberechnung. Um dies zu verhindern, wird der bereits implementierte *cached_allocator* [3] verwendet, welcher vor der Allokierung überprüft, ob entsprechend großer Speicher frei und bereits allokiert ist. Falls ein entsprechend großer Speicherblock existiert, wird dieser wiederverwendet und kein neuer Speicher allokiert. Da in der späteren Simulation die Partikelzahl über die Simulation hinweg konstant ist, wird nur im ersten Zeitschritt Speicher zur Präfixsummenberechnung allokiert und für die weiteren Zeitschritte wiederverwendet. Im Diagramm wird diese Variante durch *thrust-cached* dargestellt. Es zeigt sich, dass diese deutlich schneller als ohne *cached_allocator* ist. Bei der CUB-Präfixsummenberechnung muss der temporäre Speicher vorher allokiert werden, wodurch die Probleme mit der Thrust-Variante ohne Verwendung eines *cached_allocators* gelöst werden. Vergleicht man die Laufzeiten dieser beiden Varianten, so sind diese nahezu identisch. Interessant sind die Ergebnisse für größere Anzahlen von Zahlenwerten, da hier die CUB-Berechnung schneller als die Thrust-Variante ist. Da in der Simulation für gewöhnlich sehr große Partikelzahlen verwendet werden, ist die CUB-Präfixsummenberechnung für die Simulationen besser geeignet als die Thrust-Präfixsummenberechnung.

3.2 Datenblatt der benutzten GPUs

Für die Laufzeitmessungen werden unterschiedliche GPUs verwendet. In der nachfolgenden Tabelle werden die wichtigsten Daten der einzelnen GPUs dargestellt. Ein wichtiger Unterschied zwischen den GPUs ist die Architektur. Die GTX 970 und

Name:	SMs:	Takt:	Bandbreite:	Speicher:
GTX 970	13	1050 MHz/1178 MHz	224.4 GB/s	4 GB
GTX 980 Ti	22	1000 MHz/1076 MHz	336.6 GB/s	6 GB
Titan V	80	1200 MHz/1455 MHz	652.8 GB/s	12 GB

TABELLE 3.1: Daten der verwendeten GPUs [2]

GTX 980 Ti gehören zur Maxwell-Architektur, wohingegen die Titan V der Volta-Architektur angehört.

3.3 Vorgehen bei den Laufzeitmessungen

Um Performanceoptimierungen bewerten zu können, sind aussagekräftige Laufzeitmessungen nötig. Von Bedeutung sind hier sowohl die durchschnittlichen Laufzeiten der zeitkritischen Kernels als auch die durchschnittlichen Laufzeiten der gesamten Simulation. Hierfür werden alle Laufzeitmessungen zehnmal durchgeführt und gemittelt. Um die durchschnittlichen Zeiten der Kernels zu erhalten, wird der Visual-Profiler bzw. Nvidia-Nsight verwendet. Um die Dauer der gesamten Simulation zu messen, wird die Ausführungszeit der Zeitschleife gemessen, da diese den gesamten performancekritischen Code enthält. Hierfür wird die *high-resolution-clock* aus der C++-Bibliothek *chrono* verwendet. Dass alle Kernels auf der GPU die Berechnungen beendet haben, wird vor dem Messen der Endzeit mit dem Befehl *cudaDeviceSynchronize()* sichergestellt. Bei Skalierung der Partikelzahl mit einem konstanten Faktor wird zugleich der Radius der Gitterzellen durch diesen Faktor dividiert. Somit skaliert die Zahl der Gitterpunkte mit der Zahl der Partikel.

4. Implementierung einer GPU Version

Im Folgenden wird eine erste GPU-Implementierung der PIC-Verfahren entwickelt. Diese soll im weiteren Verlauf dieser Arbeit als Ausgangspunkt für Optimierungen und Performancevergleiche dienen und wird in den nachfolgenden Kapiteln als Standardversion bezeichnet. Insbesondere beinhaltet diese Version bereits alle Standardoptimierungen. Dies sind Optimierungen, welche sich nicht durch den jeweiligen Programmtyp unterscheiden, sondern essenziell für ein effektives GPU-Programm sind. Besonders wichtig ist hier das *Global Memory Coalescing* und die Minimierung von Speicheroperationen wie *cudaMalloc*, *cudaMemcpy* und *cudaFree*. Außerdem ist die Vermeidung von divergentem Kontrollfluss von Bedeutung.

4.1 Koaleszenter Speicherzugriff

Ein Schlüsselfaktor bei der Optimierung von GPU-Programmen ist die Optimierung der Lese- und Schreibzugriffe. Um hier die bestmögliche Performance zu erhalten, ist es notwendig, ebendiese Speicherzugriffe koaleszent zu gestalten. Dies ist genau dann der Fall, wenn die drei nachfolgenden Kriterien erfüllt sind.

Zuerst muss sichergestellt sein, dass die einzelnen Worte mindestens 32 Bit lang sind. Das Lesen und Schreiben von kürzeren Worten ist nie koaleszent. Dieses Kriterium ist hier immer erfüllt, da entweder mit einfacher oder doppelter Genauigkeit gerechnet wird. Des Weiteren ist es erforderlich, dass die Adressen, auf welche die Threads im Warp zugreifen, kontinuierlich aufsteigend im Speicher liegen. Zu guter Letzt muss der Zugriff auf die Basisadresse, auf welcher der erste Thread im Warp zugreift, schematisch ausgerichtet sein. So benötigt man z. B. bei 32 Bit Worten eine 64-Byte-Ausrichtung und bei 64 Bit Worten eine 128-Byte-Ausrichtung. Sobald eines dieser Kriterien nicht erfüllt ist, gelten die Speicherzugriffe als nicht koaleszent und sorgen für eine Performanceeinbuße, da man nicht mehr alle Speicherzugriffe in einem Warp zusammen ausführen kann [vgl. 9, S. 143-147].

Wichtig sind außerdem die Auswirkungen von lesenden und schreibenden Speicherzugriffen auf dieselbe Speicheradresse. Angenommen alle Threads in einem Warp benötigen Daten von derselben Speicheradresse. Bei einem lesenden Zugriff

wird die benötigte *cache-line* des ersten Threads angefordert, welcher noch auf Daten wartet. Danach wird überprüft, ob benötigte Daten von anderen Threads in dieser *cache-line* liegen. Falls dies der Fall ist, erhalten diese die Daten ohne weiteren Speicherzugriff. Im Gegensatz hierzu steht das Verhalten bei schreibenden Zugriffen auf dieselbe Speicheradresse. Um inkonsistentes Schreiben von Daten zu vermeiden werden hierfür atomare Operationen z. B. *atomicAdd()* benötigt. Diese wiederum sorgen für eine Seqenzialisierung der Speicherzugriffe und verlangsamen somit das betrachtete System. Koaleszente Speicherzugriffe werden für die nachfolgenden Optimierungen noch von entscheidender Bedeutung sein, insbesondere in Bezug auf die hier vorgestellten Wechselwirkungen zwischen lesenden und schreibenden Zugriffen auf dieselben Speicheradressen.

4.2 Minimierung der Speicheroperationen zwischen CPU und GPU

Elementarer Bestandteil von GPU-Implementierungen ist die Minimierung der Datentransfers zwischen Host und GPU. Dies liegt besonders an der Bandbreite der PCI-Express Schnittstellen, welche mit 15,754 GByte/s bei PCI-Express 3.0 mit 16 Lanes die langsamste Verknüpfung zwischen Host und GPU darstellen. Um die Datentransfers zu optimieren, welche mit der Funktion *cudaMemcpy* ausgeführt werden, ist es erforderlich, alle nur von der GPU verwendeten Daten zu Beginn der Simulation auf die GPU zu kopieren und nur die benötigten Daten am Ende der Simulation wieder auf den Host zu kopieren.

Ein weiterer Aspekt ist das Generieren von Logdateien. Bisher wurden in jedem Zeitschritt Ergebnisse in die Logdateien geschrieben. Um dieselben Logdateien auf diese Weise auf der GPU zu generieren, müssen die Daten zuerst per *cudaMemcpy* auf den Host übertragen werden und dann in der jeweiligen Datei gespeichert werden. Dies ist ineffizient, weswegen die Daten entweder in jedem Zeitschritt gesammelt und am Ende des Zeitschritts zusammen auf den Host kopiert werden oder entsprechender Speicher auf der GPU allokiert wird, sodass die Ergebnisse pro Zeitschritt bis zum Ende der Simulation auf der GPU gespeichert und zum Schluss mit einem *cudaMemcpy* auf den Host kopiert werden können.

Des Weiteren muss das Allokieren und Freigeben von Speicher auf der GPU außerhalb der Zeitschleife stattfinden, da *cudaMalloc* und *cudaFree* zu den zeitkritischsten Operationen aus dem CUDA-Framework gehören. Dies ist bei den Partikeldaten einfach realisierbar, da sich die Anzahl der Partikel im Laufe der Simulation nicht ändert. Da vor der Simulation nicht bekannt ist, wie sich die Partikel währenddessen bewegen, ist die genau benötigte Speichergröße der Felder für die Felddaten ebenfalls unbekannt. Ein erster Ansatz ist hier die grobe Abschätzung der maximal möglichen Ausdehnung, was allerdings zu zwei Problemen führen kann. Zum einen

kann die Abschätzung zu klein sein, wodurch das Programm auf nicht allokierten Speicher zugreift und abstürzt, zum anderen kann die Abschätzung zu großzügig sein, wodurch mehr Speicher als nötig allokiert wird und unter Umständen für eine Limitierung der Partikelzahl bzw. der Feldgröße sorgt. Eine bessere Alternative ist hier das Ausführen der Simulation für den Partikelfaktor eins und dem Speichern der maximalen Ausdehnung der Partikel in diesem Simulationsdurchlauf. Weil sich das Verhalten bei größeren Partikelzahlen nicht ändert, da die maximale Ausdehnung sich bei feineren Gittern nur um ebendiesen Partikelfaktor erhöht, ist es so möglich, immer die maximale Ausdehnung an Gitterpunkten zu berechnen.

4.3 Vermeidung von divergentem Kontrollfluss

Unter divergentem Kontrollfluss in Bezug auf GPU-Programme versteht man die durch bedingte Verzweigung verursachte Ausführung von unterschiedlichen Instruktionen innerhalb eines Warps. Bei einer kurzen, bedingten Sequenz an Instruktionen z. B. einer if-Bedingung ohne else verwendet der Compiler Predication, wobei je nach Vorhersage die Codesequenz entweder ausgeführt oder nicht ausgeführt wird [vgl. 9, S. 267-269]. Konkret in den Simulationen wird z. B. in den propagate_particles Funktionen überprüft, ob Partikel einen gewissen Mindestabstand von der ersten Gitterzelle haben. Diese Sequenz an Instruktionen ist klein genug, sodass sie mittels Predication ausgeführt werden kann.

5. Profiling der GPU-Codes zur Identifizierung von Bottlenecks

Der erste Schritt bei der Performanceoptimierung ist die Identifizierung von Flaschenhälsen und damit das Finden von Startpunkten für mögliche Optimierungen. Hierfür ist es erforderlich, die einzelnen Kernellaufzeiten zu analysieren, um diejenigen zu finden, welche den größten Anteil der Laufzeit benötigen. Das Diagramm 5.1 zeigt die prozentualen Anteile der jeweiligen Berechnungen an der Gesamtlaufzeit für ca. 15 Millionen Partikel und 1000 Zeitschritte.

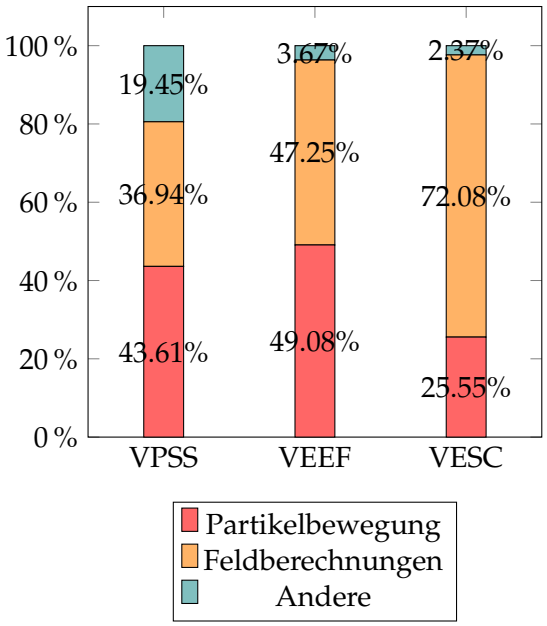


ABBILDUNG 5.1: Prozentuale Laufzeiten der zeitkritischen Kernels ohne Optimierungen

Es zeigt sich, dass die Berechnung der Partikelbewegung und der Felder die mit Abstand zeitaufwendigsten Berechnungen darstellen. So beanspruchen diese bei der VPSS-Simulation 80,55%, bei der VEEF-Simulation 96,33% und bei der VESC-Simulation 97,63%. Für die weitere Analyse wird nur die VEEF-Simulation betrachtet, da die Ergebnisse für die anderen beiden Simulationen analog sind.

