DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Studiengang Informationstechnik

Bachelorarbeit

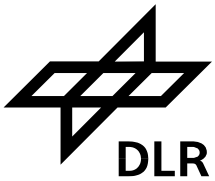
Bearbeitungszeitraum: 23. Juni 2014 - 19. September 2014

Parallele Python-Programmierung auf Multi-Core-Architekturen und Grafikkarten für numerische Algorithmen aus der Strömungstechnik und den Materialwissenschaften

von **Joachim Illmer**

- Matrikelnr.: 5285595 -

- Kurs: TINF11ITIN -



Deutsches Zentrum für
Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Standort: Köln

Einrichtung für Simulations- und Softwaretechnik
Abteilung: Verteilte Systeme und Komponentensoftware

Betreuer:
Dr.-Ing. Achim Basermann
Melven Röhrig-Zöllner (M. Sc.)
Prof. Dr. Harald Kornmayer

Zusammenfassung

Die Programmiersprache Python wird heutzutage vielfach verwendet, um schnell kompakte Prototypen zu entwickeln. Für HPC-Anwendungen wird sie aufgrund der geringeren Performanz im Vergleich zu Hardware-nahen Programmiersprachen wie C oder Fortran nicht eingesetzt. Eine effiziente parallele Programmierung auf Multi-Core-Architekturen und Grafikkarten ist generell als schwierig anzusehen. Hier soll getestet werden, ob Python einen einfachen Zugang zu diesen Systemen bietet. In dieser Bachelorarbeit soll daher die Performanz von Python-Implementierungen mit unterschiedlichen Zusatzbibliotheken untersucht und mit Implementierungen in C oder Fortran verglichen werden. Dazu wurde ein Ausschnitt eines Hubschrauber-Simulationscodes aus der Strömungstechnik ausgewählt. Dieser Simulationscode lag in Fortran für die Ausführung auf Single-Core-, Multi-Core- und Grafikkarten-Systemen vor. In Python wurden mit unterschiedlichen Zusatzbibliotheken kompakte Versionen für diese Systeme implementiert. Der Simulationscode wurde mit einem Performanz-Modell analysiert, um eine Abschätzung der zu erwartenden Performanz zu erhalten. Die Performanz der verschiedenen Implementierungen wurde verglichen und mit Hilfe von Tools analysiert. Dabei zeigte sich, dass eine Implementierung mit Python-Syntax auf Single-Core-Systemen ungefähr ein Sechstel der Performanz von Fortran erreicht. Die Performanz auf Multi-Core-Systemen und Grafikkarten beträgt ungefähr ein Zehntel der Performanz der Fortran-Implementierung. Eine höhere Performanz erreicht eine hybride Implementierung aus C und Python. Diese erzielte sowohl auf Single-Core- als auch auf Multi-Core-Systemen ungefähr die Hälfte der Performanz von Fortran.

Joachim Illmer, September 2014

Abstract

The programming language Python is widely used to create fast compact software prototypes. However, compared to low-level programming languages like C or Fortran low performance is preventing its use for HPC applications. Efficient parallel programming on multi-core systems and graphic cards is generally a complex task. Python with add-ons might provide a simple approach to program those systems. This bachelor thesis evaluates the performance of Python implementations with different libraries and compares it to implementations in C or Fortran. As a test case from the field of fluid dynamics a part of a rotor simulation code was selected. Fortran versions of this code were available for use on single-core, multi-core and graphic-card systems. For all these computer systems, multiple compact versions of the code were implemented in Python with different libraries. For performance analysis of the rotor simulation kernel, a performance model was developed. This model was then used to assess the performance reached with the different implementations. In order to obtain detailed performance information on the codes versions developed, performance tools were exploited in addition. This showed that an implementation with Python syntax is six times slower than Fortran on single-core systems. The performance on multi-core systems and graphic cards is about a tenth of the Fortran implementations. A higher performance was achieved by a hybrid implementation in C and Python. The latter reached about half of the performance of Fortran.

Joachim Illmer, September 2014

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Studiengang Informationstechnik

Bachelorarbeit

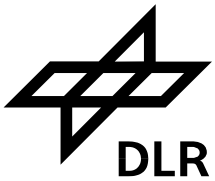
Bearbeitungszeitraum: 23. Juni 2014 - 19. September 2014

Parallele Python-Programmierung auf Multi-Core-Architekturen und Grafikkarten für numerische Algorithmen aus der Strömungstechnik und den Materialwissenschaften

von **Joachim Illmer**

- Matrikelnr.: 5285595 -

- Kurs: TINF11ITIN -



Deutsches Zentrum für
Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Standort: Köln

Einrichtung für Simulations- und Softwaretechnik
Abteilung: Verteilte Systeme und Komponentensoftware

Betreuer:
Dr.-Ing. Achim Basermann
Melven Röhrig-Zöllner (M. Sc.)
Prof. Dr. Harald Kornmayer

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich
die vorliegende Arbeit selbstständig und nur unter
Verwendung der angegebenen Quellen und
Hilfsmittel angefertigt habe.

Köln, der 17. September 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

API	**A**pplication **P**rogramming **I**nterface
ccNUMA	**c**ache-**c**oherent **N**on**u**niform **M**emory **A**ccess
CPU	**C**entral **P**rocessing **U**nit
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
DP	**D**ouble **P**recision
FLOPS	**Fl**oating Point **O**perations **p**er **S**econd
GIL	**G**lobal **I**nterpreter **L**ock
GPGPU	**G**eneral-**P**urpose Computing on **G**raphics **P**rocessing **U**nits
GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputing
JIT	**J**ust **I**n **T**ime
MKL	**M**ath **K**ernel **L**ibrary
MPI	**M**essage **P**assing **I**nterface
OpenMP	**Open M**ulti **P**rocessing
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SMX	**S**treaming **M**ultiprocessor

SP **S**ingle **P**recision

UMA **U**niform **M**emory **A**ccess

1 Einleitung

1.1 Motivation der Arbeit

Die Programmiersprache Python wird heutzutage vielfach verwendet. Sie erlaubt eine einfache Programmierung von Prototypen, ähnlich wie die interaktive Plattform MATLAB von MathWorks.

In der Liste der 500 schnellsten Computer der Welt von Juni 2014 ist zu erkennen, dass mittlerweile 62 dieser Systeme Grafik-Co-Prozessoren nutzen [Top14]. Da Multi-Core-Architekturen und „**G**eneral-**P**urpose Computing on **G**raphics **P**rocessing **U**nits“(GPGPU) mit vielen Prozessorelementen außerdem heute für jedermann zugänglich sind, wird die Entwicklung hochperformanter paralleler Programme für diese Computing-Systeme zunehmend wichtiger. Effiziente parallele Programmierung auf Multi-Core-Architekturen und GPGPUs ist generell als schwierig anzusehen. Bei der Entwicklung paralleler Programme muss beachtet werden, welcher Teil parallelisiert werden kann, wie groß jede parallele Task sein soll, wie der Transport von Daten minimal gehalten werden kann, wie eine Synchronisation der Daten sichergestellt wird und wie die Performanz modelliert und überprüft werden kann.

Unter anderem in der Strömungstechnik und den Materialwissenschaften ersetzen Computer-Simulationen mehr und mehr Experimente. Simulationen bieten die Möglichkeit, Theorien zu überprüfen, die nicht durch ein Experiment getestet werden können, weil sie zu teuer, zu schwer, zu langsam oder zu gefährlich sind oder es nicht möglich ist,

die zu beobachtende Größe zu messen. Simulationscodes müssen moderne Computer-Hardware möglichst effizient nutzen, um kurze Rechenzeiten zu erzielen.

1.2 Aufgabenstellung

Bei der Auswahl der eingesetzten Programmiersprache muss ein Trade-off zwischen der Performanz auf der einen und der Abstraktionsebene auf der anderen Seite eingegangen werden. Je höher die Sprache abstrahiert ist, desto größer ist der dadurch entstehende Overhead und desto geringer fällt die Performanz des Codes aus [Tra14]. Da die Performanz eines Codes im **H**igh-**P**erformance-**C**omputing-Bereich (HPC) sehr wichtig ist, werden fast ausschließlich systemnahe Programmiersprachen wie C oder Fortran eingesetzt. Diese Programmiersprachen bieten aber nur ein geringes Abstraktionslevel. Im Gegensatz dazu bietet Python ein hohes Abstraktionslevel, erreicht ohne Zusatzbibliotheken aber nur eine geringe Performanz. Der Einsatz der richtigen Bibliotheken erlaubt es, den sehr abstrahierten Python-Code zu beschleunigen. Ziel dieser Arbeit ist die parallele Programmierung numerischer Algorithmen aus Simulationscodes aus der Strömungstechnik und den Materialwissenschaften in Python. Die Performanz der Python-Implementierung soll möglichst nah an vergleichbaren C- oder Fortran-Implementierungen liegen und moderne Computer-Hardware mit Multi-Core-Prozessoren und GPGPUs möglichst effizient nutzen. Es soll am Beispiel einzelner Berechnungs-Kernel aus wissenschaftlicher Software gezeigt werden, ob folgende Hypothese bestätigt werden kann:

Mit der Programmiersprache Python und dazu erhältlichen Bibliotheken ist es möglich, eine ähnliche Performanz wie mit C oder Fortran bei komplexen Anwendungskernels zu erreichen.

1.2.1 Umfeld der Arbeit

Die vorliegende Arbeit wurde im Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) in der Einrichtung für Simulations- und Softwaretechnik erstellt.

Das **D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V. (DLR) ist die von der Bundesrepublik Deutschland beauftragte Forschungseinrichtung für Luft- und Raumfahrt und ist dadurch für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig. Die primären Forschungsthemen des DLR sind Luftfahrt, Raumfahrt, Energie und Verkehr. Die Forschungs- und Entwicklungsarbeiten in diesen Bereichen werden in Kooperation mit nationalen und internationalen Partnern durchgeführt.

Das DLR beschäftigt ca. 7700 Mitarbeiterinnen und Mitarbeiter. Die 32 Institute des DLR verteilen sich auf 16 nationale Standorte. Zusätzlich ist das DLR mit 4 internationalen Büros in Brüssel, Paris, Tokio und Washington D.C. vertreten [DLR14].
Die Abteilung „Verteilte Systeme und Komponentensoftware“wird von Herrn Andreas Schreiber geleitet. Die inhaltlichen Schwerpunkte dieser Abteilung liegen in den Bereichen Software Engineering, High Performance Computing und der Entwicklung verteilter Softwaretechnologie. Weiterhin hat diese Abteilung im DLR eine Berater- und Entwicklerrolle in den Bereichen Automatisierung und praktische Nutzung von Software-Engineering-Prozessen inne (vgl. [SC 13]).

Innerhalb des DLR bestehen die wesentlichen Aufgaben dieser Abteilung darin, neue Softwaretechnologien für das DLR und dessen Wissenschaftler zu entwickeln und einzuführen sowie bereits bestehende Projekte zu betreuen. Teilweise werden entwickelte Softwarewerkzeuge und -bibliotheken auch außerhalb des DLR zur Verfügung gestellt und eingesetzt.

Bei der vorliegenden Arbeit handelt es sich um die Bachelorarbeit in dem Studiengang Informationstechnik an der Dualen Hochschule Baden-Württemberg (DHBW) in Mannheim.