Betrachtet man konkrete Kernels, so sind die zeitkritischsten Kernels der VEEF-Simulation propagate_particles mit 49,08% und comp_t_s mit 47,25%, was den Felddberechnungen entspricht. Zuerst muss man für diese beiden Kernels herausfinden, was die Performance limitiert. Es zeigt sich, dass beide Kernels durch die Latenz von arithmetischen und speicherbasierten Operationen limitiert werden. Das bedeutet, dass die Performance von propagate_particles und comp_t_s limitiert ist durch den Durchsatz von arithmetischen Operationen und der genutzten Speicherbandbreite. Analysiert man nun das Verhältnis zwischen Speicherzugriffen und arithmetischen Operationen, so ist eine Limitierung durch die Speicherbandbreite unter idealen Bedingungen realistisch. Ideal bedeutet hier bei aufgelösten Abhängigkeiten und koaleszentem Speicherzugriffsmuster.

Der nächste Schritt ist die Analyse des bisherigen GPU-Codes, um konkrete Programmstellen zu identifizieren, welche die Performance beeinträchtigen. Da das Ziel die Optimierung von Speicherzugriffen ist, werden im Folgenden die zeitkritischen Kernels auf Speicherabhängigkeiten und nicht koaleszente Speicherzugriffe untersucht. Abbildung 5.2 zeigt eine Übersicht über prop_particles.

```
1         const auto rr = data_r[j];
2         const auto gg = data_ang2[j];
3         const auto ww = data_w[j];
4
5         const int intr = ceilf(rr / drfield);
6
7         const auto b_left = b[intr - 1];
8         const auto b_right = b[intr];
9         const auto alpha_left = alpha[intr - 1];
10        const auto alpha_right = alpha[intr];
11        const auto t_left = T11false[intr - 1];
12        const auto t_right = T11false[intr];
13        const auto rho_left = rhofalse[intr - 1];
14        const auto rho_right = rhofalse[intr];
15
16        /*...do calculations*/
17
18        data_r[j] = r_new;
19        data_w[j] = w_new;
20        data_ang2[j] = ang2_new;
```

ABBILDUNG 5.2: Übersicht propagate_particles

In Zeile 5 ist die Berechnung von intr, dem Index der zugehörigen rechten Gitterzelle zum Partikel j, abhängig von der Gitterposition rr, welche erst aus dem Speicher geladen werden muss. Von Zeile 7 – 14 wird der Index intr zum Indizieren der Felddaten benötigt. Dies bedeutet, dass sowohl die Berechnung von intr als auch das Laden der Felddaten warten muss, bis rr aus dem Speicher geladen ist. Dies stellt eine nicht auflösbare Speicherabhängigkeit dar. Abgesehen von dieser Speicherabhängigkeit ist außerdem das Laden der Felddaten von Zeile 7-14 nicht koaleszent,

da sich die Partikel im Laufe der Simulation durch das Gitter bewegen, ihre Positionen im Speicher allerdings nicht verändert werden. Da nicht koaleszente Speicheroperationen einen immensen Einfluss auf die Performance haben, liegt in der Partikelsortierung und der damit verbundenen Lokalitätserhöhung der Partikel der erste Ansatz zur Performanceoptimierung von propagate_particles.

```
1         const auto rr = data_r[j];
2         const auto ww = data_w[j];
3         const auto gg = data_ang2[j];
4         const auto ff = data_f[j];
5
6         const int intr = ceilf(rr / drfield);
7
8         /*...do calculations*/
9
10        atomicAdd(&sfalse[intr], s_right);
11        atomicAdd(&sfalse[intr - 1], s_left);
12
13        atomicAdd(&t11_false[intr], t_right);
14        atomicAdd(&t11_false[intr - 1], t_left);
```

ABBILDUNG 5.3: Übersicht comp_t_s

Als nächstes wird comp_t_s anhand des Codeausschnitts 5.3 betrachtet. Die in propagate_particles identifizierte Speicherabhängigkeit ist hier wieder in Zeile 6 zu finden. Einen Unterschied im Vergleich zu propagate_particles stellen hier die atomaren Additionen dar. Problematisch sind diese in Bezug auf die vorgeschlagene Partikelsortierung, da bei einem schreiben Zugriff von mehreren Threads auf dieselbe Speicheradresse ebendiese sequenzialisiert werden und damit die Performance negativ beeinflussen. Da die Zahl an Gitterzellen sehr viel kleiner ist als die Zahl der Partikel, lässt sich diese Sequenzialisierung bei einer perfekten Sortierung nicht verhindern.

Im späteren Verlauf dieser Arbeit wird diese Wechselwirkung zwischen propagate_particles und comp_t_s mit konkreten Messungen belegt und behandelt. Hierfür ergibt sich ein Tradeoff zwischen dem Maß an Sortierung und der korrespondierenden Performance von propagate_particles und comp_t_s. Eine weitere Möglichkeit der Optimierung, welche sich unabhängig von den Daten des Visual-Profilers ergibt, ist die Fusionierung von Kernels. Hierbei wird die Anzahl an durchlaufenen Schleifeniterationen pro Zeitschritt durch eine Verschmelzung von Kernels minimiert.

6. Ansätze zur Optimierung der Performance

In diesem Kapitel werden die theoretischen Hintergründe der verschiedenen Optimierungsansätze erläutert. Zuerst wird der Ansatz der Partikelsortierung betrachtet. Hierfür werden die Partikel zum einen mit Radixsort und zum anderen mit Countingsort sortiert. Ein weiterer Ansatz ist die Kernelfusionierung, wodurch man die Zahl der Schleifendurchläufe minimiert.

6.1 Sortierung mittels Radixsort

Die Implementierung des verwendeten Radixsorts stammt aus der Thrust-Bibliothek. Es existiert auch eine Implementierung in der CUB-Bibliothek, welche hier aber nicht sinnvoll verwendbar ist, da nicht mehr als zwei Partikelkomponenten simultan sortiert werden können. Das Ziel ist die Sortierung der vier Partikelkomponenten nach der Gitterposition r. Da drei Partikelkomponenten entsprechend r sortiert werden, eignet sich hierfür am besten der Thrust::sort_by_key Algorithmus. Hierbei spezifiziert man Start- und Zieladresse der Schlüssel und die Startadresse der zu sortierenden Werte. Bisher kann lediglich eine Komponente nach r sortiert werden, weshalb ein Zusammenfügen der drei Partikelkomponenten nötig ist, sodass man drei unabhängige Sequenzen zu einer Sequenz umwandelt, welche Tupel mit den drei Partikelkomponenten enthält. Realisiert wird dies mit dem Thrust::zip_iterator und dem Befehl Thrust::make_tuple(). Weil die Position im Gitter durch eine Fließkommazahl dargestellt wird, werden beim Radixsort viele *Buckets* benötigt, entsprechend der Zahl der unterschiedlichen r-Werte. Effizienter ist hier die Verwendung von ganzzahligen Schlüsseln, in diesem Fall der zu der Gitterposition r gehörige rechts liegende Gitterpunkt. Die verwendete Sortierung ist nicht stabil d.h. die Reihenfolge von gleichen Schlüsselwerte ist nach der Sortierung unter Umständen anders als vor der Sortierung, was aber kein Problem darstellt, da nur die Zugehörigkeit zu den Gitterzellen von Bedeutung ist.

Wie bereits bei der Präfixsumme der Thrust-Bibliothek erläutert, wird auch hier bei jedem Aufruf der Sortierung temporärer Speicher angelegt und wieder freigegeben. Aus diesem Grund verwendet man auch hier den *cached_allocator*, um nur bei der

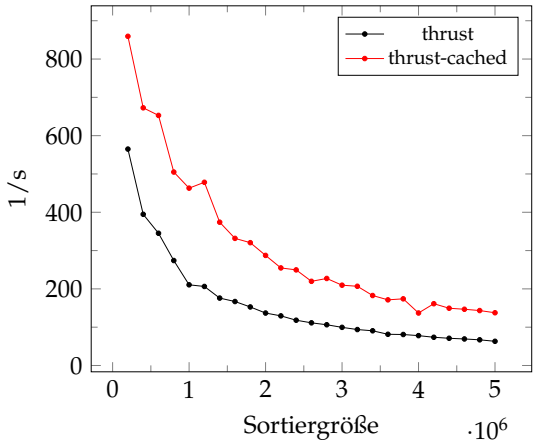


ABBILDUNG 6.1: Thrust Radixsort-Cached und Nicht-Cached

ersten Sortierung Speicher zu allokieren. Den Performanceunterschied zeigt Abbildung 6.1. Es zeigt sich, dass der Cached-Radixsort für alle getesteten Sortiergrößen schneller ist, als der Standard-Radixsort ohne *cached_allocator*.

6.2 Sortierung mittels Countingsort

Der parallele Countingsort wurde im Rahmen diese Arbeit selbst implementiert, da er bisher in keiner Bibliothek zur Verfügung steht. Der Algorithmus besteht aus vier Schritten. Zuerst muss ein Histogramm erstellt werden, welches die Anzahl der Partikel beinhaltet, die zur selben Gitterzelle gehören. Zur späteren Berechnung der neuen Indizes wird außerdem der lokale Index der Partikel innerhalb einer Gitterzelle benötigt. Als nächstes berechnet man ausgehend vom Histogramm mittels exklusiver Präfixsumme den Indexbereich für jede Gitterzelle. Addiert man nun den lokalen Index des Partikels auf den Startindex der zugehörigen Gitterzelle, so erhält man den neuen Index für das Partikel, mit welchem die Partikelkomponenten sortiert im Speicher abgelegt werden können. Da die Partikelkomponenten beim Umsortieren in temporärem Speicher abgelegt werden, müssen diese zum Schluss zurück in den primären Speicher kopiert werden.

6.3 Kernelfusionierung mittels Abhängigkeitsgraphen

Der nächste Ansatz zur Optimierung der Performance ist die Reduzierung von Schleifendurchläufen durch die Fusionierung von Kernels. Damit man Kernels miteinander verschmelzen kann, müssen diese die gleiche Zahl an Berechnungen durchführen, da sonst ein divergenter Kontrollfluss entsteht. Am einfachsten lässt sich die Kernelfusionierung bei voneinander unabhängigen Berechnungen umsetzen, da man hier die Berechnungen nur zusammenfassen muss. Dies ist in Abbildung 6.2 dargestellt. Es werden von Kernel a und Kernel b jeweils acht Berechnungen durch-

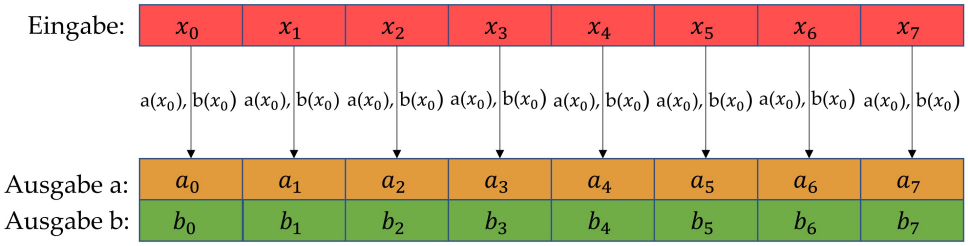


ABBILDUNG 6.2: Fusionierung unabhängiger Berechnungen

geführt. Da die Berechnungen in a und b unabhängig voneinander sind, können diese in einer beliebigen Reihenfolge mit nur einem Kernel berechnet werden.

Auch voneinander abhängige Berechnungen können unter bestimmten Voraussetzungen miteinander fusioniert werden. Ein Beispiel hierfür zeigt Abbildung 6.3. Angenommen die Berechnung b() benötigt das Ergebnis von a(). Hier ist es erfor-

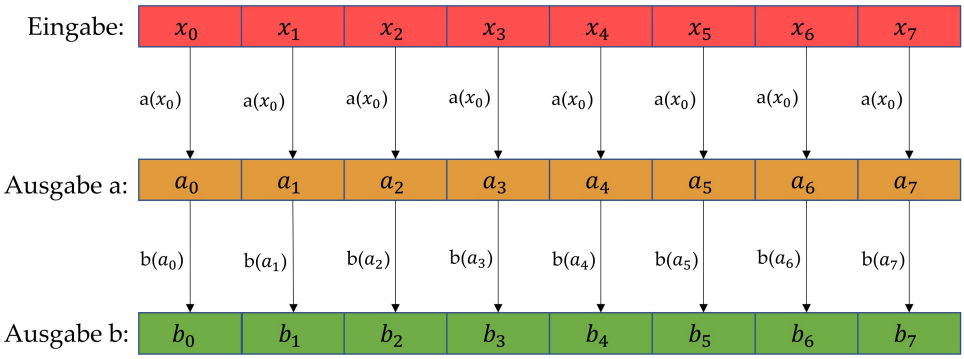


ABBILDUNG 6.3: Fusionierung abhängiger Berechnungen

derlich, dass das benötigte a_x von dem Thread berechnet wird, welcher anschließend b_x berechnet. Grundsätzlich lassen sich so nur lokale Abhängigkeiten auflösen. Benötigt man z. B. für b() die Präfixsumme der Ergebnisse von a(), so kann keine Kernelfusionierung stattfinden. Die einfachste Art und Weise, voneinander abhängige bzw. unabhängige Berechnungen zu identifizieren, ist das Konstruieren von Kontrollfluss- bzw. Abhängigkeitsgraphen. Dabei werden alle Kernels in einen Graphen eingezeichnet und basierend auf den Abhängigkeiten mit anderen Kernels verbunden. Mögliche Verschmelzungen werden mit derselben Farbe im Graphen

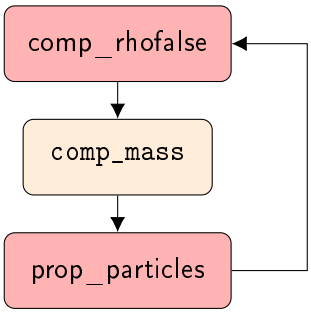


ABBILDUNG 6.4: Kernelabhängigkeiten der VPSS-Simulation

dargestellt. Abbildung 6.4 enthält den Graphen für die VPSS-Simulation. Aus dem Graph ist ersichtlich, dass comp_rhofalse und prop_particles miteinander fusioniert werden können, da beide Berechnungen nur lokal voneinander abhängen und die Berechnung in beiden Kernels für alle Partikel durchgeführt wird.

Abbildung 6.5 zeigt eine Übersicht über die Kernelabhängigkeiten in der VEEF-Simulation. Wie man dem Diagramm entnehmen kann, kann man die unabhängi-

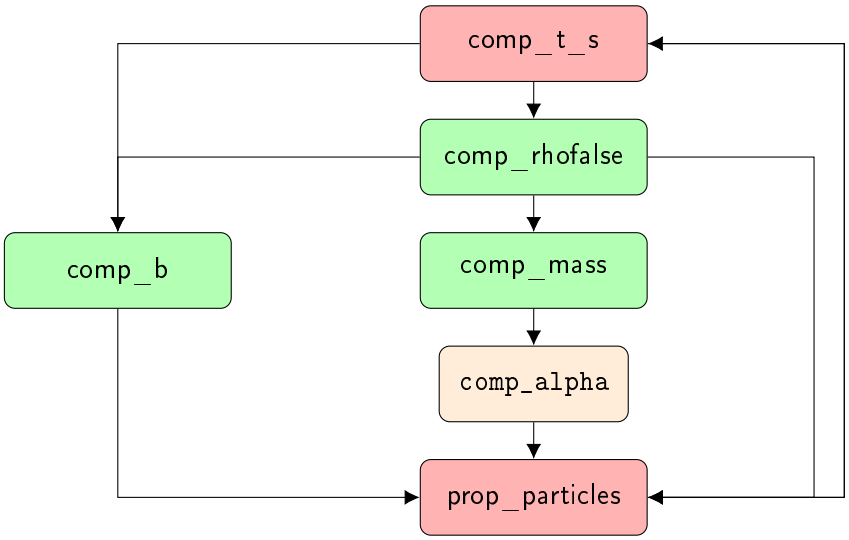


ABBILDUNG 6.5: Kernelabhängigkeiten der VEEF-Simulation

gen Feldberechnungen comp_b, comp_rhofalse und die lokal von comp_rhofalse abhängige Feldberechnung comp_mass miteinander verschmelzen. Für comp_alpha ist die Präfixsumme über die Masse erforderlich, weswegen man diese Berechnung nicht mit den anderen Feldberechnungen fusionieren kann. Wie man in dem Graph außerdem sehen kann, gibt es zwischen comp_t_s und prop_particles eine wechselseitige lokale Abhängigkeit. Um diese aufzulösen ist es erforderlich, die neuen Werte von t zwischenzuspeichern, da die alten Werte in der Berechnung von unterschiedlichen Threads benötigt werden. Diese werden nach der Beendigung von prop_particles zurückkopiert.

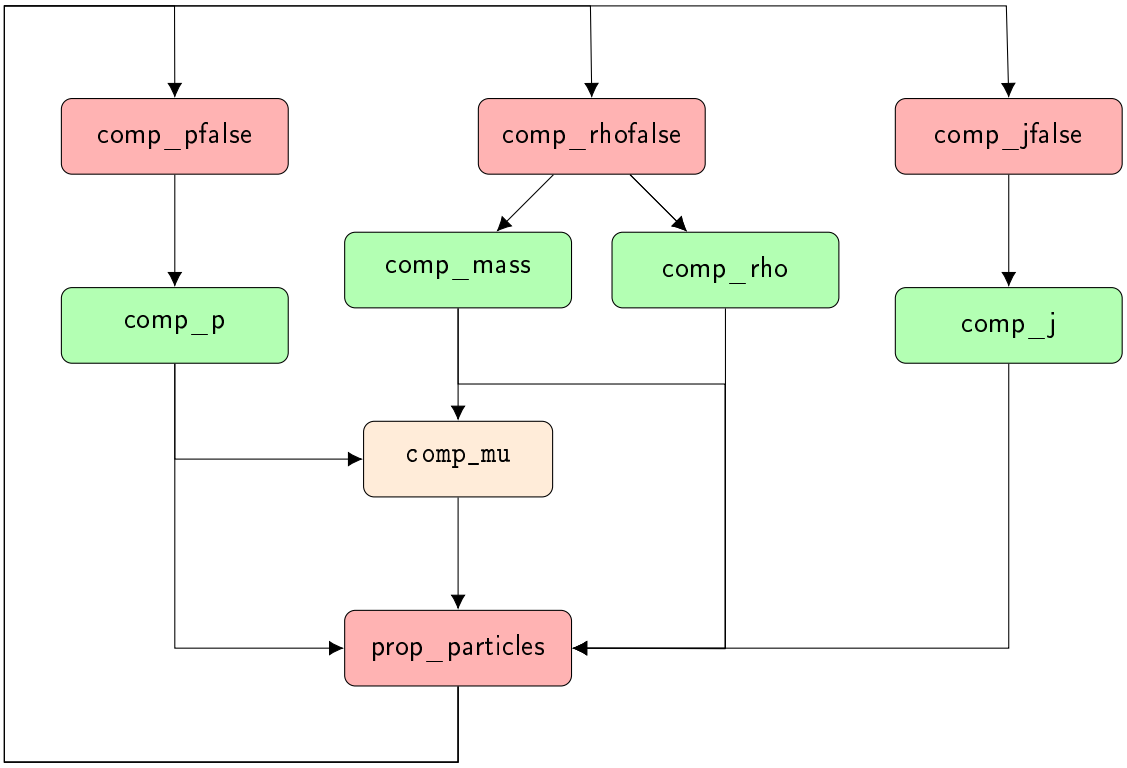


ABBILDUNG 6.6: Kernelabhängigkeiten der VESC-Simulation

Abbildung 6.6 zeigt die Kernelabhängigkeiten der VESC-Simulation. Die Struktur des Diagramms ähnelt sehr stark dem der VEEF-Simulation in Abbildung 6.5 und dem der VPSS-Simulation in Abbildung 6.4. Zuerst lassen sich hier die unabhängigen Feldberechnungen comp_p, comp_mass, comp_rho und comp_j zu einem Kernel fusionieren. Man kann die Feldberechnung comp_mu nicht mit den anderen Feldberechnungen verschmelzen, da diese als Eingabe die Präfixsumme über die Masse benötigt. Für die Verschmelzung von prop_particles mit den Feldberechnungen comp_pfalse, comp_rhofalse und comp_jfalse benötigt man hier kein temporäres Zwischenspeichern, da keine wechselseitigen Abhängigkeiten existieren.

7. Bewertung der optimierten GPU-Codes

Im Folgenden werden die Ergebnisse der Laufzeitmessungen der optimierten Kernels und der gesamten Simulationen mit den Ergebnissen der ersten GPU-Codes verglichen und bewertet. Verwendet wird hierfür die Nvidia Geforce GTX 970.

7.1 Vlasov-Einstein- und Vlasov-Poisson-Simulation mit Sortierung

Die erste Optimierung, welche betrachtet wird, ist das Sortieren der Partikel. Dies wurde zum einen per Radixsort durch den Thrust::sort_by_key-Algorithmus und zum anderen durch einen selbst implementierten Countingsort-Algorithmus durchgeführt.

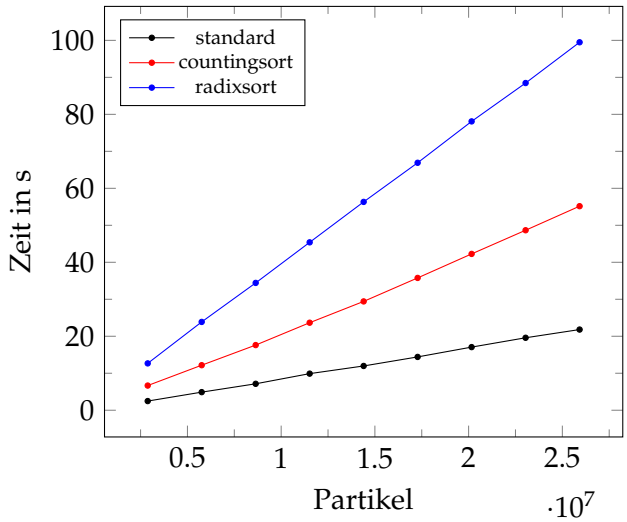


ABBILDUNG 7.1: Sortierung der Partikel in jedem Zeitschritt

Abbildung 7.1 zeigt die Sortierung der Partikel in jedem Zeitschritt. Da die Ergebnisse für die VESC- und VPSS-Simulation analog zu denen der VEEF-Simulation sind und die Darstellung nur zur Veranschaulichung dient, wird hier nur die VEEF-Simulation dargestellt. Auf den ersten Blick verlangsamen beide Sortierverfahren die Simulation. Um herauszufinden, was genau diese Verlangsamung bewirkt, werden die Kernellaufzeiten mittels Visual-Profiler bzw. Nvidia-Nsight ermittelt.