1.2.2 Vorgehensweise und Aufbau dieser Arbeit

Die zur Bewältigung der Aufgabe gewählte Vorgehensweise lässt sich in vier Kategorien einteilen. Zu Beginn steht die Einarbeitung in verschiedene Möglichkeiten zur effizienten parallelen Programmierung mit Python auf Multi-Core-Architekturen und Grafikkarten, gefolgt von einer Einarbeitung in relevante numerische Algorithmen, einer exemplarischen Implementierung paralleler numerischer Python-Algorithmen auf Multi-Core-Architekturen und Grafikkarten, sowie Test- und Performanz-Analyse der implementierten numerischen Algorithmen. Zu den verschiedenen Möglichkeiten der effizienten parallelen Programmierung mit Python auf Multi-Core-Architekturen gehören „Python Bindings for Global Array Toolkit“, Numbapro und Cython. Die Möglichkeit, **G**eneral-**P**urpose Computing on **G**raphics **P**rocessing **U**nits (GPGPU)s mit Python zu programmieren, wird mit Numbapro getestet, das einen Zugang zu der GPU-Schnittstelle CUDA bietet. Darauf folgt die Einarbeitung in einige numerische Algorithmen aus Simulationscodes aus der Strömungstechnik und den Materialwissenschaften, die als C- oder Fortran-Implementierung vorliegen. Anschließend werden die ausgewählten Algorithmen unter Beachtung geeigneter Software-Testverfahren exemplarisch mit Python implementiert. Aus Zeitgründen werden lediglich Python-Implementierungen für Kernels aus der Strömungstechnik untersucht. Erkenntnisse aus diesen Untersuchungen lassen sich im Wesentlichen auf iterative Lösungsverfahren in den Materialwissenschaften übertragen, da die zugehörigen Codes ähnliche Strukturen wie die in der Strömungstechnik aufweisen. Zusätzlich wurde die in den iterativen Lösungsverfahren aus den Materialwissenschaften äußerst wichtige dünnbesetzte Matrix-Vektor-Multiplikation bereits in Praxisbericht T3000[Ill14] behandelt. Zur exemplarischen Python-Implementierung gehört eine optimierte Single-Core-Variante, Multi-Core-Varianten mit Python Bindings for Global Array Toolkit, Numbapro und Cython sowie eine GPGPU-Variante mit Numbapro. Abschließend werden funktionale Tests der implementierten Routinen unter Verwendung von Unit-Tests, eine Performanz-Analyse des implementierten Algorithmus

unter Verwendung eines Performanz-Modells und von Performanz-Analyse-Tools und ein Performanz-Vergleich mit C- und Fortran-Implementierungen der verschiedenen Python-Implementierungen durchgeführt.

2 Grundlagen

In diesem Kapitel wird auf die theoretischen Grundlagen dieser Arbeit eingegangen. Diese beginnen mit einer Einführung in die Projekte, aus denen Algorithmen ausgewählt wurden. Darauf folgt eine Erläuterung der grundlegenden Architektur einer modernen **C**entral **P**rocessing **U**nit (CPU) und **G**raphics **P**rocessing **U**nit (GPU). Anschließend werden Besonderheiten der HPC-Programmierung, sowie die verwendete Programmiersprache Python einschließlich Zusatzbibliotheken, vorgestellt. Abschließend wird das Testsystem, auf dem alle Rechnungen durchgeführt wurden, beschrieben.

2.1 Freewake

Freewake ist Teil des Rotorsimulationscodes S4 der Abteilung Hubschrauber des Instituts für Flugsystemtechnik des DLR. Dieser dient dazu, dreidimensionale Strömungen um einen aktiv gesteuerten Rotor eines Helikopters zu simulieren [BRZH14].

2.1.1 Beschreibung des Projekts

Der Freewake-Code wurde zwischen 1994 und 1996 in Fortran entwickelt. Die Freewake-Methode basiert auf experimentellen Daten des HART-Programms [SKS+14]. Die Farbe und Dicke der dargestellten Röhren in Abbildung 1 veranschaulicht die Stärke und den Durchmesser der Wirbel im Nachlauf des Rotors.

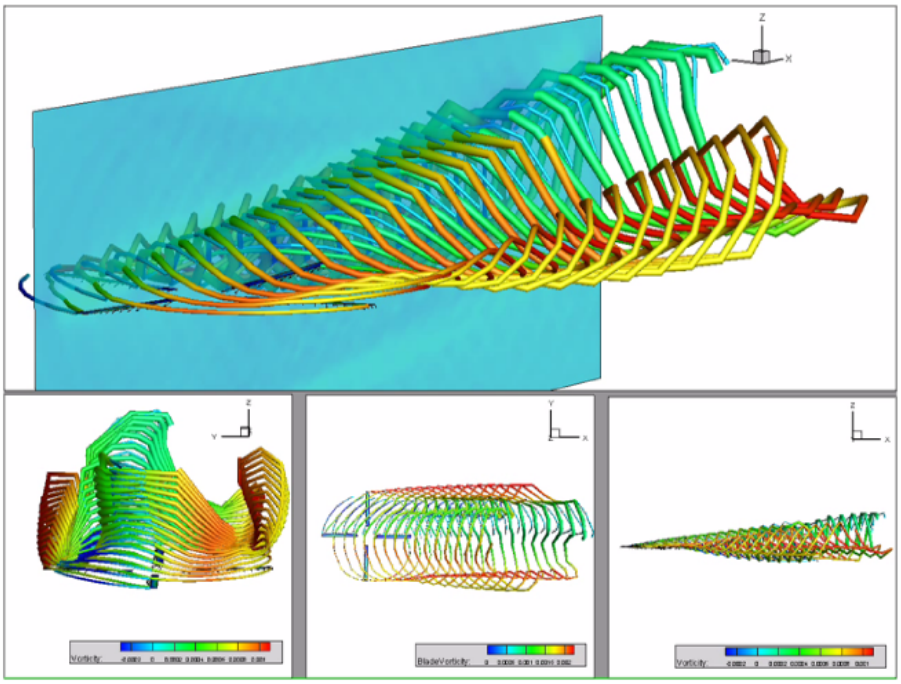


Abbildung 1: Visualisierung der von einem Hubschrauberrotor erzeugten Luftwirbel bei schnellem Vortwärtsflug

2.1.2 Technische Grundlagen

In der Vorwärtsbewegung bildet sich ein sehr asymmetrisches Strömungsverhalten. Dieses Strömungsverhalten ist die Ursache von einigen der Hauptprobleme von Hubschraubern: hohe Lautstärke, starke Vibrationen, hoher Energieverbrauch und Probleme bei der Flugstabilität. Die für einen Hubschrauber charakteristischen Geräusche entstehen, wenn ein Rotorblatt auf einen Luftwirbel trifft [BSSS87]. Im Gegensatz zu traditionellen CFD-Simulationen, bei denen die Berechnung der Strömungsgleichung auf jede Zelle in der Nähe des Rotors angewandt wird, wird bei Freewake ein 2D-Gitter in einem 3D-Raum verwendet, um die Wirbel darzustellen. In jedem Zeitschritt wird die induzierte Geschwindigkeit aller Wirbelelemente auf alle Knotenpunkte durch das Biot-Savert-Gesetz [Ada14] berechnet. Je nach Entfernung zwischen Wirbelelement und Knotenpunkt werden unterschiedliche Formeln verwendet. Der Code wurde ursprünglich mit Blick auf massiv parallele Systeme mit verteilten Speicher entwickelt. Die Kommu-

nikation wurde mit MPI umgesetzt. Mittlerweile unterstützt Freewake Multicore-CPUs sowie GPGPUs. Als Beispiel dient in dieser Arbeit ein kleiner Benchmark, in dem die induzierten Geschwindigkeiten mit einer vereinfachten Formel bestimmt werden.

2.2 Architektur der Recheneinheiten

In diesem Kapitel wird auf die Besonderheiten von CPU und GPU eingegangen. Dabei wird ein besonderes Augenmerk auf die für die parallele Programmierung und Performanz-Analyse interessanten Aspekte der jeweiligen Architekturen gelegt.

2.2.1 CPU

Eine moderne CPU besitzt eine geringe Anzahl an Prozessoren, sogenannte Cores. Diese Cores sind voneinander unabhängige leistungsstarke Rechenwerke. Durch mehrere Cores ist es möglich, mehrere Operationen parallel auszuführen. Zusätzlich ermöglichen **S**ingle **I**nstruction **M**ultiple **D**ata-Operationen (SIMD) eine Operation auf mehreren Daten auszuführen. Moderne Prozessoren bieten hierfür „große“Register, in welche zum Beispiel zwei **D**ouble **P**recision (DP)-Werte oder vier **S**ingle **P**recision (SP)-Werte gespeichert werden können. Auf diesen Daten kann dann eine Operation ausgeführt werden [HW10]. Ein SIMD-Beispiel ist in Abbildung 2 dargestellt.

Im Zuge der Performanzbetrachtung sind zwei Werte der CPU von besonderem Interesse. Dies ist zum einen die Peak-Performance und die maximale Speicherbandbreite. Die Peak-Performance gibt an, wie viele Operationen pro Sekunde bearbeitet werden können. Demnach wird sie in „**Fl**oating Point **O**perations **p**er **S**econd“(FLOPS) angegeben. Die CPU bekommt alle Daten vom Arbeitsspeicher. Deshalb ist die maximale Datentransferrate zwischen Arbeitsspeicher und CPU auch ein Wert, der bei der Performanz-Analyse verwendet wird. Dieser Wert wird maximale Speicherbandbreite genannt und in B/s angegeben.

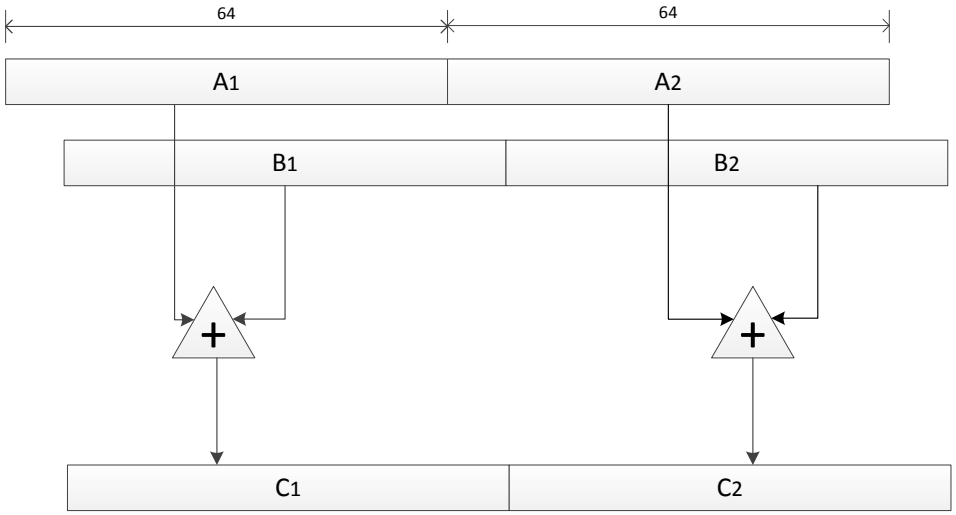


Abbildung 2: In den 128 Bit großen Registern A und B sind jeweils zwei 64 Bit große DP-Daten abgelegt und werden mit einer Instruktion addiert und in Register C geschrieben

Ursprünglich waren CPUs darauf ausgelegt, klassische serielle Anwendungen optimiert auszuführen. Dazu verwendet die CPU zum Beispiel Caches, welche örtliche und zeitliche Lokalität der Daten ausnutzen. Bei parallelen Programmen können dadurch auch Effekte auftreten, die die Performanz erheblich reduzieren.

2.2.2 GPU

Bei dem klassische Anwendungsfall einer GPU werden auf einer großen Datenmenge gleiche Befehle ausgeführt. GPUs sind am performantesten, wenn möglichst viele einfache Operationen gleichzeitig durchgeführt werden.

Die aktuelle Architektur von nVidia heißt Kepler. Diese besteht aus 15 **S**treaming **M**ultiprocessors (SMX) mit jeweils 192 SP CUDA[1]-Cores und 64 DP Units. 32 Cores bilden eine Gruppe (Warp). Alle Cores eines Warps führen die gleichen Instruktionen aus. Der **S**treaming **M**ultiprocessor (SMX)-Scheduler kann auf vier Warps gleichzei-

[1]CUDA ist eine NVIDIA Architektur für parallele Berechnungen, die die Rechenleistung des Systems durch Nutzung der Leistung des Grafikprozessors deutlich steigern kann [CUD14a].

Abbildung 3: Blockdiagramm des Nvidia Kepler GK110 Chip [Kep14]

tig Berechnungen durchführen. Dies ermöglicht es, 60 unterschiedliche Instruktionen gleichzeitig durchzuführen. Insgesamt können dadurch 1920 CUDA-Cores angesteuert werden.

Im Vergleich zu einer CPU besitzt eine GPU damit deutlich mehr Cores, hat aber eine niedrigere Taktrate. Dadurch kann paralleler Code auf einer GPU deutlich schneller ausgeführt werden als auf einer CPU. Zusätzlich kann eine GPU schneller zwischen Threads wechseln. Im Gegensatz dazu ist eine CPU durch eine höhere Taktrate bei seriellem Code deutlich im Vorteil.