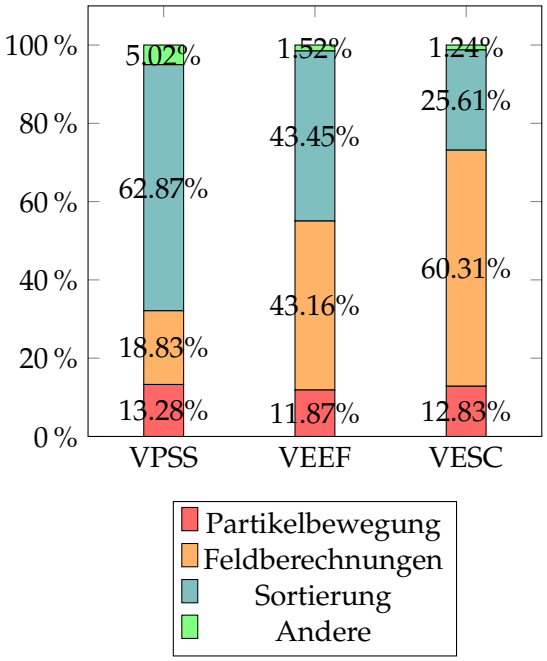


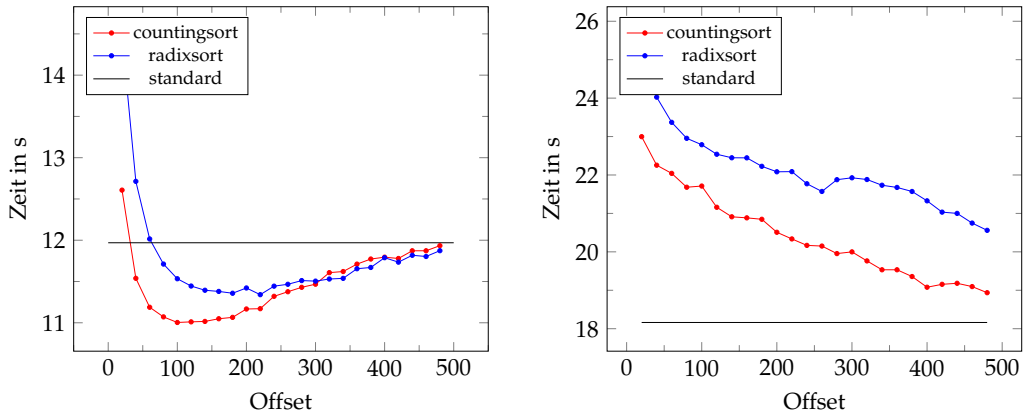
ABBILDUNG 7.2: Prozentuale Laufzeiten der zeitkritischen Kernels bei Sortierung

Das Diagramm 7.2 stellt die prozentualen Laufzeiten der Kernels für ca. 15 Millionen Partikel und 1000 Zeitschritte bei Sortierung mittels Countingsort über die gesamte VPSS- , VEEF- und VESC-Simulation dar. Der Anteil an Laufzeit, welcher für die Sortierung benötigt wird, ist für die hier durchgeführte Betrachtung unwichtig. Im Unterschied zu Abbildung 5.1 ergibt sich nach der Sortierung in jedem Zeitschritt bei der VEEF-Simulation ein Anteil von 11,87% für die Partikelbewegung und 43,16% für die Feldberechnungen. Somit benötigen diese nun prozentual eine viermal größere Laufzeit in Bezug auf die Partikelbewegung als in der Standardversion. In der VESC-Simulation ergibt sich ein Anteil von 12,83% für die Partikelbewegung und 60,31% für die Feldberechnungen. Hier ist die Berechnung der Partikelbewegung fünfmal schneller als die Feldberechnungen, wohingegen bei der unsortierten Variante die Partikelbewegung lediglich dreimal schneller als die Feldberechnung ist. Bei der VPSS-Simulation benötigen die Partikelbewegung mit 13,28% und die Feldberechnungen mit 18,83% in etwa dieselbe Laufzeit, sodass sich das Verhältnis der beiden Berechnungen durch Sortierung nicht verändert. Betrachtet man die Simulationsvarianten, so ist hier ein großer Unterschied in der anteiligen Laufzeit der Feldberechnungen festzustellen. Dieser ist auf die unterschiedliche Zahl an atomaren Operationen in den Feldberechnungen zurückzuführen. Bei der VPSS-Simulation benötigt man zwei atomare Additionen, bei der VEEF-Simulation vier und bei der VESC-Simulation sechs. Betrachtet man nun das Diagramm 7.2, so lassen sich diese atomaren Additionen direkt auf die prozentualen Laufzeiten der Feldberechnungen abbilden. Diese betragen 18,83% bei der VPSS-, 43,16% bei der VEEF- und 60,31% bei der VESC-Simulation. Eine atomare Addition entspricht demnach in etwa 10% der Laufzeit der Feldberechnungen der jeweiligen Simulation. Betrachtet man nun das

Verhältnis zwischen der Zahl an lesenden Speicherzugriffen in der Partikelbewegung und der Anzahl an atomar schreibenden Speicherzugriffen in den Feldberechnungen, so erhält man für die VPSS-Simulation ein Verhältnis von 1:1, bei der VEEF-Simulation von 2:1 und bei der VESC-Simulation von 1,7:1. Das Verhältnis zwischen lesenden und schreiben Zugriffen stellt das Verhältnis von Speicherzugriffen dar, welche in Bezug auf die Performance zum einen durch ein koaleszentes Speicherzugriffsmuster profitieren und zum anderen dadurch verlangsamt werden. Dies bedeutet, dass je größer dieses Verhältnis ist, desto besser eignet sich die Sortierung als Performanceoptimierung. Somit hat die Sortierung bei der VEEF-Simulation das größte Potential für eine Performanceverbesserung.

Eine weitere Möglichkeit, mit welcher die Sortierung weiter optimiert werden kann, ist das Sortieren der Partikel mit einem Offset an Zeitschritten. Dies bedeutet, dass die Partikel nicht mehr in jedem Zeitschritt sortiert werden, sondern jeweils ein bestimmter Abstand x an Zeitschritten zwischen den Partikelsortierungen liegt. Dadurch sind die Partikel nur noch im Zeitschritt t % x = 0 perfekt sortiert, wodurch eine Sequenzialisierung von atomaren Additionen bei comp_fields vermindert wird. Comp_fields steht hier für die simulationsabhängigen Feldberechnungen, da hier die genaue Bezeichung wie in Kapitel fünf und sechs überflüssig ist. Auf der anderen Seite beeinflusst die Sortierung mittels Offsets die Performance von propagate_particles negativ, da diese von einem koaleszentem Speicherzugriffsmuster profitiert. Es ist demnach wichtig, den Offset so zu wählen, dass die Partikel zum einen nicht perfekt sortiert sind, um eine Sequenzialisierung der atomaren Operationen von comp_fields zu vermindern, zum anderen aber trotzdem so sortiert sind, dass die Performance von propagate_particles von der höheren Partikellokalität profitieren kann. Zusammengefasst muss die kumulierte Laufzeit von comp_fields und propagate_particles minimal sein. Hierfür erforderlich ist eine Ausführung der Simulation mit unterschiedlichen Sortieroffsets, um den perfekten Offset zu finden.

Dies wird für die Countingsort- und Radixsort-Variante der VESC-, VEEF- und VPSS-Simulation mit ca. 15 Millionen Partikeln und 1000 Zeitschritten durchgeführt. Zuerst betrachtet werden die Ergebnisse der Vlasov-Einstein-Simulationen, die durch die Diagramme in Abbildung 7.3 visualisiert werden. Wie man den Diagrammen entnehmen kann, hat der Offset an Zeitschritten einen direkten Einfluss auf die Performance der gesamten Simulation. Für die VEEF-Simulation mit Countingsort nimmt die Laufzeit bis zu einem Offset von 100 Zeitschritten ab, wohingegen dies beim Radixsort bis zu einem Offset von 220 der Fall ist. Danach nähern sich die Versionen dann der Laufzeit der Standardvariante an. Dies bedeutet, dass es für die sortierten Varianten der VEEF-Simulation einen perfekten Abstand an Zeitschritten gibt, sodass die Laufzeit minimal wird. Für die VESC-Simulation zeigt sich, dass es keinen perfekten Offset gibt, da die Laufzeit sich von Beginn an der Laufzeit der



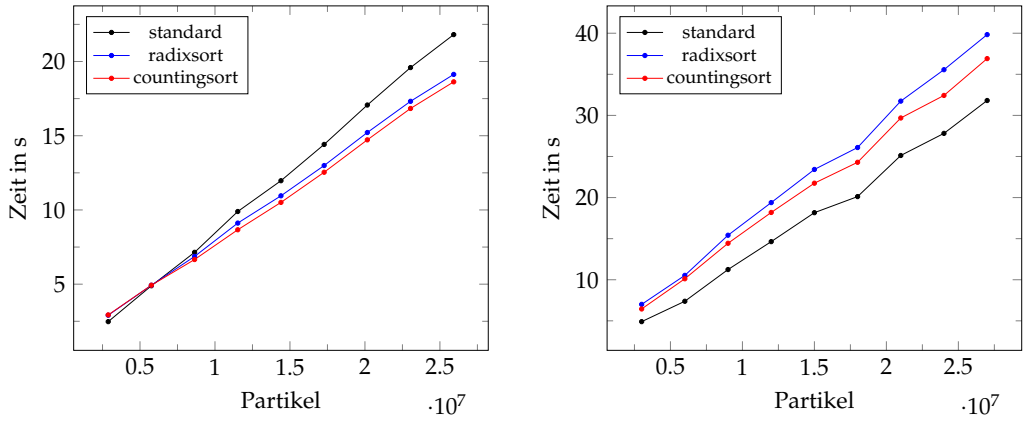
(A) Laufzeit der VEEF-Simulation bei unterschiedlichen Sortieroffsets

(B) Laufzeit der VESC-Simulation bei unterschiedlichen Sortieroffsets

ABBILDUNG 7.3: Sortierung auf GTX 970

Standardversion annähert. Weitergehend bedeutet dies, dass die Sortierung die Simulation nicht beschleunigt, sondern ausbremst. Da dies unterschiedlich ist für verschiedene Simulationskonfigurationen, kann diese Erkenntnis nicht verallgemeinert werden und muss für jede Simulationskonfiguration erneut überprüft werden.

Die nachfolgenden Diagramme in Abbildung 7.4 stellen die Laufzeiten für verschiedene Partikelzahlen dar, welche mit Offset sortiert werden. Bei der VEEF-Simulation werden für den Countingsort 100 Zeitschritte und für den Radixsort 220 Zeitschritte als Offset gewählt, da mit diesem Offset die Laufzeiten der beiden Sortiervarianten minimal werden. Bei der VESC-Variante wird ein willkürlicher Offset von 60 gewählt, da das Diagramm nur verdeutlichen soll, dass das Sortieren keinen Vorteil bringt. Beide Diagramme unterstützen die bisherigen Ergebnisse in den vorange-



(A) Laufzeit der VEEF-Simulation bei unterschiedlichen Partikelzahlen

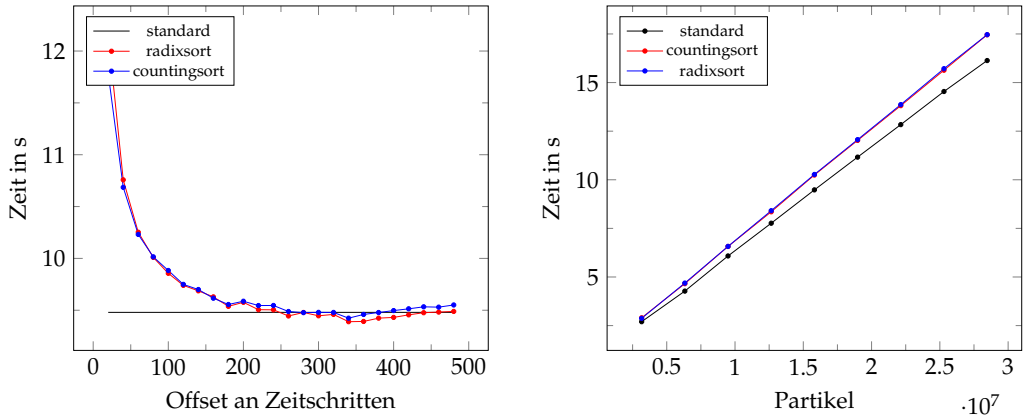
(B) Laufzeit der VESC-Simulation bei unterschiedlichen Partikelzahlen

ABBILDUNG 7.4: Sortierung auf GTX 970

henden Diagrammen. So ist bei der VESC-Simulation die Laufzeit der Standardversion die mit Abstand schnellste Implementierung. Bei der VEEF-Simulation zeigt

sich, dass beide Sortierverfahren die Simulation beschleunigen. Lediglich bei kleineren Partikelzahlen ist die Standardversion schneller, da die Kernellaufzeiten hier zu kurz und die Latenzen zu groß sind, um durch die Sortierung beschleunigt zu werden.

Als letzte Simulation werden hier die Ergebnisse für die VPSS-Simulation dargestellt.



(A) Laufzeit der VPSS-Simulation bei unterschiedlichen Sortieroffsets

(B) Laufzeit der VPSS-Simulation bei unterschiedlichen Partikelzahlen

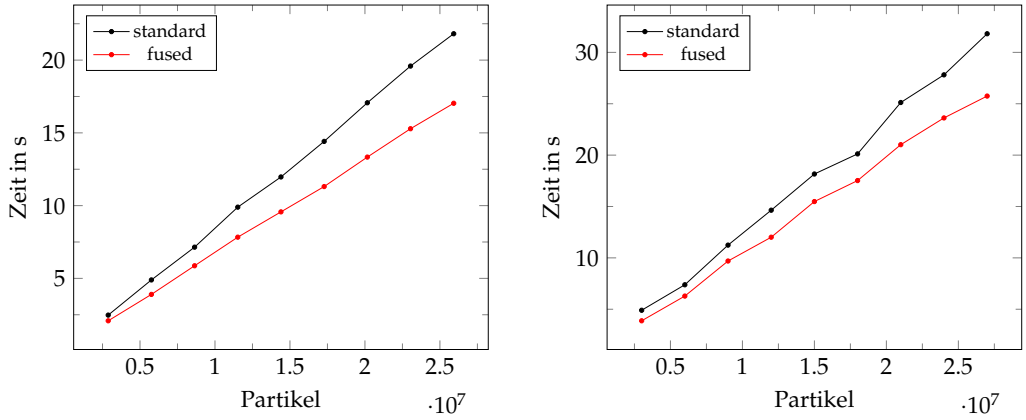
ABBILDUNG 7.5: Sortierung auf GTX 970

In Diagramm 7.5a ist zu sehen, dass sich sowohl die Countingsort-Variante als auch die Radixsort-Variante der VPSS-Simulation der Standardversion annähern und dabei deren Laufzeit nur für wenige Offsets minimal übertreffen. Hieraus ist zu Schließen, dass die Sortierung für beliebige Offsets zu keiner nennenswerten Performancesteigerung führt.

In Abbildung 7.5b ist die Sortierung mit einem Offset von 60 für beide Varianten mit neun unterschiedliche Partikelzahlen dargestellt. Das resultierende Diagramm stimmt mit der Analyse der Sortieroffsets überein und zeigt, dass die Standardversion für einen Offset von 60 die deutlich schnellste Implementierung darstellt.

7.2 Vlasov-Einstein- und Vlasov-Poisson-Simulation mit Kernelfusionierung

In diesem Abschnitt werden die Ergebnisse der Kernelfusionierung dargelegt. Durch die Fusionierung eignet sich die Analyse der Kernellaufzeiten nicht mehr zum Vergleich mit den anderen Varianten, da nur noch fusionierte Kernels ausgeführt werden. Weil bei dieser Optimierung lediglich die Anzahl der Schleifendurchläufe minimiert wird, eignet sich somit die Gesamtlaufzeit als Indikator für die Effizienz der Optimierung. Im nachfolgenden sind in den Abbildungen 7.6 und 7.7 die Laufzeiten der Simulationen mit Kernelfusionierung dargestellt. Es zeigt sich, dass die Einspa-



(A) VEEF-Simulation mit Kernelfusionierung (B) VESC-Simulation mit Kernelfusionierung

ABBILDUNG 7.6

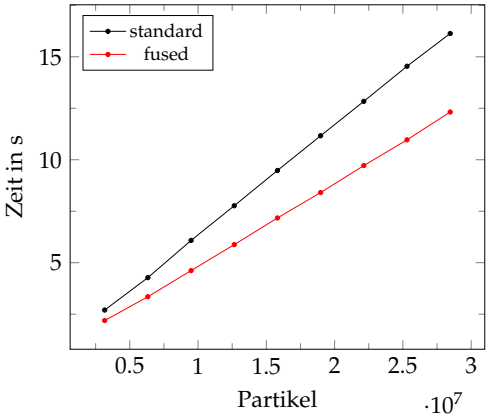


ABBILDUNG 7.7: VPSS-Simulation mit Kernelfusionierung

rung von Iterationen für jede getestete Partikelzahl eine Beschleunigung für die Gesamtlaufzeit der Vlasov-Einstein- und Vlaov-Poisson-Simulationen bedeutet. So ergibt sich für die VEEF-Simulation eine durchschnittliche Beschleunigung von 25,5% und für die VESC-Simulation eine durchschnittliche Beschleunigung von 19,4% im Vergleich zur Standardversion.

Betrachtet man die Ergebnisse für die VPSS-Simulation, so sind diese identisch zu denen der VEEF- und VESC-Simulation. Hier erreicht man durch die Fusionierung von Kernels eine durchschnittliche Beschleunigung von 30,6%. Nach den vorangehenden Messungen beschleunigt die Kernelfusionierung sowohl die Vlasov-Einstein- als auch die Vlasov-Poisson-Simulation.

7.3 Vlasov-Einstein-Simulation mit Kernelfusionierung und Sortierung

Die Kombination von Fusionierung und Sortierung ist nur dann sinnvoll, wenn beide Optimierungen die Laufzeit der Simulation beschleunigt haben. Da bei der VESC- und VPSS-Simulation beide Sortierverfahren die Performance verschlechtern, wird im folgenden nur die VEEF-Simulation behandelt. Für die Laufzeiten ergibt sich das in Abbildung 7.8 dargestellte Diagramm. Es zeigt sich, dass die Kombi-

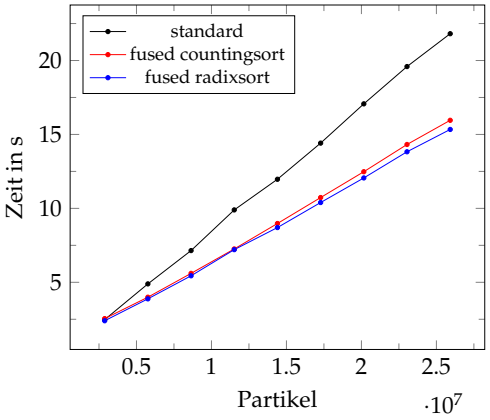


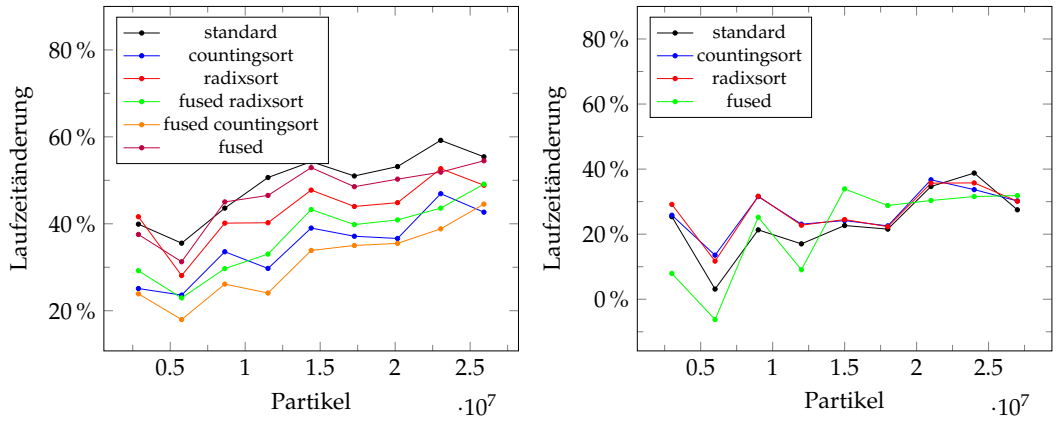
ABBILDUNG 7.8: VEEF-Simulation mit Kernelfusionierung und Sortierung

nation aus Sortierung und Kernelfusionierung bei der VEEF-Simulation eine durchschnittliche Beschleunigung von 29,1% mit Countingsort und 33,3% mit Radixsort erzielt. Der Unterschied zur reinen Kernelfusionierung beträgt im Mittel bei den betrachteten Partikelzahlen 14%. Betrachtet man jedoch die Verteilung der prozentualen Unterschiede, so ist dieser Unterschied bei der Kernelfusionierung über die Partikelzahlen hinweg in etwa konstant, wohingegen bei der Kombination aus Sortierung und Kernelfusionierung der prozentuale Laufzeitunterschied immer größer wird. So erhält man bei der Kombination aus Radixsort und Kernelfusionierung bei 2.880.000 Partikel eine Laufzeitverbesserung von rund 3,2% und für 25.920.000 Partikel eine Performancesteigerung von 42,3%. Für größere Partikelzahlen ist die Kombination aus Radixsort und Kernelfusionierung der reinen Kernelfusionierung vorzuziehen, sofern die Sortierung für die gegebene Simulationskonfiguration eine Laufzeitverbesserung erzielt.

7.4 Analyse der Skalierbarkeit der Vlasov-Einstein-Simulation

Bisher wurden die Messergebnisse für die GTX 970 dargestellt. Um die Skalierbarkeit untersuchen zu können, werden diese Messungen mit denen auf der GTX 980 Ti und Titan V verglichen.

Abbildungen A.1, A.2 und A.3 zeigen die Diagramme zu den Messergebnissen auf der GTX 980 Ti. Es zeigt sich, dass die strukturellen Ergebnisse analog zu denen der GTX 970 sind. Lediglich die Laufzeiten unterscheiden sich auf den beiden GPUs. Da die Simulationen an die Bandbreite gebunden sind, dient ebendiese als Maß für die Skalierbarkeit. Aus Abbildung 3.1 folgt, dass die maximal mögliche Bandbreite 224,4 GB/s auf der GTX 970 und 336,6 GB/s auf der GTX 980 Ti beträgt, sodass die Bandbreite auf der GTX 980 Ti um 50% schneller ist. Die folgenden Diagramme in Abbildung 7.9 stellen die prozentualen Änderungen der Laufzeit auf den beiden GPUs dar. Eine positive Änderung bedeutet hier eine Beschleunigung der Laufzeit.



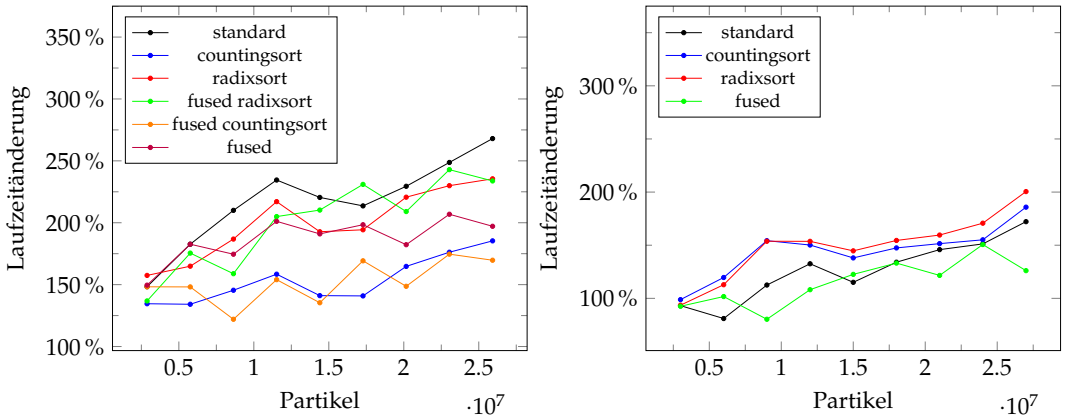
(A) Prozentuale Laufzeitänderung der VEEF-Simulation zwischen GTX 970 und 980 Ti

(B) Prozentuale Laufzeitänderung der VESC-Simulation zwischen GTX 970 und 980 Ti

ABBILDUNG 7.9

Es zeigt sich, dass für größere Partikelzahlen alle VEEF-Implementierungen gegen die genannten 50% Beschleunigung streben, wobei die Kernelfusionierung identisch zur Standardversion skaliert, gefolgt von den Radixsort- und Countingsort-Varianten. Bei der VESC-Simulation skalieren alle Versionen fast identisch, besonders bei größeren Partikelzahlen. Im Vergleich zur VEEF-Simulation ergibt sich bei der VESC-Simulation eine Beschleunigung zwischen 30% und 40% bei größeren Partikelzahlen bei einer um 50% schnelleren Bandbreite. Damit skaliert die VESC-Simulation deutlich schlechter als die VEEF-Simulation. Sowohl die GTX 970 als auch die GTX 980 Ti gehören zur Maxwell-Architektur von Nvidia. Im Folgenden wird die Volta-Architektur mit der Titan V verwendet, um das Verhalten der Varianten auf einer anderen Architektur zu analysieren. Das Vorgehen ist dasselbe wie bei der vorangehenden Skalierbarkeitsanalyse. Die Bandbreite beträgt auf der GTX 970

224,4 GB/s und auf der Titan V 652,8 GB/s. Ausgehend von der GTX 970 ist dies eine Bandbreitenerhöhung von 194%. Die Diagramme in Abbildung 7.10 zeigen die prozentuale Änderung der Laufzeit auf der GTX 970 und Titan V und basieren auf den Diagrammen A.5, A.6 und A.7. Zuerst betrachtet wird die VEEF-Simulation.