Der Speicher einer Grafikkarte ist, wie in Abbildung 3 dargestellt, in unterschiedliche Ebenen eingeteilt. Eine GPU besitzt einen globalen Speicher, der von allen Threads aufgerufen werden kann, gemeinsamen Speicher, der innerhalb eines Blocks geteilt wird, lokalen Speicher, auf welchen nur ein lokaler Thread zugreifen kann und speziellen

Speicher, der abhängig von der Hardware ist. Der Cache auf der GPU ist in der Regel deutlich kleiner als auf einer CPU.

2.3 Besonderheiten der parallelen Programmierung

In diesem Kapitel wird zuerst auf die Besonderheiten der Speicherverteilung bei parallelen Systemen eingegangen, bevor die Besonderheiten der GPGPU-Programmierung und die CUDA-Architektur beschrieben werden

2.3.1 Systeme mit verteiltem Speicher

Daten in einem verteilten Speicher-System sind mit einem Prozessor verknüpft. Um auf Daten von einem anderen Prozessor zugreifen zu können, muss der Prozessor, auf dem die Daten gespeichert sind sowie der Speicherort der Daten auf dem zugehörigen Prozessor bekannt sein. Der Datenaustauch geschieht über ein externes Kommunikationsmedium (siehe Abbildung 4).

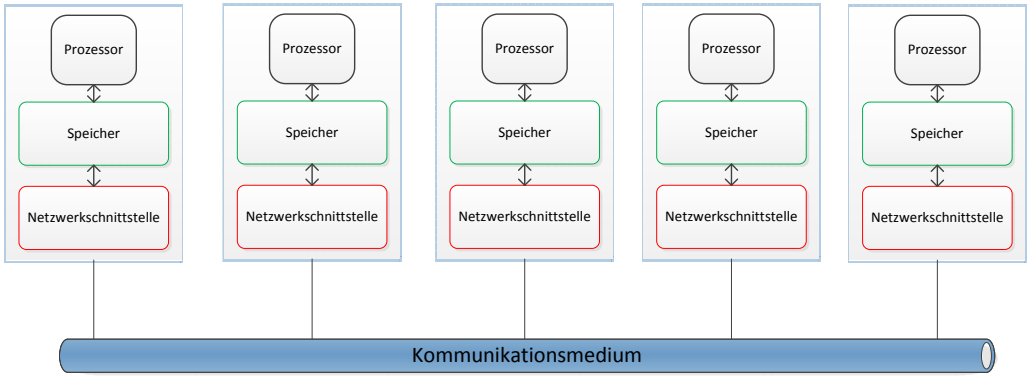


Abbildung 4: Verteilter Speicher

2.3.2 Systeme mit gemeinsamem Speicher

Bei einem System mit gemeinsamem Speicher liegen alle Daten in einem Speicher, auf den alle Prozessoren Zugriff haben. Dabei werden zwei Arten von Gemeinsame-Speicher-Systemn unterschieden.

- Bei einem „**U**niform **M**emory **A**ccess“(UMA)-System liegen alle Daten in einem gemeinsamen Speicher, auf welchen alle CPUs mit gleicher Latenz und Bandbreite zugreifen können (Abbildung 5).
- Ein „**c**ache-**c**oherent **N**on**u**niform **M**emory **A**ccess“(ccNUMA)-System ist physikalisch oft mit einem verteilten Speichersystem zu vergleichen. Ein solches System besitzt aber eine Logik, die es wie ein UMA-System erscheinen lässt. Da die Daten verteilt sind, dauert der Zugriff auf Daten, die auf einer anderen CPU liegen, länger als der Zugriff auf lokale Daten.

Ein ccNUMA-System gibt einem verteilten Speichersystem die Möglichkeit, auf Daten wie in einem UMA-System zuzugreifen [HW10].

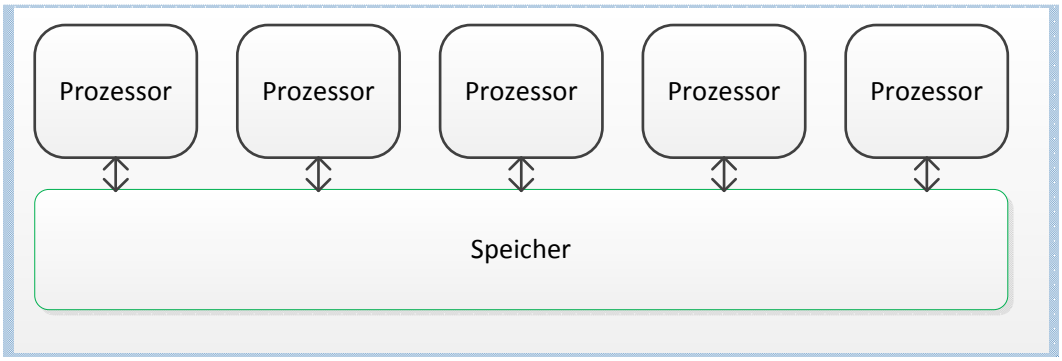


Abbildung 5: Gemeinsamer Speicher

2.3.3 Besonderheiten der GPGPU-Programmierung

Die Programmierung von GPGPUs ist deutlich komplexer als die Programmierung der CPU. GPGPUs haben üblicherweise eigenen Speicher. Daher müssen die Daten zwischen der CPU und der GPGPU transferiert werden. Zusätzlich müssen die verschiedenen Ebenen der Parallelität auf der GPGPU gesteuert werden. Nur wenn dies effizient ausgeführt wird, kann eine hohe Performanz erreicht werden.

2.3.4 CUDA

> „NVIDIA® CUDA® is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)“[Nvi14a]

CUDA ist eine Computing-Architektur von Nvidia für parallele Berechnungen. CUDA bietet über verschiedene Programmierschnittstellen Zugang zu den hochparallelen Funktionen von Nvidia GPUs [Par14].

Die Aufteilung auf GPU-Threads geschieht hierbei durch eine Einteilung der Rechnung in sogenannte Blöcke und Gitter (siehe Abbildung 6). Eine Device kann aus mehreren Grids bestehen. Die Grids wiederum bestehen aus Blöcken, die aus mehreren Threads bestehen.

2.4 Pythonbibliotheken zur Lösung wissenschaftlicher Problemstellungen

In diesem Kapitel werden die Programmiersprache Python, das Cython-Projekt sowie die Python-Bibliotheken Numpy, Numba und „Python Bindings for global Array Toolkit“ vorgestellt. Dabei wird besonders auf die für das HPC benötigten Eigenschaften geachtet.

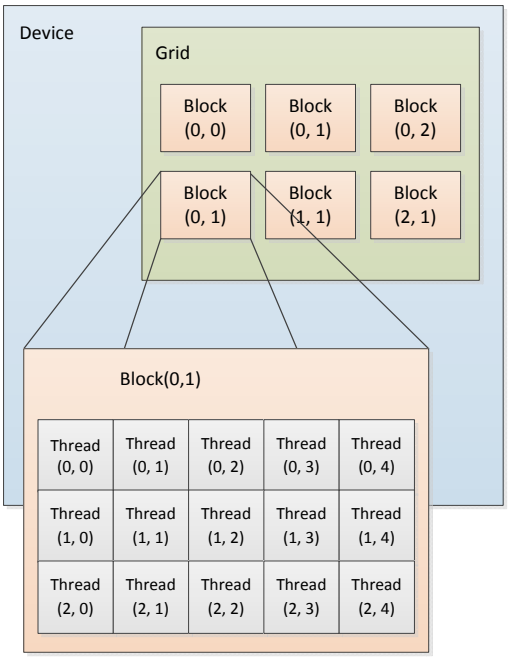


Abbildung 6: Darstellung der „CUDA Calculation Architecture“

2.4.1 Python

Python ist eine interpretierte, objektorientierte Script- und Programmiersprache. Python wurde von 1989 an von Guido van Rossum entwickelt. Mittlerweile ist Python zu einer der am weitesten verbreiteten Programmiersprachen aufgestiegen [Tio14] und ist die populärste Einstiegssprache an den Elite-Universitäten in Amerika [Guo14]. Die Hauptgründe, Python einzusetzen, sind:

Software Qualität

Python ist entwickelt worden, um lesbaren und dadurch wartbaren und wiederverwendbaren Code zu ermöglichen.

Entwickler-Produktivität

In Python entwickelter Code benötigt allgemein zwischen einem Drittel und einem Fünftel der Größe des gleichen Codes in C, C++ und Java. Dies ermöglicht es dem Entwickler, schneller lauffähigen Code zu produzieren.

Portabilität

Nahezu alle Python Programme laufen ohne Änderungen auf allen gängigen Betriebssystemen.

Unterstützte Bibliotheken

Python bietet eine große Anzahl an Standardfunktionen. Zusätzlich ist es möglich, Python durch eigene und Bibliotheken Dritter zu erweitern. Es existiert eine große Anzahl an Erweiterungen Dritter für einen breiten Bereich an Anforderungen. Diese reichen von der Erstellung von Websites über numerische Bibliotheken und Zugriff auf serielle Ports zur Entwicklung von Spielen.

Integration weiterer Komponenten

Python Code bietet eine einfache Möglichkeit, mit C-, C++-, Java-, .NET-Funktionen zu kommunizieren. Deshalb wird Python häufig eingesetzt, um verschiedene in anderen Sprachen entwickelte Funktionen übersichtlich zu einem Programm zusammen zu fügen.

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Abbildung 7: The Zen of Python

Die wichtigsten Prinzipien von Python und der Community dahinter sind in den in Abbildung 7 dargestelltem, „Zen of Python“ festgehalten.

Der größte Nachteil von Python gegenüber C oder C++ ist die geringere Geschwindigkeit, mit welcher Programme ausgeführt werden. Um diesen Effekt zu verringern oder gar zu eliminieren, wurden die in den folgenden Kapiteln vorgestellten Bibliotheken getestet [Lut13].

2.4.2 Cython

Cython ist ein Open-Source Projekt, ein Python-Code-Übersetzer und eine erweiterte Python-Programmiersprache [Cyt14c]. Die erweiterte Python-Programmiersprache ermöglicht es, C-Funktionen aufzurufen und C-Variablentypen und Klassenattribute zu verwenden. Der Python-Code wird in C-Code umgewandelt und von einem C-Compiler kompiliert [Cyt14a]. Sobald der Code ohne Python-Aufrufe kompiliert werden kann, ist es in Cython möglich, den „**G**lobal **I**nterpreter **L**ock“(GIL)[2] von Python zu lösen. Danach bietet Cython die Möglichkeit, mit OpenMP[3] zu parallelisieren.

2.4.3 Numpy

Die Numpy Bibliothek bietet Unterstützung für große N-dimensionale Arrays, Unterstützung für Problemlösungen im Bereich der linearen Algebra, die Möglichkeit, Fourier-Transformationen durchzuführen, sowie Tools zum Einbinden von C/C++- und Fortran-Code [Num14d]. Die Kombination von Python mit Numpy wird oft mit Matlab[4] verglichen. Die von Numpy ausgeführten Funktionen sind in der Regel deutlich schneller

[2] Der **G**lobal **I**nterpreter **L**ock (GIL) verhindert in Python, dass Python-Code parallel ausgeführt wird.
[3] OpenMP definiert eine API für portable und skalierbare Modelle zur Entwicklung paralleler Anwendungen [Ope14c].
[4] Matlab ist eine höhere Programmiersprache für numerische Berechnungen, Visualisierungen und Programmierung [Mat14].

als Standard-Python, da die rechenaufwändigen Funktionen in der Programmiersprache C implementiert sind.