(A) Prozentuale Laufzeitänderung der VEEF-Simulation zwischen GTX 970 und Titan V

(B) Prozentuale Laufzeitänderung der VESC-Simulation zwischen GTX 970 und Titan V

ABBILDUNG 7.10

Auch hier skaliert die Standardversion am besten, gefolgt von den Radixsort-Implementierungen, welche nahezu identisch zur Standardversion skalieren, der Kernelfusionierung und der Countingsort-Varianten. Im Vergleich zur vorangehenden Skalierbarkeitssanalyse auf der GTX 980 Ti liegen die Skalierungen der einzelnen Implementierungen weiter auseinander. Außerdem skalieren die Radixsort-Varianten besser als die Version mit Kernelfusionierung. Bei der VESC-Simulation skalieren die beiden Varianten mit Sortierung besser und die Kernelfusionierung schlechter als die Standardversion. Im Gegensatz zur Skalierbarkeitsanalyse zwischen GTX 970 und GTX 980 Ti skalieren die Implementierungen mit Sortierung besser als die Variante mit Kernelfusionierung. Ein weiterer, auffallender Unterschied in Bezug auf alle Vlasov-Einstein-Simulationen und Versionen ist die prozentuale Laufzeitverbesserung. So erreichen die Standardversion, Radixsort-Varianten und die Version mit Kernelfusionierung der VEEF-Simulation eine Beschleunigung von 225% bis 250% im Vergleich zur GTX 970 bei einer um 194% schnelleren Bandbreite. Dies bedeutet, dass die Varianten besser als nur mit der Bandbreite skalieren. Bei der VESC-Simulation ist dies ebenfalls zu beobachten. So werden bei größeren Partikelzahlen Skalierungen um die 200% erreicht, obwohl die Skalierbarkeit bei der Analyse zwischen GTX 970 und GTX 980 Ti immer schlechter als die berechnete Bandbreitenerhöhung war. Um herauszufinden, was diese zusätzliche Laufzeitverbesserung verursacht, werden die erreichten Bandbreiten auf der GTX 970, GTX 980 Ti und Titan V miteinander verglichen.

Ein wichtiger Aspekt bei der Bandbreitenmessung ist die Abnahme der Bandbreite bei zunehmender Unsortiertheit im Verlauf der Simulation. Es ist also sinnvoll,

bei der unsortierten und der kernelfusionierten Variante den ersten Kernel zu messen, da die Partikel hier perfekt sortiert sind. Bei den Implementierungen mit Sortierung muss immer genau nach der Sortierung gemessen werden, um verlässliche und vergleichbare Messergebnisse zu erhalten, da diese dann nicht von den unterschiedlichen Sortieroffsets beeinflusst werden. Betrachtet man nämlich unterschiedliche Kernels in den unterschiedlichen Simulationsvarianten, so sind diese nicht vergleichbar, da die Bandbreite durch den Verlauf der Simulation beeinflusst wird. Die Ergebnisse dieser Messungen für die VEEF-Simulation sind in Tabelle 7.1 dargestellt. Die unsortierte und kernelfusionierte Variante werden hier nicht mit dargestellt, liefern aber analoge Ergebnisse. Betrachtet man die prozentuale Auslastung der Band-

GPU/Variante:	countingsort:	radixsort:	fused counting	fused radix:
GTX 970	60.61%	58.55%	27.42%	27.16%
GTX 980 Ti	64.30%	60.25%	26.55%	24.60%
Titan V	83.88%	85.36%	41.86%	36.27%

TABELLE 7.1: Bandbreitenauslastung von propagate_particles für die VEEF-Simulation

breite für propagate_particles auf den unterschiedlichen GPUs, so sind die Werte der Maxwell-GPUs nahezu identisch. Für die Titan V und die Volta-Architektur erreicht man im Durchschnitt eine um ca. 20% höhere Bandbreitenauslastung als für die Maxwell-Architektur. Da auf beiden Architekturen derselbe Code verwendet wird, ist diese Erhöhung der Bandbreitennutzung auf die Verwendung der Volta-Architektur zurückzuführen. Im Gegensatz hierzu steht die prozentuale Bandbrei-

GPU/Variante:	countingsort:	radixsort:
GTX 970	9.82%	9.78%
GTX 980 Ti	8.86%	11.31%
Titan V	13.25%	12.65%

TABELLE 7.2: Bandbreitenauslastung von comp_fields für die VEEF-Simulation

tennutzung für comp_fields in Tabelle 7.2. Hier fallen die architekturspezifischen Unterschiede deutlich geringer aus. Es folgt also, dass die Architektur einen direkten Einfluss auf die Laufzeit von propagate_particles hat und erklärt die über die 194% hinausgehende Beschleunigung der Laufzeit.

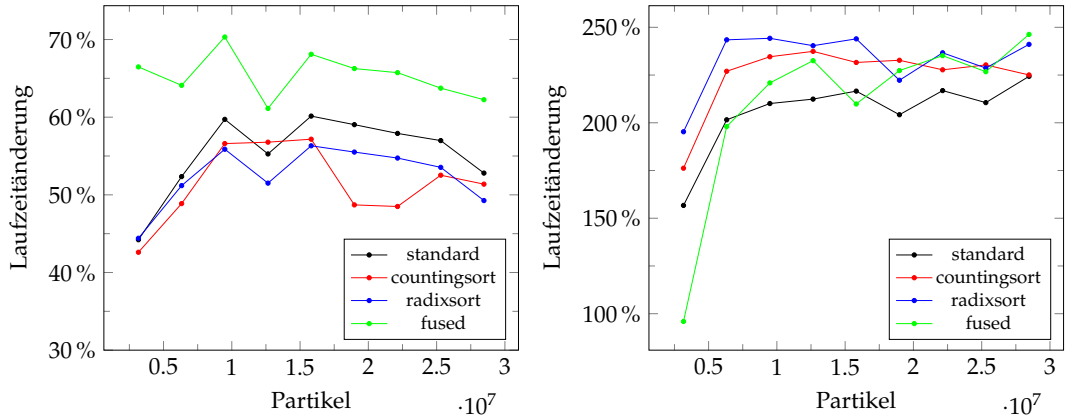
Für die VESC-Simulation ergeben sich ähnliche Ergebnisse in Bezug auf die architekturspezifischen Unterschiede zwischen Maxwell und Volta. Tabelle 7.3 stellt die Bandbreitennutzung für propagate_particles dar. Die resultierende Auslastung der Bandbreite liegt für die Maxwell-GPUs bei ca. 60%, wohingegen die Titan V ca. 80% erreicht. Es ergibt sich also auch hier eine architekturspezifisch höhere Bandbreite von 20%. Da die Ergebnisse von comp_fields nahezu identisch zu denen der VEEF-Simulation sind, werden diese hier nicht aufgeführt.

GPU/Variante:	countingsort:	radixsort:
GTX 970	58.30%	58.31%
GTX 980 Ti	61.52%	61.84%
Titan V	79.46%	79.22%

TABELLE 7.3: Bandbreitenauslastung von propagate_particles für die VESC-Simulation

7.5 Analyse der Skalierbarkeit der Vlasov-Poisson-Simulation

Abbildung 7.11 zeigt die Skalierbarkeit der einzelnen Varianten für die VPSS-Simulation zwischen GTX 970 und GTX 980 Ti und der Titan V. Diese basieren auf den Messungen in Abbildung A.8 und Abbildung A.4.



(A) Prozentuale Laufzeitänderung zwischen GTX 970 und GTX 980 Ti

(B) Prozentuale Laufzeitänderung der VPSS-Simulation zwischen GTX 970 und Titan V

ABBILDUNG 7.11

Es zeigt sich, dass die kernelfusionierte Variante auf der GTX 980 Ti am besten skaliert, wohingegen die unsortierte und die beiden sortierten Implementierungen in etwa gleich skalieren. Man erreicht bei einer um 50% schnelleren Bandbreite eine Skalierbarkeit zwischen 50% und 65%, wobei die kernelfusionierte Variante bei größeren Partikelzahlen gegen 60% strebt und die anderen Versionen sich den 50% annähern. Auf der Titan V skalieren die Countingsort- und Radixsort-Varianten wieder nahezu identisch, liegen hier aber deutlich über der Standardversion. Lediglich für größere Partikelzahlen liegt die Skalierung der Radixsort-Implementierung über der Skalierung der Countingsort-Variante. Die kernelfusionierte Variante skaliert ebenfalls besser als die Standardversion, skaliert für kleinere Partikelzahlen aber schlechter als die beiden sortierten Varianten und nähert sich erst bei größeren Partikelzahlen der Skalierung der Radixsort-Version an. Bei einer um 194% schnelleren Bandbreite wird hier eine Skalierbarkeit zwischen 220% und 250% erreicht. Die Ergebnisse in Bezug auf die von der GPU abhängigen prozentualen Laufzeitänderungen

sind demnach analog zu denen der Vlasov-Einstein-Simulationen. Um dies weiter zu belegen, werden auch hier die prozentualen Bandbreitenauslastungen analysiert. Die Ergebnisse sind in Tabelle 7.4 und Tabelle 7.5 dargestellt.

GPU/Variante:	countingsort:	radixsort:
GTX 970	62.25%	63.57%
GTX 980 Ti	64.30%	60.25%
Titan V	83.11%	83.10%

TABELLE 7.4: Bandbreitenauslastung von propagate_particles für die VPSS-Simulation

GPU/Variante:	countingsort:	radixsort:
GTX 970	10.39%	9.28%
GTX 980 Ti	9.76%	9.27%
Titan V	13.72%	12.90%

TABELLE 7.5: Bandbreitenauslastung von comp_fields für die VPSS-Simulation

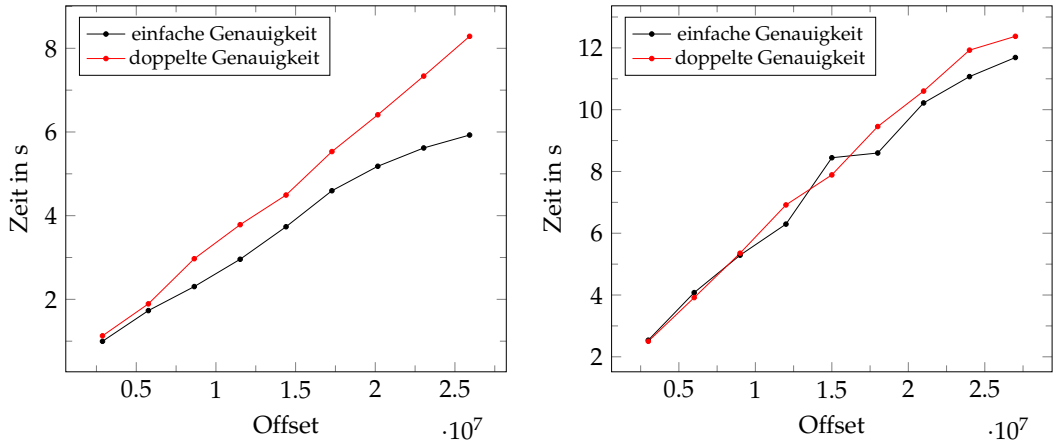
Auch hier wird durch die Verwendung der Volta-Architektur mit der Titan V eine um ca. 20% höhere Bandbreitenauslastung bei propagate_particles erreicht, wohingegen bei comp_fields eine um 3% bis 4% höhere Bandbreitenauslastung erreicht wird. Die über die 194% hinausgehende Skalierbarkeit ist also auch hier auf die Volta-Architektur zurückzuführen.

8. Einfluss der Fließkomma-Genauigkeit auf die Simulationen

In diesem Kapitel wird die Veränderung der Laufzeit und Messgenauigkeit bei Verwendung von Fließkommazahlen mit einfacher und doppelter Genauigkeit untersucht. Hintergrund hierfür ist der große Performanceunterschied zwischen Berechnungen mit einfacher und doppelter Genauigkeit. Bei an die Rechenleistung gebundenen Kernels können bei einfacher Genauigkeit bis zu 32 Mal mehr FLOPS ausgeführt werden als bei doppelter Genauigkeit, je nach verwendeter GPU. Für diese Messungen wird die Nvidia Titan V verwendet, welche bei einfacher Genauigkeit maximal 14.899 GFLOPS und bei doppelter Genauigkeit 7.450 GFLOPS erreichen kann. Im nachfolgenden wird analysiert, ob der Unterschied zwischen der Messgenauigkeit eine längere Ausführungszeit durch Verwendung von doppelter Genauigkeit rechtfertigt.