2.4.4 Numba

Numba ist ein Open-Source-Optimierungskompiler, welcher Python und das Paket Numpy kompilieren kann. Es verwendet wie in Abbildung 8 dargestellt den LLVM-Compiler, um Python in Maschinencode zu kompilieren. Dabei werden unter anderem OpenMP, OpenCL[5] und CUDA unterstützt. Numba verfolgt das Ziel, bestehenden Code möglichst ohne Änderungen verarbeiten zu können. Trotzdem gibt es Einschränkungen bei der

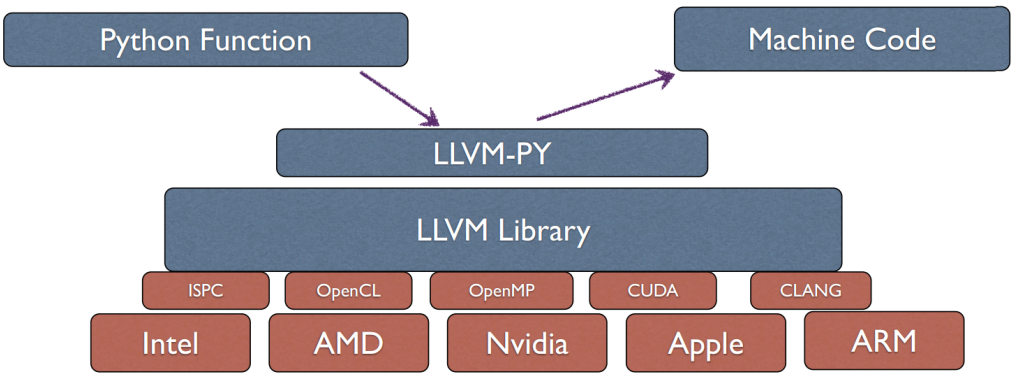


Abbildung 8: Kompilieren von Python Funktionen [Fun14]

Programmierung, die von der Art der Annotation abhängig sind. Durch Annotationen teilt der Entwickler einer Funktion dem Numba-Compiler mit, welche Funktionen kompiliert werden sollen. Dabei können unterschiedliche Geräte (Single-Core-CPUs und Multi-Core-CPUs) ausgewählt werden [Num14a].

[5] OpenCL ist ein freier plattformunabhängiger Standard für die Programmierung paralleler Prozessoren [Ope14b].

Numbapro

Numbapro ist eine von Continuum Analytics entwickelte kommerzielle Erweiterung von Numba. Es bietet die Möglichkeit, Code auf Singlecore-CPUs und Multicore-Prozessoren berechnen zu lassen. Numbapro erlaubt zusätzlich, mit Hilfe der CUDA-Schnittstelle auf nVidia GPGPUs Berechnungen durchzuführen [Nvi14a]. Außerdem unterstützt Numbapro die von Intel entwickelte „**M**ath **K**ernel **L**ibrary"(MKL). Sie ist laut Hersteller die schnellste und geläufigste mathematische Bibliothek für Intel-Prozessoren [MKL14].

2.4.5 Python Bindings for global Array Toolkit

Python Bindigs for global Array Toolkit ermöglicht es, auf verteilte Daten durch ein Shared-Memory-Interface zuzugreifen. Dadurch ist es möglich, auf ein verteiltes Array zuzugreifen, als wäre es in einem Gemeinsamen-Speicher-System. Folglich gehört es zu der Gruppe von **c**ache-**c**oherent **N**on**u**niform **M**emory **A**ccess (ccNUMA)-Systemen. Die als Erweiterungen für das **M**essage **P**assing **I**nterface (MPI)[6] gedachten global Arrays erlauben es dem Entwickler, in einem Programm zwischen dem Global-Array-Memory-Interface und dem verteilten Speicher-System von MPI zu wechseln. Der Zugriff auf ein wie in Abbildung 9 auf mehrere Prozessoren aufgeteiltes Array geschieht bei global Arrays zum Beispiel durch *ga.get(a,(3,2))*. Bei MPI müsste für einen Zugriff explizit eine Nachricht zwischen beiden Prozessoren ausgetauscht werden.

2.5 Testsystem

Die CPU des Testsystems ist ein Intel Xeon E5645 [E5614]. Diese besitzt sechs Cores, die mit jeweils 2,4 GHz [Int14] getaktet sind und einen 12288KB großen gemeinsamen Cache

[6] MPI ist eine Spezifikation für den Nachrichtenaustausch zwischen parallelen Prozessoren [MPI14].

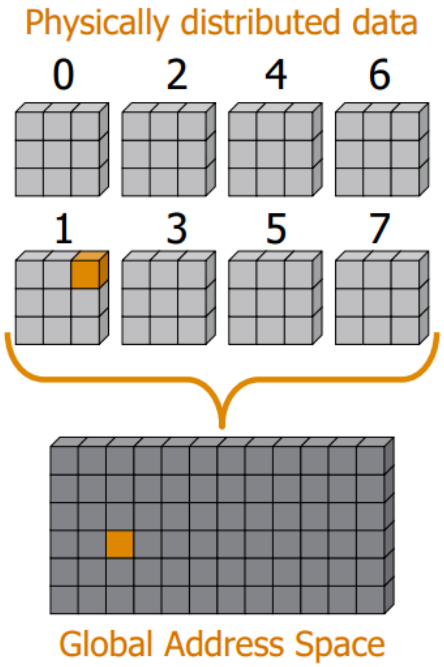


Abbildung 9: Datenzugriff bei Global Arrays [DSP+11]

besitzen. Die verbaute GPU ist eine Nvidia Tesla C2075. Die 448 CUDA-Cores sind mit 1150 MHz getaktet. Die Speicherbandbreite der GPU beträgt 144 GB/s [Nvi14b].

2.5.1 Theoretische Höchstleistung

In diesem Kapitel wird auf die theoretisch maximale Leistung der eingesetzten CPU und GPU eingegangen. Die sogenannte Peak-Performance beschreibt die maximale Anzahl an Gleitkomma Operationen pro Sekunde (**Fl**oating Point **O**perations **p**er **S**econd (FLOPS)).

Maximale Leistung der CPU

Die Peak-Performance lässt sich aus dem Produkt der CPU Geschwindigkeit (Hz), der Anzahl an Instruktionen pro Zyklus, der Anzahl der Cores und der Anzahl der CPUs pro Knoten bestimmen (siehe Gleichung 2.1).

$$FLOPS = Hz \times \frac{\text{instructions}}{\text{cycle}} \times \frac{\text{Cores}}{\text{CPU}} \times \frac{\text{CPU}}{\text{Knoten}} \tag{2.1}$$

Die Performanz-Analyse wurde auf einem Computer mit einer Intel Xeon E5645 CPU durchgeführt. Die sechs Kerne dieser CPU sind mit 2.40 GHz getaktet. Pro Kern und CPU-Takt können vier Instruktionen gleichzeitig ausgeführt werden. Wie in Gleichung 2.2 zu sehen ist, ergibt dies eine theoretisch Höchstleistung (Peak-Performance) von 57,6 GFLOPS.

$$57,6\,GFLOPS = 2.4GHz \times 4\,\frac{\text{instructions}}{\text{cycle}} \times 6\,\frac{\text{Cores}}{\text{CPU}} \times 1\,\frac{\text{CPU}}{\text{Knoten}} \tag{2.2}$$

Dies gilt für SP-Berechnungen. Bei DP-Rechnungen kann die CPU nur 2 Instruktionen pro Zyklus berechnen. Daher liegt die DP-Peak-Performance bei 28,8 GFLOPS.

Neben der Peak-Performance ist vor allem die Speicherbandbreite ein wichtiger Faktor bei der Performanz-Analyse. Die gemessene Speicherbandbreite liegt bei 19 GB/s. Diese wurde mit einem Stream-Triad-Benchmark[7] von likwid-bench[8] gemessen.

In Abbildung 10 werden die für die Performanz-Analyse benötigten Informationen in einer symbolischen Darstellung der CPU veranschaulicht.

[7] Addition zweier Vektoren und Multiplikation eines Skalar mit einen vierten Vektor
[8] Anwendung und Framework zum Ermitteln von Leistungsdaten [Lik14b]

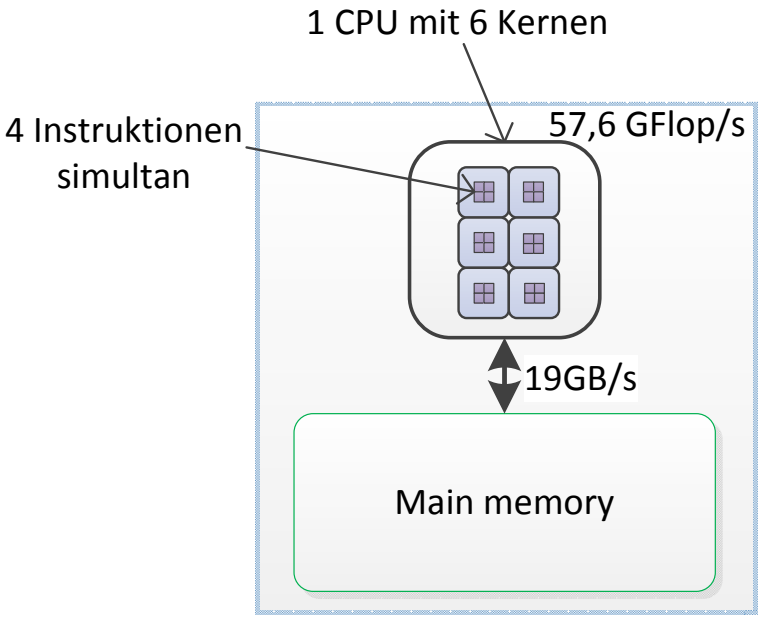


Abbildung 10: Symbolische Darstellung der CPU

Maximale Leistung der GPU

Die eingesetzte Nvidia Tesla C2075 GPU besitzt 448 mit 575 GHz getaktete Cores. Diese können pro Zyklus eine DP-Instruktion oder zwei SP-Instruktionen ausführen. Daraus ergibt sich für SP-Berechnungen die Gleichung 2.2:

$$1030\,GFLOPS = 1150\,MHz \times 448\,\text{Cores} \times 2\,\frac{\text{instructions}}{\text{cycle}}\,. \tag{2.3}$$

In Abbildung 11 sind die für die Performanz-Analyse benötigten Informationen in einer symbolischen Darstellung der GPU dargestellt. Bei DP-Berechnungen kann die GPU auf Grund der halben Anzahl von Instruktionen pro Zyklus mit 515 GFLOPS nur die Hälfte der SP-Peak-Performanz erreichen.

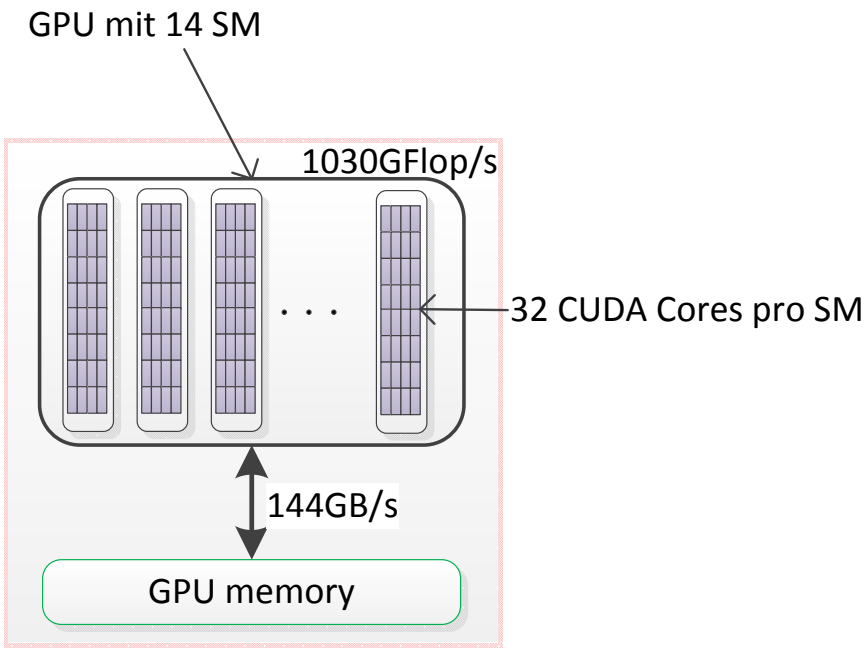


Abbildung 11: Symbolische Darstellung der GPU

2.5.2 Performanz-Modellierung

Eine einfache Möglichkeit, die Performanz zu modellieren, beschreibt das Roofline-Modell. Es dient dazu, eine obere Performanzgrenze für einen Algorithmus, abhängig von dem Verhältnis von Daten zu Operationen [WWP09], zu ermitteln.

Das Roofline-Modell nennt dieses Verhältnis Balance. Die Balance einer Maschine lässt sich durch den Quotienten von Speicherbandbeite und der theoretischen Leistung der CPU bestimmen:

$$B_m = \frac{\text{Speicherbandbreite}\,[\text{GBytes/s}]}{\text{Peak Performanz}\,[\text{GFlops/s}]}. \tag{2.4}$$

Um die Anforderungen eines Algorithmus zu bestimmen, kann die Code-Balance ermittelt

werden. Dafür wird, wie in Gleichung 2.5 dargestellt, das Verhältnis von Datentransfers und Gleikommaoperationen bestimmt:

$$B_c = \frac{\text{data traffic[Bytes]}}{\text{floating point ops[Flops]}}. \tag{2.5}$$

Data Traffic bezieht sich auf den limitierenden Datentransportweg. Mit der Code-Balance und der Maschinen-Balance kann der maximal zu erreichende Faktor der theoretischen Leistung der CPU berechnet werden.

$$l = \min\left(1, \frac{B_m}{B_c}\right) \tag{2.6}$$

Der betrachtete Code wird aufgrund von Bandbreitenbegrenzungen keine höhere Performanz als $\text{Peak Performance} \cdot l$ erreichen können [HW10].

3 Implementierung von Freewake

In diesem Kapitel wird die Implementierung des Freewake-Algorithmus mit Python, Cython, Numpy, Numba und „Python Bindings for global Array Toolkit“beschrieben.

Bei der Implementierung von Freewake wird die Zeit gemessen, die benötigt wird, um die von Wirbeln induzierten Geschwindigkeiten bei einer Drehung des Rotors um 1 Grad zu simulieren. Dazu wird ein Gitter zur Diskretisierung der Wirbel über den Rotor gelegt. Für jeden Gitterknoten wird die induzierte Geschwindigkeit berechnet. Die hierbei benötigten Daten pro Gitterpunkt werden in einem 4-dimensionalen Array gespeichert. Die Geschwindigkeit setzt sich aus der induzierten Geschwindigkeiten der Längswirbel und der Querwirbel zusammen. Daraus ergibt sich eine quadratisch zu der Anzahl der Gitterpunkten wachsende Laufzeit des Algorithmus. Die in Quellcode 14 und 15 im Anhang dargestellten Funktionen waren der Ausgangspunkt der folgenden Implementierungen. Zur Berechnung der induzierten Geschwindigkeiten gibt es zwei verschiedene Methoden: Zum einen ist es möglich, der Funktion ein Array zu übergeben und auf dem Array elementweise Operationen auszuführen. Zum anderen können in die Schleife vor dem Funktionsaufruf drei weitere Schleifen eingefügt werden, um der Funktion einzelne Werte zu übergeben.

3.1 Implementierung mit der Python-Standard-Bibliothek

Mit Python-Listen ist es nicht möglich zu rechnen. Dies liegt daran, dass eine Liste aus Elementen mit unterschiedlichen Datentypen bestehen kann und der Interpreter nicht entscheiden kann, welche Operation ausgeführt werden soll. Man kann jedoch die benötigten Operationen für Listen implementieren, bei denen der Datentyp der Elemente bekannt ist. Diese Berechnungen auf Listen sind in Python langsamer, als drei weitere Schleifen einzufügen und der aufzurufenden Funktion einzelne Werte zu übergeben. Daher erfolgt der Schleifendurchlauf in Python wie in Quellcode 1 dargestellt.

```
1  for iblades in range(numberOfBlades):
2      for iradial in range(1, dimensionInRadialDirection):
3          for iazimutal in range(dimensionInAzimualDirectionTotal):
4              for i1 in range(len(vx[0])):
5                  for i2 in range(len(vx[0][0])):
6                      for i3 in range(len(vx[0][0][0])):
7                          #wilin-Aufruf 1
8  for iblades in range(numberOfBlades):
9      for iradial in range(dimensionInRadialDirection):
10         for iazimutal in range(1, dimensionInAzimualDirectionTotal):
11             for i1 in range(len(vx[0])):
12                 for i2 in range(len(vx[0][0])):
13                     for i3 in range(len(vx[0][0][0])):
14                         #wilin-Aufruf 2
15 for iDir in range(3):
16     for i in range(numberOfBlades):
17         for j in range(dimensionInRadialDirection):
18             for k in range(dimensionInAzimualDirectionTotal):
19                 x[iDir][i][j][k] = x[iDir][i][j][k] + dt * vx[iDir][i][j][k]
```

Listing 1: Aufruf der Funktion *wilin* in Python

In den ersten sieben Zeilen wird die Induktion der Querwirbel und in den Zeilen acht bis vierzehn die Induktion der Längswirbel berechnet. Anschließend werden die Gitterpunkte aktualisiert. Bei dem ersten Aufruf bekommt die Funktion *wilin* aus Quellcode 2 die Parameter:

- x[:][i1][i2][i3] - x[:][iblades][iradial][iazimutal]
- x[:][i1][i2][i3] - x[:][iblades][iradial - 1][iazimutal]
- transversalVorticity[iblades][iradial][iazimutal]
- transversalVorticity[iblades][iradial - 1][iazimutal]
- transversalVortexLength[numberOfBlades-1][dimensionInRadialDirection-1][iazimutal]
- vx[:][i1][i2][i3]

Die Rückgabewerte dieser Funktion werden in vx[:][i1][i2][i3] gespeichert. Bei dem zweiten Aufruf wird bei den Parametern eins, zwei und vier anstatt *iradial*, *iazimutal* um eins subtrahiert. Hierbei wird jede Differenz doppelt berechnet. Eine Wiederverwendung der Differenzen der Gitterpunkte benötigt einen zur Anzahl an Schleifendurchlaufen quadratischen Speicher oder ein neues Anordnen der Schleifen. Dies wurde hier aber weder in der Fortran- noch in den Python-Implementierungen umgesetzt, um einen vergleichbaren Algorithmus zu erhalten. In Quellcode 2 sieht man die Funktion *wilin*, mit der die induzierten Geschwindigkeiten eines Wirbelsegments berechnet wird.

```
def wilin(dax, day, daz, dex, dey, dez, ga, ge, wl, vx, vy, vz):
    rcq = 0.1
    daq = dax**2 + day**2 + daz**2
    deq = dex**2 + dey**2 + dez**2
    da = math.sqrt(daq)
    de = math.sqrt(deq)
    dae = dax * dex + day * dey + daz * dez
    sqa = daq - dae
    sqe = deq - dae
    sq = sqa + sqe
    rmq = daq * deq - dae**2
    fak = ((da + de) * (da * de - dae) * (ga * sqe + ge * sqa) + (ga - ge) * (da
        ↪  - de) * rmq) / (sq * (da * de + eps) * (rmq + rcq * sq))
    fak = fak * wl / math.sqrt( sq )
    vx = vx + fak * (day * dez - daz * dey)
    vy = vy + fak * (daz * dex - dax * dez)
    vz = vz + fak * (dax * dey - day * dex)
    return vx, vy, vz
```

Listing 2: Funktion zur Berechnung der vom Wirbel induzierten Geschwindigkeit

3.2 Implementierung mit Cython

Cython unterscheidet sich von den anderen Implementierungen vor allem, durch die Typenangaben der Variablen. Alle Cython-Variablen werden mit

```
cdef <Variablentyp> <Variablenname>
```

erzeugt. Die Arrays werden zuerst als Numpy-Arrays initialisiert, danach aber sofort in Memoryviews umgewandelt. Diese sind ein Cython-Feature und ermöglichen es, effizient auf Numpy-Arrays ohne den bremsenden Python-Overhead zuzugreifen. Die

Funktion mit dem Aufruf der *wilin*-Funktion wurde in Cython überarbeitet. Die Arrays werden als Memoryviews verwendet, die Reihenfolge der Schleifen wurde verändert, der Rückgabewert der aufgerufenen Funktion wird in ein Struct geschrieben und der Schleifendurchlauf wurde parallelisiert. Die Schleifen mit den Laufvariablen *iblades*, *iradial* und *iazimutal* aus Quellcode 1 wurden hinter die Schleife mit der Laufvariable *i3* verschoben, um eine Parallelisierung der Schleifen zu ermöglichen. Dies ist notwendig, da die Geschwindigkeiten in *vx* über die Schleifen *iblades*, *iradial* und *iazimutal* aufsummiert wird. Der Rückgabewert der Funktion 5 ist ein aus mehreren Werten bestehendes Struct. Daher muss diese in ein Struct geschrieben und anschließend aus dem Struct in die entsprechenden Felder des Arrays geschrieben werden. Nach diesen Änderungen wird für die Ausführung der Schleifen auf keine Python-Objekte mehr zugegriffen. Dies ermöglicht es, diesen Teil des Programms mit dem Statement *nogil* im *Nogil*-Modus auszuführen. Dadurch ist es möglichden GIL zu umgehen und die Funktion *prange* zu verwenden. Diese parallelisiert Schleifen mit OpenMP. Die besten Ergebnisse beim Parallelisieren werden erreicht, wenn sich die *prange*-Schleife möglichst weit außen befindet. Dies verhindert, dass die Threads bei jedem Schleifendurchlauf der äußeren Schleife erzeugt und beendet werden müssen. Die Schleife mit der Laufvariablen *i1* hat in dem Test-Code nur vier Durchläufe. Da die CPU des Testsystems sechs Cores besitzt, wird nicht die optimale Performanz erreicht, wenn diese Schleife parallelisiert wird. Die Schleife mit der Laufvariable *i2* wird in diesem Beispiel 12 mal durchlaufen und erzielt bei Parallelisierung die beste Performanz. Ein Parallelisieren von mehreren Schleifen erzielt keine bessere Performanz, da der Overhead beim Erzeugen und beim Beenden der Threads eine erhöhte Performanz verhindert. Einen weiteren Performanz-Schub von 20% erzeugt ein Vertauschen der Schleifen mit den Laufvariablen i1 und i2, so dass die Schleifen wie in Quellcode 3 angeordnet sind. Damit der Code von Cython parallel ausgeführt wird, muss dem Cython-Compiler noch das Compiler-Argument und das Linker-Argument *-fopenmp* mitgeteilt werden. Durch dieses Argument wird dem Compiler angezeigt, dass der Code mit OpenMP parallelisiert werden soll.

```
1 for i2 in prange(dimensionInRadialDirection):
2     for i1 in xrange(numberOfBlades):
3         for i3 in xrange(dimensionInAzimualDirectionTotal):
4             for iblades in xrange(numberOfBlades):
5                 for iradial in xrange(1, dimensionInRadialDirection):
6                     for iazimutal in xrange(dimensionInAzimualDirectionTotal):
```

Listing 3: Optimierte Anordnung der Schleifen in Cython

Für die oben beschriebene Parallelisierung muss auch die *wilin*-Funktion umgeschrieben werden, sodass sie auf keine Python-Objekte zugreifen muss. Um erkennen zu können auf welche Python-Objekte zugegriffen wird, wurde die *–annotate*-Option von Cython verwendet. Diese gibt an, an welchen Stellen auf Python-Objekte zugegriffen wird und zeigt durch Farben kodiert an, wie lange jede Zeile benötigt. Zusätzlich können Funktionen annotiert werden, um die Verwendung bestimmter Compiler-Direktiven zu erzwingen. *Cdivision(True)* deaktiviert Python Tests bei Divisionen. Die Python-Tests können die Ausführung des Codes um 35 % bremsen. Durch die Direktive *boundscheck(False)* nimmt Cython an, dass keine Fehler bei der Indizierung von Arrays auftreten [Cyt14b]. Wie in C ist es auch in Cython nicht möglich von einer Funktion aus mehrere Werte zurück zu geben. Dies wurde durch das Erstellen des Structs *myret* umgangen (siehe Quellcode 4).

```
1 cdef struct myret:
2     double vz1
3     double vx1
4     double vy1
```

Listing 4: Struct mit drei Double zur Rückgabe in Cython

Die Funktion *wilin* bekommt eine Variable des Struct übergeben und besitzt als Rüchgabetyp dieses Struct. Die Rückgabewerte werden in den Zeilen 19-21 des Quellcodes 5 in das Struct gepackt und anschließend zurückgegeben. Die Berechnungen der Quadrat-

wurzel in Zeile 7, 8 und 15 wurden durch den Import von *libc.math* an C-Funktionen übergeben.

```
1  @cython.cdivision(True)
2  @cython.boundscheck(False)
3  cdef myret wilin_mem(double dax, double day, double daz, double dex, double dey,
       ↪  double dez, double ga, double ge, double wl, double vx1, double vy1,
       ↪ double vz1, myret ret)nogil:
4      cdef double rcq = 0.1
5      cdef double daq = dax**2 + day**2 + daz**2
6      cdef double deq = dex**2 + dey**2 + dez**2
7      cdef double da = sqrt(daq)
8      cdef double de = sqrt(deq)
9      cdef double dae = dax*dex + day*dey + daz*dez
10     cdef double sqa = daq - dae
11     cdef double sqe = deq - dae
12     cdef double sq = sqa + sqe
13     cdef double rmq = daq * deq - dae**2
14     cdef double fak = ((da + de) * (da*de - dae) * (ga*sqe + ge*sqa) + (ga - ge)
           ↪  * (da - de) * rmq) / (sq * (da*de + eps) * (rmq + rcq*sq))
15     cdef double fak2 = fak * wl / sqrt(sq)
16     ret.vx1 = vx1 + fak2 * (day*dez - daz*dey)
17     ret.vy1 = vy1 + fak2 * (daz*dex - dax*dez)
18     ret.vz1 = vz1 + fak2 * (dax*dey - day*dex)
19     return ret
```