8.1 Performanceunterschied bei Verwendung von einfacher und doppelter Genauigkeit

Die Diagramme 8.1a und 8.1b zeigen die Laufzeiten der nichtsortierten Variante für die VEEF- und VESC-Simulation bei der Verwendung von Fließkommazahlen mit einfacher und doppelter Genauigkeit. Es zeigt sich, dass bei der VEEF-Variante die Laufzeitunterschiede mit steigender Partikelzahl größer werden, wohingegen bei der VESC-Simulation die Unterschiede zwischen einfacher und doppelter Genauigkeit deutlich geringer ausfallen. Dies liegt daran, dass die Genauigkeit der Fließkommazahlen hauptsächlich die Berechnung der Partikelbewegung beeinflusst und damit den mit der Partikelbewegung verbundenen prozentualen Anteil der Laufzeit. Abbildung 5.1 zeigt, dass bei der VEEF-Simulation 49,08% und bei der VESC-Simulation 25,55% der gesamten Laufzeit für die Partikelbewegung benötigt werden. Aus diesem Grund ist der Unterschied zwischen einfacher und doppelter Genauigkeit bei der VEEF-Simulation größer als bei der VESC-Simulation.



(A) Standardvariante der VEEF-Simulation auf der Titan V

(B) Standardvariante der VESC-Simulation auf der Titan V

ABBILDUNG 8.1

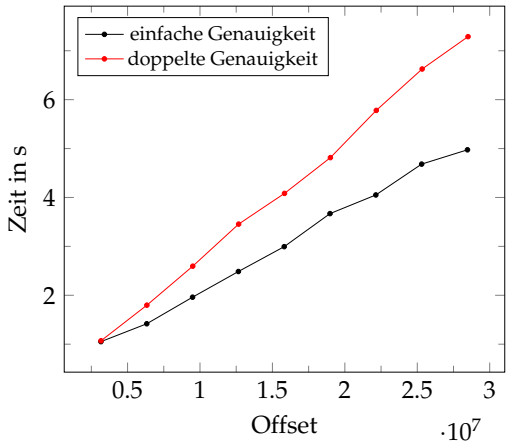


ABBILDUNG 8.2: Standardvariante der VPSS-Simulation auf der Titan V

Das Diagramm 8.2 zeigt die Laufzeiten der nichtsortierten Variante für die VPSS-Simulation bei einfacher und doppelter Genauigkeit der Fließkommazahlen. Bei der VPSS-Simulation beträgt der prozentuale Anteil der Laufzeit für die Partikelbewegung 43,61% und ist damit dem der VEEF-Simulation sehr ähnlich. Vergleicht man die Diagramme der beiden Simulationen, so werden auch hier die Laufzeitunterschiede mit steigender Partikelzahl größer.

8.2 Unterschiede der Messgenauigkeit bei der Verwendung von einfacher und doppelter Genauigkeit

Um die Messgenauigkeit zu analysieren, werden die mittleren prozentualen Abweichungen für die vier Partikelkomponenten *r, w, f und ang2* bei einfacher und doppelter Genauigkeit untersucht. Betrachtet wird die VEEF-Simulation, da hier die Partikelzahl bei einfacher und doppelter Genauigkeit übereinstimmt und man so die einzelnen Partikel miteinander vergleichen kann. Durchgeführt wird diese Analyse mit 28.8 Millionen Partikel, welche über 20000 Zeitschritte auf dem Gitter bewegt werden. Tabelle 8.1 zeigt eine Übersicht über die gemessenen Abweichungen von den Ergebnissen mit einfacher Genauigkeit auf der GPU zu den Ergebnissen der MPI-Version mit doppelter Genauigkeit. Es zeigt sich, dass der maximal gemessene

	mittlerer Fehler	maximaler Fehler	minimaler Fehler
r	$2.833 \times 10^{-2}\%$	$4.342 \times 10^{-1}\%$	$3.937 \times 10^{-7}\%$
w	$4.755 \times 10^{-2}\%$	2.111%	$4.629 \times 10^{-9}\%$
f	$7.815 \times 10^{-4}\%$	$1.131 \times 10^{-1}\%$	$7.818 \times 10^{-10}\%$
ang2	$6.966 \times 10^{-2}\%$	$8.692 \times 10^{-1}\%$	$6.459 \times 10^{-7}\%$

TABELLE 8.1: Abweichungen der Partikeldaten zwischen einfacher und doppelter Genauigkeit nach 20000 Zeitschritten

Fehler in etwa 2,111% beträgt. Dies ist ein Indiz dafür, dass die Verwendung von einfacher Genauigkeit bei dieser Simulation keinen Einfluss auf die Messergebnisse hat, da die 2,1% hier als Maß der Ähnlichkeit zu den Ergebnissen mit doppelter Genauigkeiten angesehen werden können. Dies liegt daran, dass die Ergebnisse mit doppelter Genauigkeit ebenfalls auf einer numerischen Simulation basieren und so auch diese mit Fehler behaftet sind, welche unter Umständen größer als 2,1% sein können. Betrachtet man die mittlere Abweichung, so ist diese im Bereich von 10^{-2} bis 10^{-4} und so klein, dass man auch hier davon ausgehen kann, dass die Erhöhung der Laufzeit durch Verwendung von Fließkommazahlen mit einfacher Genauigkeit sinnvoll ist.

Um zu zeigen, dass die Messergebnisse bei doppelter Genauigkeit auf CPU und GPU nahezu identisch sind, wird Tabelle 8.2 verwendet. Hier beträgt der maximal

	mittlerer Fehler	maximaler Fehler	minimaler Fehler
r	$5.589 \times 10^{-5}\%$	$5.061 \times 10^{-4}\%$	$8.245 \times 10^{-11}\%$
w	$9.296 \times 10^{-5}\%$	$8.797 \times 10^{-4}\%$	$9.362 \times 10^{-11}\%$
f	$9.410 \times 10^{-5}\%$	$4.991 \times 10^{-4}\%$	$7.818 \times 10^{-10}\%$
ang2	$1.203 \times 10^{-4}\%$	$7.943 \times 10^{-4}\%$	$1.444 \times 10^{-10}\%$

TABELLE 8.2: Abweichungen der Partikeldaten zwischen einfacher und doppelter Genauigkeit nach 20000 Zeitschritten

gemessene Fehler 8.797×10^{-4}%. Er ist also um vier Größenordnungen kleiner als im vorangehenden Absatz.

Im Vergleich zu den mittleren Abweichungen von einfacher zu doppelter Genauigkeit liegen die Abweichungen von doppelter Genauigkeit auf der Titan V zu denen auf der CPU im Bereich von 10^{-4} bis 10^{-5}, was einen Unterschied von bis zu drei Größenordnungen ausmacht.

Zusammenfassend ist zu sagen, dass die gemessenen Fehler im Mittel kleiner als 1% sind und numerische Simulationen grundsätzlich fehlerbehaftet sind. Dies bedeutet, dass durch doppelte Genauigkeit die Abweichung zu den Referenzwerten um bis zu drei Größenordnungen verringert werden kann, für diese Werte allerdings nicht bekannt ist, wie fehlerbehaftet bzw. genau sie sind. Da die Performanceunterschiede zwischen einfacher und doppelter Genauigkeit nicht vernachlässigbar klein sind, ist es sinnvoll, hier einfache statt doppelter Genauigkeit zu verwenden.

9. Fazit

Der erste Schritt in dieser Arbeit war die Implementierung einer Standardversion, welche bereits Standardoptmierungen wie das Einhalten von *Global Memory Coalescing*, die Minimierung von Speicheroperationen und die Vermeidung von divergentem Kontrollfluss enthielt. Durch die Analyse von propagate_particles ergab sich die Limitierung der Performance durch die Speicherbandbreite unter idealen Bedingungen und damit die Möglichkeit der Optimierung der Performance durch Partikelsortierung, da so die Partikellokalität und damit das koaleszente Speicherzugriffsmuster verbessert werden können. Es zeigte sich auch, dass die Partikelsortierung die Performance der Feldberechnungen aufgrund der vorhandenen atomaren Additionen verschlechtert, wodurch der Ansatz der Sortierung mit unterschiedlichen Offsets analysiert wurde. Hier zeigte sich, dass lediglich bei der VEEF-Simulation eine Performanceverbesserung durch Sortierung erreicht wurde, da hier das Verhältnis von Speicherzugriffen auf Felddaten in propagate_particles zu den atomaren Speicherzugriffen in den Feldberechnungen ausreichend groß war. Mit der Kernelfusionierung wurde ein weiterer Ansatz zur Performanceoptimierung untersucht. Hier ergab sich für alle Simulationen eine Performanceverbesserung, was durch die Minimierung von Schleifendurchläufen zu erwarten war. Schlussendlich war die Kernelfusionierung bei der VPSS- und VESC-Simulation die schnellste Variante, wohingegen die Kernelfusionierung mit Radixsort die schnellste Variante für die VEEF-Simulation darstellte. Hiernach wurde die Skalierbarkeit ausgehend von den Ergebnissen auf der GTX 970 für die Maxwell-Architektur mit der GTX 980 Ti und der Volta-Architektur mit der Titan V analysiert. Es zeigte sich, dass die Varianten auf der Maxwell-Architektur im Schnitt mit der Bandbreite skalierten. Auf der Volta-Architektur skalierten die Varianten deutlich besser, obwohl die verwendeten GPU-Codes auf beiden Architekturen identisch waren. Um den Hintergrund hierfür zu identifizieren, wurden die prozentual erreichten Bandbreiten gemessen und miteinander verglichen. Es zeigte sich eine um ca. 20% höhere Bandbreitennutzung auf der Volta-Architektur im Vergleich zur Maxwell-Architektur, sodass die höhere Skalierbarkeit auf die architekturspezifische Bandbreite zurückzuführen ist. Im Anschluss wurde die numerische Stabilität bei einfacher und doppelter Genauigkeit überprüft. Hier zeigte sich, dass die Fehler zwischen einfacher und doppelter Genauigkeit bei 20000 Zeitschritten so gering waren, dass die Verwendung von Fließkommazahlen mit einfacher Genauigkeit als numerisch stabil angesehen werden kann.

10. Ausblick

In diesem Abschnitt werden weiterführende Ansätze zur Optimierung der Particle-In-Cell-Verfahren vorgestellt. Diese umfassen die Implementierung einer adaptiven Schrittweite der Gitterpunkte und eine Anpassung der Feldberechnungen für nicht uniforme Schrittweiten der Gitterpunkte.

10.1 Adaptive Schrittweite der Gitterpunkte

Bisher war der Radius zwischen zwei Gitterpunkten über die Simulation hinweg konstant. Dies bedeutet, dass bei einer Konzentration von vielen Partikeln in wenigen Gitterzellen die Auflösung der Simulation schlechter wird. Um dafür zu sorgen, dass die Zahl der Gitterzellen auch die Auflösung der Simulation während der Ausführung repräsentiert, kann die bisherige Schrittweite so verändert werden, dass zum einen die Zahl der Gitterpunkte gleichbleibt, zum anderen aber auch in jeder Gitterzelle ein Partikel enthalten ist.

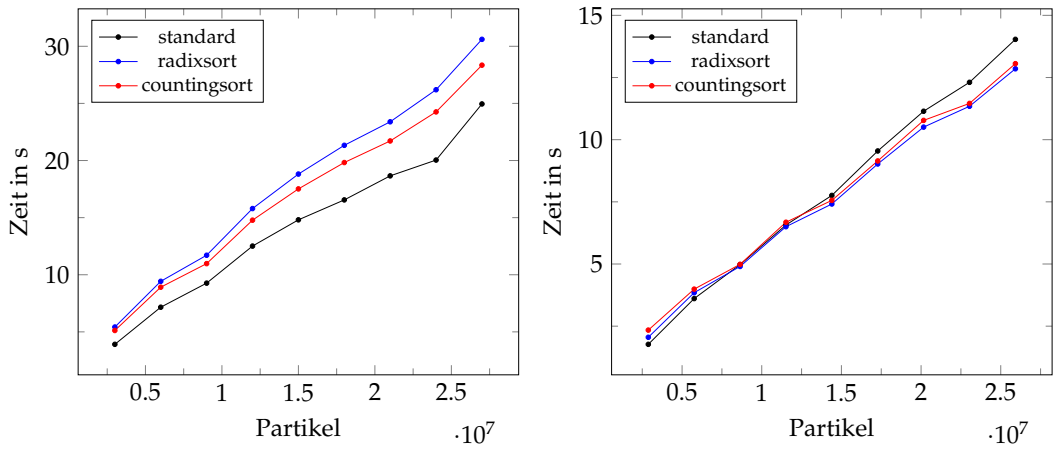
10.2 Anpassung der Feldberechnungen für nicht uniforme Schrittweiten

Eine weitere Anpassung in Bezug auf die Schrittweite bzw. den Abstand zweier Gitterzellen besteht in der Verwendung von nicht uniformen Schrittweiten. Auf diese Art und Weise ist es möglich, die Schrittweite so zu wählen, dass in jeder Zelle die gleiche Anzahl an Partikeln enthalten ist. Mithilfe dieser Änderung der Schrittweite kann man die Berechnung der Felder so umformen, dass bei perfekter Sortierung die atomaren Operationen überflüssig werden. Dies ist auf die gleichverteilte Zahl an Partikeln zurückzuführen, da so die Feldberechnungen für eine Zelle unabhängig durchgeführt werden können.