Listing 5: Funktion zur Berechnung der von einem Wirbelsegment induzierten Geschwindigkeit in Cython

3.3 Implementierung mit Numpy

Mit dem Import von Numpy eröffnet sich in Python die Möglichkeit, schneller mit Arrays zu rechnen. Alle Listen, die in Python unter ausschließlicher Verwendung der Standardbibliotheken erstellt wurden, können wie in Quellcode 6 dargestellt als Numpy Arrays erstellt werden. Das Tupel "shape" gibt die Größe des Arrays und die Variable "dtype" den Datentyp der Arrayelemente an.

```
import numpy as np
x = np.ones(shape = (dim1, dim2, dim3, dim4), dtype = np.float64)
```

Listing 6: Erstellung eines Numpy-Arrays

Auf diesen Arrays können nun eine Vielzahl an Operationen ausgeführt werden. Diese Operationen sind wie in Kapitel 2.4.4 beschrieben in C implementiert. Daher erhält man die beste Performanz, wenn der Funktion *wilin* Arrays übergeben werden. Deshalb wird der Aufruf der Funktion wie in Codeausschnitt 7 dargestellt ausgeführt.

```
1  for iblades in xrange(numberOfBlades):
2      for iradial in xrange(1, dimensionInRadialDirection):
3          for iazimutal in xrange(dimensionInAzimualDirectionTotal):
4              #wilin-Aufruf 1
5  for iblades in xrange(numberOfBlades):
6      for iradial in xrange(dimensionInRadialDirection):
7          for iazimutal in xrange(1, dimensionInAzimualDirectionTotal):
8              #wilin-Aufruf 2
9  for iDir in range(3):
10     for i in range(numberOfBlades):
11         for j in range(dimensionInRadialDirection):
12             for k in range(dimensionInAzimualDirectionTotal):
13                 x[iDir][i][j][k] = x[iDir][i][j][k] + dt * vx[iDir][i][j][k]
```

Listing 7: Aufruf der *wilin*-Funktion mit Numpy

In den ersten vier Zeilen wird die Induktion der Querwirbel und in den Zeilen fünf bis 8 die Induktion der Längswirbel berechnet. Anschließend werden die Gitterpunkte aktualisiert. Dadurch ändern sich auch die Parameter, die der Funktion übergeben werden. Aus *x[0][i1][i2][i3] - x[0][iblades][iradial][iazimutal]* wird *x[0] - x[0, iblades, iradial, iazimutal]*. Diesem Schema folgend ändert sich auch die Zugriffsart auf die anderen Parameter. Die Operatoren +, *, - und / führen bei Numpy-Arrays zu elementweisen Operationen. Werden zwei Arrays miteinander multpliziert, dividiert, subtrahiert oder addiert wird jedes Element mit dem Element des anderen Arrays mit der ausgewählte Operation verknüpft, das an derselben Position liegt. In Abbildung 12 wird hierfür beispielhaft eine Multiplikation zweier gleich großer Arrays dargestellt.

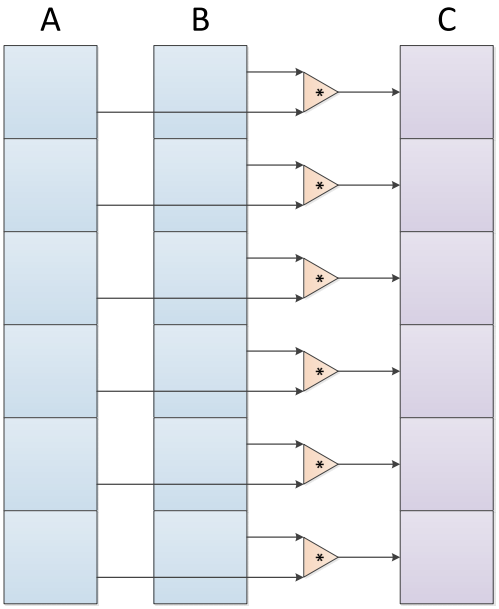


Abbildung 12: Multiplikation zweier gleich großer Numpy-Arrays

Die Funktion *wilin* aus der reinen Python-Implementierung kann bis auf eine Änderung bei der Wurzelfunktion übernommen werden. Hier wird anstelle der *math.sqrt*-Funktion die *numpy.sqrt*-Funktion aufgerufen.

3.4 Implementierung mit Numba

Unter Verwendung von Numba wurden drei Versionen implementiert. Alle Versionen bauen auf der Numpy-Implementierung auf. Mit Numba wurde der Code für einen Core einer CPU optimiert, für Multi-Core-CPUs und für GPGPUs.

3.4.1 Single-Core-Optimierung

Numba versucht soviel wie möglich im beschleunigten „nopython"-Modus auszuführen. Das Erstellen von Numpy-Arrays ist darin allerdings nicht möglich. In der Single-Core Variante wurde der mit Numpy erstellte Code als Grundlage verwendet. Bei diesem wurde die Funktion 2 und die Funktion, die die Schleifen aus Quellcode 7 beinhaltet, mit der Numba Annotation *autojit* versehen. Die *autojit*-Annotation teilt Numba mit, dass diese Funktion in Maschinencode kompiliert werden soll. Bei der *autojit*-Annotierung erkennt Numba beim Aufruf der Funktion automatisch, welche Daten und Datentypen übergeben werden.

3.4.2 Multi-Core-Optimierung

Bis zur Numba-Version 0.11 gab es das Kommando „prange". Dies ermöglichte es, Schleifen ähnlich wie in Cython mit OpenMP zu parallelisieren. Dabei wurde für jeden Schleifendurchlauf ein eigener Thread gestartet und die Threads auf alle CPU-Cores verteilt. Die Möglichkeit, „prange" zu verwenden, wurde mit der Version 0.12 entfernt, da es nach Aussage eines Entwicklers Probleme mit der Stabilität und der Performanz gab [Num14c]. Eine weitere Möglichkeit Funktionen zu parallelisieren, bietet das „parallel"-Backend der *Vectorize*-Funktion. Dabei wird die gleiche Operation auf alle Einträge einer Liste ausgeführt [Num14b]. Der getestete Freewake-Code ist nicht vektorisierbar, weil eine mit Numba vektorisierte Funktion nur einen Rückgabewert besitzen kann und

weil das Aufsummieren der Geschwindigkeiten in so einer Funktion nicht möglich ist. Daher war eine Multi-Core-Optimierung mit Numba nicht möglich.

3.4.3 GPU Optimierung

Die CUDA-JIT-Annotation bietet einen systemnahen Cuda-Einstiegspunkt. Eine damit annotierte Funktion wird auf der GPU ausgeführt. Ein solcher Kernel benötigt neben den übergebenen Parametern auch die zusätzlichen Parameter *blockdim* und *griddim*. Der Parameter *griddim* gibt an, wie viele Thread-Blöcke in einem Grid und der Parameter *blockdim*, wie viele Threads pro Thread-Block gestartet werden sollen. In Abbildung 13 ist dargestellt, wie die Threads auf der GPU aufgeteilt wurden. Bevor auf der GPU

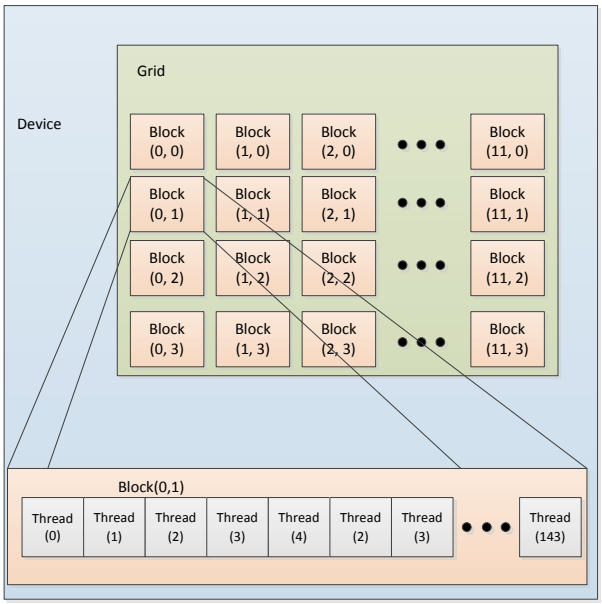


Abbildung 13: Aufteilung in Blöcke und Threads

Berechnungen durchgeführt werden können, müssen die Daten aus dem Speicher der CPU an den Speicher der GPU gesendet werden. Durch den Befehl *cuda.to_device(vx)* wird die Variable *vx* in den Speicher der GPU übertragen. Anschließend kann auf der GPU auf diese Daten zugegriffen werden. Sobald alle Berechnungen ausgeführt wurden,

werden die Daten mit *vx.to_host()* zurück in den Speicher der CPU kopiert. In Abbildung 14 ist der Weg der Daten dargestellt.

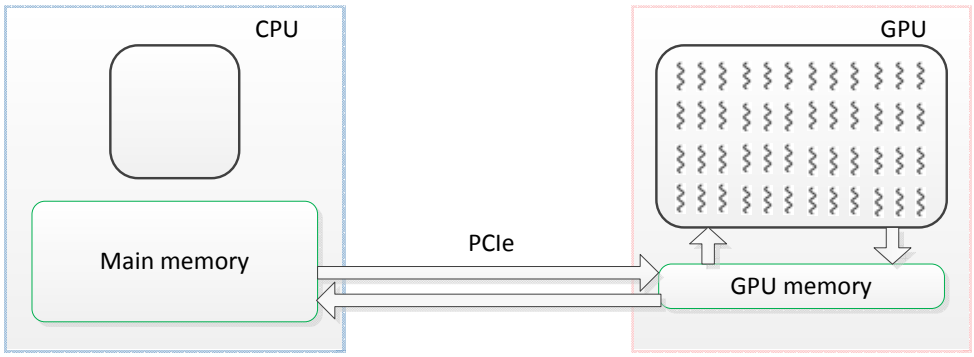


Abbildung 14: Weg der Daten bei einer Berechnung auf der GPU

```
1 i1 = cuda.blockIdx.x
2 i2 = cuda.blockIdx.y
3 i3 = cuda.threadIdx.x
```

Listing 8: Identifikation des CUDA-Threads

Für jeden Schleifendurchlauf der Schleifen i1 bis i3 wurde durch die in Abbildung 13 dargestellten Blöcke der Dimension *blockdim* und das dargestellte Block-Gitterder Dimension *griddim* ein Thread erzeugt. In dem Kernel wurde durch Quellcode 8 der Thread identifiziert. Nachdem die *iblades*-Schleife komplett durchlaufen ist, wurden mit *cuda.syncthreads()* die Threads synchronisiert, um sicher zu stellen, dass alle Berechnungen beendet wurden, bevor die Daten zurück zur CPU übertragen werden.

3.5 Implementierung mit „Python Bindings for global Array Toolkit“

Die Implementierung mit global Arrays basiert auf der Implementierung mit Numpy. Die Funktion wilin (Quellcode 2) konnte übernommen werden. Bei der Initialisierung der

Arrays müssen zusätzlich zu den Numpy-Arrays global Arrays der gleichen Dimension und desselben Datentyps angelegt werden (Quellcode 9).

```
g_x = ga.create(ga.C_DBL, [dim1, dim2, dim3, dim4])
```

Listing 9: Erstellung eines global Array

Bei dieser Implementierung wurde darauf geachtet, dass global Arrays mit g_ benannt wurden und lokale Teil-Arrays mit l_. Dabei dienen die Numpy-Arrays dazu, die Berechnungen auf lokalen Variablen auszuführen, um dann am Ende der Initialisierung die Daten aus den Numpy-Arrays in die globalen Arrays zu schreiben (Quellcode 10).

```
ga.put(g_x, l_x)
```

Listing 10: Schreiben in ein global Array

Ein Nachteil dieser Implementierung ist, dass einige Arrays auf jedem Knoten vorhanden sein müssen, wodurch sich der benötigte Speicherplatz erhöht. Zwei essentielle Funktionen bei global Arrays sind *ga.nodeid()* und *ga.nnodes()*. Durch *ga.nodeid()* kann die ID des aktuellen Knoten und durch *ga.nnodes()* die Anzahl an Knoten abgefragt werden. Diese Funktionen werden benötigt, wenn Befehle nur einmal ausgeführt werden sollen. Beispiele dafür sind die Zeitmessung und die Ausgabe. Das „global Array Toolkit“ übernimmt das Aufteilen bei Arrays, wenn dieses nicht anders festgelegt wurde. Das Array wird dabei gleichmäßig auf die Knoten verteilt. Der Bereich des Arrays, der von einem Prozess verwaltet wird, kann wie in Quellcode 11 exemplarisch dargestellt abgerufen werden.

```
lo,hi = ga.distribution(g_x, node)
```

Listing 11: Ermitteln des von einem Knoten verwalteten Array-Bereichs

An Stellen, an denen alle Prozesse warten müssen, bis alle Prozesse diesen Punkt erreicht haben, muss eine Funktion zur Synchronisierung verwendet werden (Abbildung

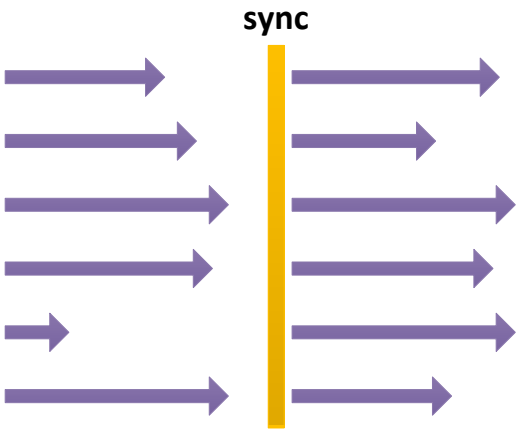


Abbildung 15: Prozess-Synchronisierung

15). In dem global Array Toolkit stellt diese Funktion (Quellcode 12) zudem sicher, dass alle Aktionen auf den global Arrays abgeschlossen wurden.

```
ga.sync()
```

Listing 12: Synchronisierung aller Prozesse

Die Idee bei der Parallelisierung besteht darin, dass jeder Prozess nur für einen Teilbereich des Arrays die induzierten Geschwindigkeiten berechnet (siehe Abbildung 16) und am Ende sein Ergebnis auf das globale Array schreibt. Eine Aufteilung des Problems in kleinere Teilprobleme ist möglich, da die Teilprobleme voneinander unabhängig sind. Dabei wird durch den Befehl

```
l_x = ga.get(g_x,lo,hi)
```

der Array-Bereich in eine lokale Variable geladen, den ein Knoten bearbeiten soll. Sobald ein Knoten mit all seinen Berechnungen fertig ist, schreibt er seine Ergebnisse aus dem lokalen Array mit

```
ga.put(g_x, l_x, lo,hi)
```

in den entsprechenden Bereich das global Array. Der Aufruf der Funktion *wilin* ähnelt dem Aufruf in Numpy. Der einzige Unterschied besteht darin, dass das erste Array aus

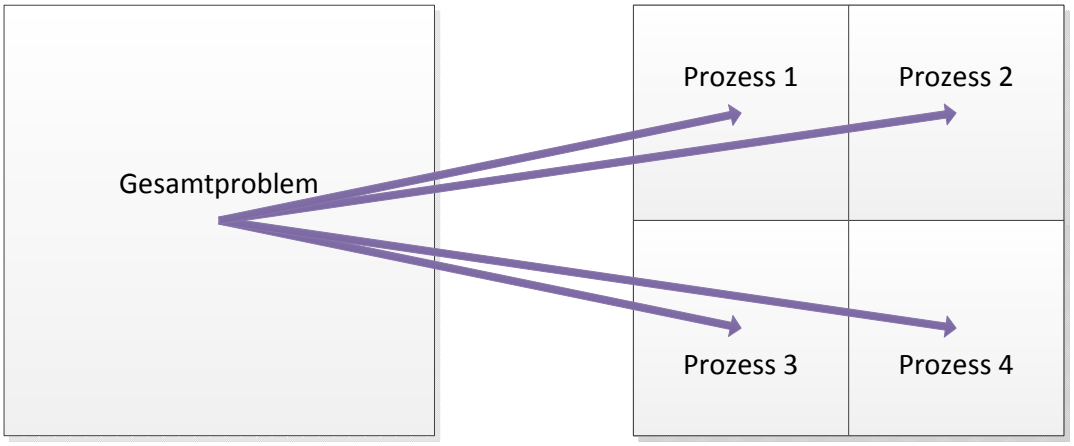


Abbildung 16: Aufteilungsschema eines Problems in Teilprobleme

x[0] - x[0, iblades, iradial, iazimutal] durch das lokale Array (l_x) ersetzt wird. Das zweite Array bezieht sich weiterhin auf das gesamte Array; in diesem Fall g_x. Gestartet wird ein solches Programm über das MPI-Franework (**M**essage **P**assing **I**nterface) wie folgt:

```
mpirun -np 6 ./globalArray.py
```

4 Test und Performanz-Analyse der implementierten Algorithmen

Zu Beginn dieses Kapitels, werden die eingesetzten Tests zur Sicherstellung der funktionalen Korrektheit der Implementierungen beschrieben. Folgend werden drei bei der Performanz-Analyse unterstützende Tools vorgestellt. Anschließend wird die Performanz der einzelnen Implementierungen analysiert und verglichen.

4.1 Eingesetzte Tests

Um die funktionale Korrektheit der unterschiedlichen Implementierungen sicher zu stellen, wurden mit dem PyUnit-Framework Tests entwickelt. Dieses Unit-Testing-Framework ist eine Python-Version des Java-Test-Framework JUnit [PyU14]. Die Tests sind unterschiedlich komplex und haben dadurch eine unterschiedliche Laufzeit. Ein Test vergleicht die Größe des Arrays *vx*, einer die Quersumme des Arrays *vx* und einer vergleicht jedes Element des Arrays *x* mit einem korrekten Ergebnis. Die ersten zwei Tests wurden entwickelt, um einen Überblick zu bekommen, ob das Ergebnis richtig sein kann. Um sicher zu gehen, dass das Ergebnis vollständig richtig ist, muss das Ergebnis mit dem länger laufenden dritten Test überprüft werden.

4.2 Verwendete Tools

In diesem Abschnitt werden die Tools vorgestellt, die bei der Optimierung des Codes und bei der Performanz-Analyse eingesetzt wurden.

4.2.1 Cython --annotate

Cython bietet durch die Option --annotate die Möglichkeit, die von Cython bei dem Übersetzen des Codes durchgeführte Analyse des Codes zu nutzen. Diese Option zeigt dem Entwickler durch Hervorheben von langsamen Python-Operationen, an welcher Stelle es lohnt, den Code zu optimieren (Abbildung 17). Zusätzlich gibt Cython für jede Zeile an, welche Funktionen aufgerufen werden, um die Optimierung zu erleichtern. Wichtig ist hierbei, dass die rechenintensiven Zeilen möglichst wenige Python-Operationen beinhalten.

```
117: def free_mem():
118:     ''' Initialisierung alle Variablen '''
119:     cdef int numberOfBlades = 4
120:     cdef int dimensionInRadialDirection = 12
121:     cdef int dimensionInAzimutalDirection = 36
122:     cdef int numberOfTurns = 4
123:     cdef int dimensionInAzimualDirectionTotal = dimensionInAzimutalDirection * numberOfTurns
124:     cdef float dt = 1.0 / dimensionInAzimutalDirection
125:     cdef int nTime = 10
126:     cdef double [:,:,:,:] x, vx
127:     cdef double [:,:,:] transversalVorticity, transversalVortexLength, longitudinalVorticity, longitudinalVortexLength
128:     cdef Py_ssize_t i1, i2, i3, iDir
129:
130:     ''' Erstellen des Rotors '''
131:     x, vx, transversalVorticity, transversalVortexLength, longitudinalVorticity, longitudinalVortexLength = initialize_mem(numberOfBlades, dimensionInRad
132:     print type(vx[0,1,1,1])
133:     #print type(vx)
134:     startSimulationTime = time.time()
135:     ''' Berechnung '''
136:     vx = calculate2_mem(numberOfBlades, dimensionInRadialDirection, dimensionInAzimualDirectionTotal, vx, x, transversalVorticity, transversalVortexLengt
137:
138:
139:     with nogil:
140:         for iDir in prange(3):
141:             for i1 in xrange(numberOfBlades):
142:                 for i2 in xrange(dimensionInRadialDirection):
143:                     for i3 in xrange(dimensionInAzimualDirectionTotal):
144:                         x[iDir,i1,i2,i3] = x[iDir,i1,i2,i3] + dt * vx[iDir,i1,i2,i3]
```

Abbildung 17: Cython-Annotierung

4.2.2 Perf top

Das Tool *perf top* zeigt die Systemaufrufe aller CPUs live an [Per14]. Abbildung 18 zeigt einen Ausschnitt von *perf top*, während die Numpy-Implementierung ausgeführt wird.

Damit können Informationen über häufig verwendete Funktionen gewonnen werden. In diesem Beispiel ist erkennbar, dass 17,9 % der CPU für die Python-Implementierung von Multiplikationen mit DP-Datentypen verwendet werden. Besonders Probleme durch häufige Speicherallokierung können mit diesem Tool am Aufruf der Funktion *_int_malloc* gut erkannt werden.

```
PerfTop:    1028 irqs/sec  kernel:12.6%  exact:  0.0% [1000Hz cycles],  (all, 12 CPUs)
-------------------------------------------------------------------------------------------

             samples  pcnt function                                    DSO
             _______ _____ ____________________________________________ ___________________

            1400.00 17.9% DOUBLE_multiply                              umath.so
            1255.00 16.1% npy_sqrt                                     umath.so
             836.00 10.7% DOUBLE_add                                   umath.so
             795.00 10.2% DOUBLE_subtract                              umath.so
             782.00 10.0% DOUBLE_divide                                umath.so
             334.00  4.3% DOUBLE_square                                umath.so
             155.00  2.0% page_fault                                   [kernel.kallsyms]
             147.00  1.9% clear_page_c                                 [kernel.kallsyms]
             122.00  1.6% intel_idle                                   [kernel.kallsyms]
              93.00  1.2% PyUFunc_d_d                                  umath.so
              92.00  1.2% PyEval_EvalFrameEx                           libpython2.7.so.1.0
              90.00  1.2% _int_malloc                                  libc-2.11.3.so
              74.00  0.9% __alloc_pages_nodemask                       [kernel.kallsyms]
              69.00  0.9% PyUFunc_GenericFunction                      umath.so
              65.00  0.8% feclearexcept                                libm-2.5.so
              62.00  0.8% PyArray_NewFromDescr                         multiarray.so
              41.00  0.5% get_page_from_freelist                       [kernel.kallsyms]
              40.00  0.5% __mem_cgroup_commit_charge                   [kernel.kallsyms]
              36.00  0.5% ufunc_generic_call                           umath.so
              36.00  0.5% PyUFunc_DefaultLegacyInnerLoopSelector       umath.so
              30.00  0.4% PyArray_ResultType                           multiarray.so
              27.00  0.3% get_ufunc_arguments                         umath.so
              27.00  0.3% tupledealloc                                 libpython2.7.so.1.0
              26.00  0.3% __GI___libc_free                            libc-2.11.3.so
              25.00  0.3% release_pages                                [kernel.kallsyms]
              24.00  0.3% alloc_pages_vma                              [kernel.kallsyms]
              23.00  0.3% PyArray_EquivTypes                           multiarray.so
              22.00  0.3% __GI___libc_malloc                          libc-2.11.3.so
              21.00  0.3% ____pagevec_lru_add_fn                       [kernel.kallsyms]
              21.00  0.3% __mem_cgroup_uncharge_common                 [kernel.kallsyms]
```