A. Diagramme

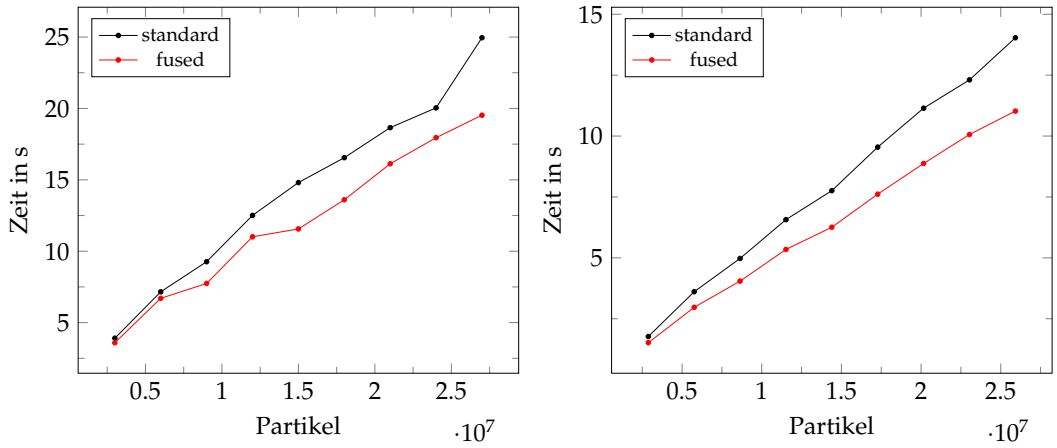
In diesem Anhang werden alle Diagramme dargestellt, welche nur indirekt im Text verwendet werden.

A.1 Messungen GTX 980 Ti



(A) VESC-Simulation mit Sortierung (B) VEEF-Simulation mit Sortierung

ABBILDUNG A.1



(A) VESC-Simulation mit Kernelfusionierung (B) VEEF-Simulation mit Kernelfusionierung

ABBILDUNG A.2

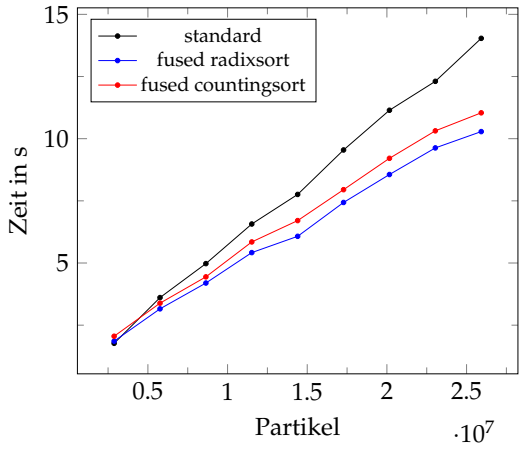
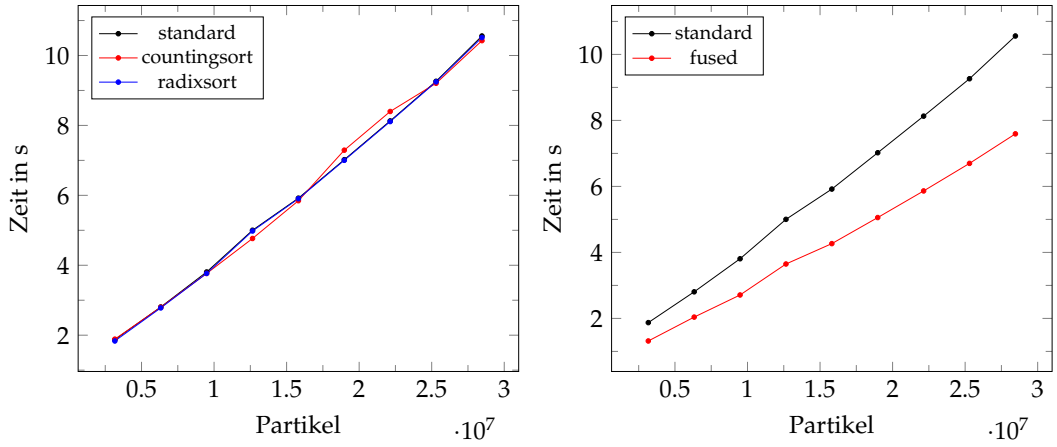


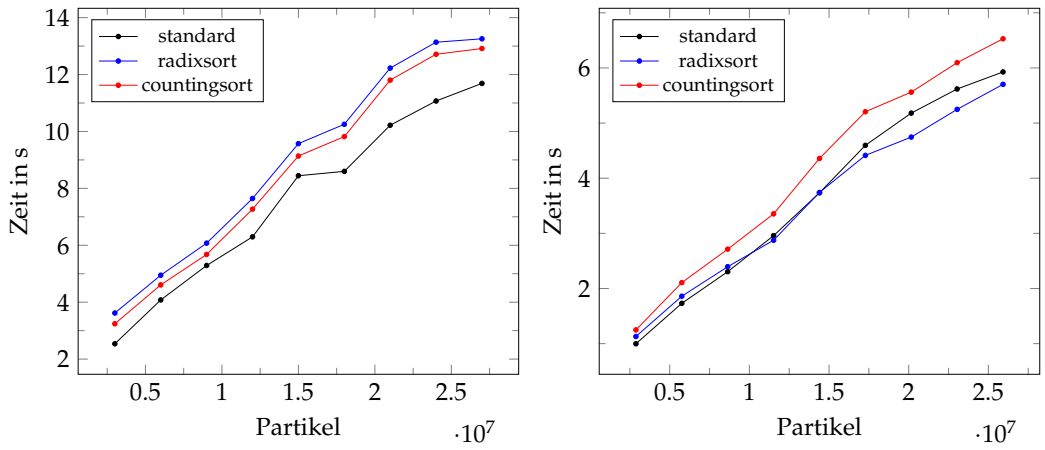
ABBILDUNG A.3: VEEF-Simulation mit Sortierung und Kernelfusionierung



(A) VPSS-Simulation mit Sortierung (B) VPSS-Simulation mit Kernelfusionierung

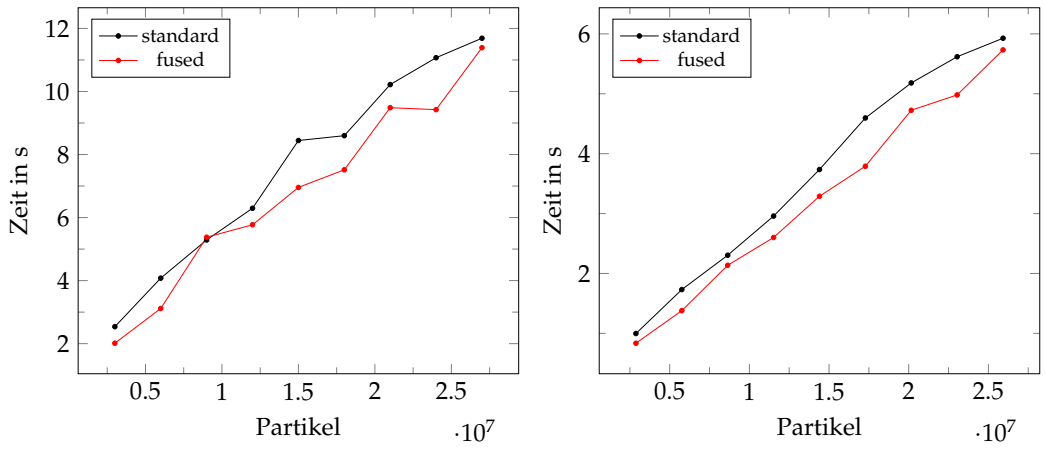
ABBILDUNG A.4

A.2 Messungen TITAN V



(A) VESC-Simulation mit Sortierung (B) VEEF-Simulation mit Sortierung

ABBILDUNG A.5



(A) VESC-Simulation mit Kernelfusionierung (B) VEEF-Simulation mit Kernelfusionierung

ABBILDUNG A.6

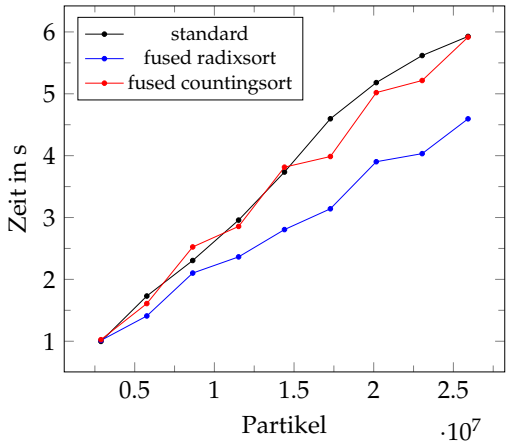
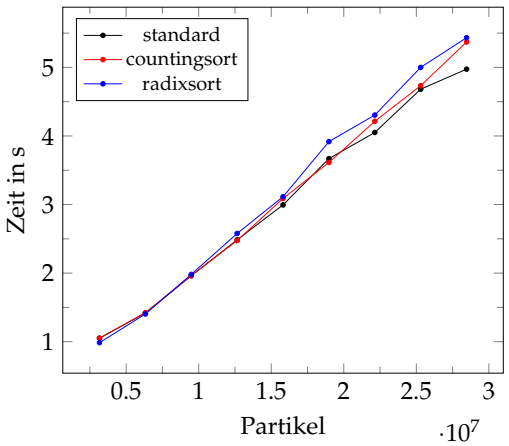
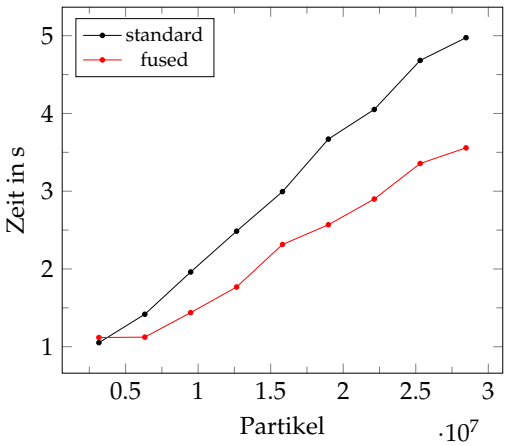


ABBILDUNG A.7: VEEF-Simulation mit Sortierung und Kernelfusionierung



(A) VPSS-Simulation mit Sortierung



(B) VPSS-Simulation mit Kernelfusionierung

ABBILDUNG A.8

Literatur

[1] Heiko Burau u. a. „PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster“. In: *IEEE Transactions on Plasma Science* 38.10 (Okt. 2010), S. 2831–2838. URL: https://ieeexplore.ieee.org/abstract/document/5556015.

[2] *GPU Specs Database*. https://www.techpowerup.com/gpu-specs. Zugriff: 09.01.2019.

[3] Jared Hoberock. *Cached Allocator for Thrust*. https://github.com/thrust/thrust/blob/master/examples/cuda/custom_temporary_allocation.cu. Zugriff: 05.01.2019.

[4] Jared Hoberock und Nathan Bell. *Thrust*. https://thrust.github.io. Zugriff: 05.01.2019.

[5] Duane Merrill und NVIDIA CORPORATION. *CUB*. https://nvlabs.github.io/cub. Zugriff: 05.01.2019.

[6] NVIDIA CORPORATION. *Cuda C - Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Zugriff: 05.01.2019.

[7] Gerhard Rein. *Numerische Simulation für das Vlasov-Poisson- und das Einstein-Vlasov-System*. Okt. 2010.

[8] H. Shah u. a. „A Novel Implementation of 2D3V Particle-in-Cell (PIC) Algorithm for Kepler GPU Architecture“. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Dez. 2017, S. 378–387.

[9] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[10] Oliver Zier. *Weiterentwicklung eines parallelen Particle-in-Cell-Codes für nichtlineare Modelle der mathematischen Physik*. März 2017.

Eidesstattliche Erklärung

Ich, Philipp Raithel, erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich versichere außerdem, dass diese Arbeit bisher noch nicht zur Erlangung eines akademischen Grades eingereicht wurde.

Unterschrift:

_____

Datum:

_____