Abbildung 18: Live-Analyse der Numpy-Implementierung mit perf top

4.2.3 Likwid-Perfctr

Das Likwid-Projekt bietet eine Sammlung von Kommandozeilen-Tools für Linux zur Unterstützung der Entwickler von **H**igh **P**erformance **C**omputing (HPC)-Programmen [lik14a]. Das Likwid-Perfctr-Tool liest die Hardware-Performanz-Zähler der CPU aus. Dadurch wird dem Entwickler ermöglicht, Informationen über die Auslastung der Hardware zu bekommen. In Abbildung 19 ist ein Ausschnitt der Ausgabe von Likwid-Perfctr für die Numpy-Implementierung dargestellt. Die ersten drei Zeilen geben an, wie viele Takte die CPU zur Berechnung benötigt. In den zwei darauf folgenden

Event	core 0
INSTR_RETIRED_ANY	3.26334e+10
CPU_CLK_UNHALTED_CORE	3.09178e+10
CPU_CLK_UNHALTED_REF	2.65027e+10
FP_COMP_OPS_EXE_SSE_FP_PACKED	3.02023e+09
FP_COMP_OPS_EXE_SSE_FP_SCALAR	113307
FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION	0
FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION	3.02035e+09
UNC_QMC_NORMAL_READS_ANY	1.40366e+06
UNC_QMC_WRITES_FULL_ANY	119339
UNC_QHL_REQUESTS_REMOTE_READS	0
UNC_QHL_REQUESTS_LOCAL_READS	383133
UNC_QHL_REQUESTS_REMOTE_WRITES	0

Abbildung 19: Ausschnitt aus dem Profiling-Ergebnis von Likwid-Perfctr

Zeilen wird angegeben, wie viele SIMD-Operationen und wie viele skalare Operationen ausgeführt werden. Aus den nächsten zwei Zeilen ist erkennbar, dass ausschließlich mit Gleitkomma-Operationen mit doppelter Genauigkeit gerechnet wird. Weiterhin kann man mit Likwid auch die übertragene Datenmenge vom Hauptspeicher und das Cache-Verhalten betrachten. Um *Likwid* mit Python zu verwenden, kann man einen auf *ctypes* basierenden Wrapper verwenden. Ein solcher ist in Quellcode 13 abgebildet.

```
1 import ctypes
2 likwid = ctypes.cdll.LoadLibrary('liblikwid.so')
3 likwid.likwid_markerStartRegion.argtypes = (ctypes.c_char_p,)
4 likwid.likwid_markerStopRegion.argtypes = (ctypes.c_char_p,)
5 likwid.likwid_markerInit()
6 likwid.likwid_markerStartRegion('init')
7 # Zu analysierender Code
8 likwid.likwid_markerStopRegion('init')
9 likwid.likwid_markerClose()
```

Listing 13: Likwid-Wrapper, um Python zu analysieren

4.3 Performanz-Analyse des Freewake-Algorithmus

In diesem Abschnitt wird die theoretisch zu erwartende Laufzeit des Freewake-Algorithmus bestimmt. Dazu wird das Roofline-Performanz-Modell verwendet. Dafür wird zuerst die Balance des Testrechners und des Algorithmus berechnet. Basierend darauf wird die theoretisch minimale Laufzeit des Algorithmus auf CPU und GPU berechnet.

4.3.1 Maschinen-Balance

Die Maschinen-Balance bezieht sich auf einen Prozessor. Daher muss eine Maschinen-Balance für die CPU und eine für die GPU bestimmt werden. Die Implementierungen rechnen alle mit doppelter Genauigkeit, daher wurde für die Berechnung der Balance auf die DP-Peak-Performanz zurückgegriffen.

CPU-Balance

Die zum Testen verwendete Workstation mit Intel Xeon E5645 Prozessor besitzt eine gemessene maximale Speicherbandbreite von 19 GB/s und eine Peak-Performance von 28,8 GFlops:

$$B_m = \frac{19\,[\mathrm{GByte/s}]}{28,8\,[\mathrm{GFlop/s}]} = 0,6597. \tag{4.1}$$

Jeder Code, der eine Balance von mehr als 0,6597 besitzt, ist auf dieser CPU bandbreitenbegrenzt. Dies bedeutet, dass die Daten nicht schnell genug an die CPU übertragen werden können, um ständig Berechnungen durchzuführen.

GPU-Balance

In dem Testsystem ist eine Nvidia Tesla C2075 GPGPU mit einer Gleitkommaleistung von 515 GFlops und einer Speicherbandbreite von 144 GB/s verbaut [Tes14]:

$$B_m = \frac{144\,[\mathrm{GByte/s}]}{515\,[\mathrm{GFlop/s}]} = 0,2796. \tag{4.2}$$

4.3.2 Performanz-Modellierung des Freewake-Algorithmus

In dem Projekt Freewake werden die Daten hauptsächlich in 6 Arrays gespeichert. Zwei dieser Arrays haben in der getesteten Variante eine Größe von $3 \cdot 4 \cdot 12 \cdot 144$ und die restlichen vier von $4 \cdot 12 \cdot 144$ Elementen. Die Größe der anderen Daten ist so gering, dass sie für die folgende Betrachtung nicht relevant sind.

$$2 \cdot (3 \cdot 4 \cdot 12 \cdot 144) \cdot 8\,\mathrm{Bytes} + 4 \cdot (4 \cdot 12 \cdot 144) \cdot 8\,\mathrm{Bytes} = 552960\,\mathrm{Bytes} \tag{4.3}$$

Insgesamt werden demnach ungefähr 550 kByte an Daten verwendet.

Die Funktion *wilin* besteht aus circa 70 Operationen, die bei jedem Schleifendurchlauf ausgeführt werden. Die Anzahl der Schleifendurchläufe (nSchleifen) ist abhängig von vier Variablen. Die Variablen sind numberOfBlades (nB), dimensionInRadialDirection (dR), dimensionInAzimutalDirection (dA) und numberOfTurns (nT). Die allgemeine Gleichung ist in Gleichung 4.4 dargestellt.

$$\mathrm{nSchleifen} = \mathrm{nB}^2 \cdot \mathrm{dR} \cdot \mathrm{dA} \cdot \mathrm{nT} \cdot ((\mathrm{dR} - 1) \cdot \mathrm{dA} \cdot \mathrm{nT} + \mathrm{dR} \cdot (\mathrm{dA} \cdot \mathrm{nT} - 1)) \tag{4.4}$$

$$\mathrm{nSchleifen} = \mathrm{nB}^2 \cdot \mathrm{dR}^2 \cdot \mathrm{dA}^2 \cdot \mathrm{nT}^2 \cdot \left(\frac{\mathrm{dR} - 1}{\mathrm{dR}} + \frac{\mathrm{dA} \cdot \mathrm{nT} - 1}{\mathrm{dA} \cdot \mathrm{nT}} \right) \tag{4.5}$$

Durch Einsetzen der verwendeten Werte in die Variablen ergibt sich wie in Gleichung 4.6 dargestellt, dass die Schleife 91238400 mal durchlaufen wird.

$$4^2 \cdot 12 \cdot 36 \cdot 4 \cdot \Big((12-1) \cdot 36 \cdot 4 + 12 \cdot (36 \cdot 4 - 1)\Big) = 91238400 \tag{4.6}$$

Demnach wird auch die Funktion 91238400 mal aufgerufen. Aus der Multiplikation der Schleifenaufrufe mit der Anzahl der Operationen der Funktion ergibt sich, dass insgesamt etwa 6,5 Milliarden Operationen ausgeführt werden.

Damit lässt sich die in Rechnung 4.7 ermittelte Code-Balance für den Freewake-Code bestimmen:

$$B_c = \frac{550.000[Bytes]}{6.500.000.000[Flops]} = 0,00008 \frac{Bytes}{Flop}. \tag{4.7}$$

Durch Einsetzen der berechneten Balance für Maschine (Gleichung 4.1 oder 4.2) und Code (Gleichung 4.7) in Formel 2.6 ergibt sich für die CPU die Gleichung 4.8 und für die GPU die Gleichung 4.9.

$$l_{CPU} = \min\left(1, \frac{0,6597}{0,00008}\right) = 1 \tag{4.8}$$

$$l_{GPU} = \min\left(1, \frac{0,2796}{0,00008}\right) = 1 \tag{4.9}$$

Dieser Code kann also die Peak-Performanz der CPU und der GPU ausnutzen, da er weder durch die Speicherbandbreite auf der CPU noch auf der GPU begrenzt ist.

Aus der Anzahl an Operationen und der Peak-Performance der CPU ergibt sich eine minimale Laufzeit des Programms von 0,23 s auf der CPU (Gleichung 4.10) und 0,0126 s auf der GPU (Gleichung 4.11).

$$T_{CPU} = \frac{6.500.000.000}{28.800.000.000 \, \frac{1}{s}} = 0,23s \tag{4.10}$$

Implementierung	Zeit
Python	638s
Numpy	12,99s
Numba	22,62s
Cython	5,50s
Fortran	2,38s

Tabelle 1: Laufzeit der Single-Core-Implementierungen

$$T_{GPU} = \frac{6.500.000.000}{515.000.000.000\,\frac{1}{s}} = 0,0126s \tag{4.11}$$

Bei einer seriellen Implementierung reduziert sich die Leistung der CPU auf ein Sechstel, wodurch sich die minimale serielle Laufzeit von 0,23 s auf 1,35 s erhöht.

$$T_{CPUseriell} = \frac{6.500.000.000}{4.800.000.000\,\frac{1}{s}} = 1,35s \tag{4.12}$$

4.4 Performanz-Analyse und Vergleich der Implementierungen

In diesem Kapitel wird die Performanz der entwickelten Python-Implementierungen mit den Referenzimplementierungen in Fortran verglichen. Zuerst werden die Single-Core-Implementierungen verglichen, bevor dann auf die Multi-Core-Implementierungen und zum Schluss auf die GPU-Implementierung eingegangen wird.

4.4.1 Single-Core

Python benötigt ohne Zusatzbibliotheken für die Durchführung des Freewake-Algorithmus 638 s. Numpy beschleunigt die Ausführung durch die im Hintergrund arbeitenden C-Routinen auf 12,99 s. Ein Kompilieren des Codes durch Numba mit dem LLVM-Compiler

erzeugt hierbei einen Overhead, der den Code 22,62 s benötigen lässt. Die Cython-Implementierung ist als schnellste Python-Single-Core-Implementierung mit 5,5 s circa halb so schnell wie Fortran mit 2,38 s (siehe Tabelle 1). In Abbildung 20 ist zu erken-

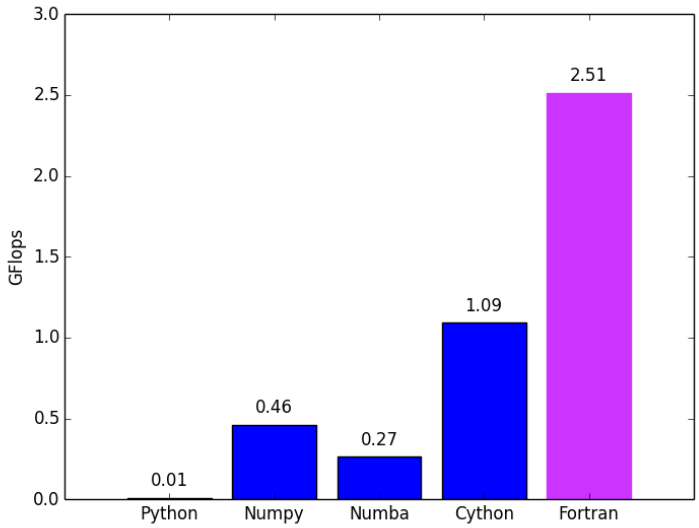


Abbildung 20: GFLOPS der seriellen Implementierungen

nen, dass Fortran die 2,3-fache Performanz der schnellsten Python-Implementierung in Cython erreicht. Bei den anderen Python-Implementierungen ist der Quotient aus der Performanz von Fortran und der Performanz dieser Python-Implementierungen größer. Bei der zweit-schnellsten Implementierung in Numpy liegt dieser Quotient bei 5,4. Auch die serielle Fortran-Implementierung kommt nicht an die maximale Leistung der CPU heran, sondern erreicht mit 2,51 GFLOPS 52% der seriellen Peak-Performance. Abgeschlagen liegt Python mit 10 MFLOPS an letzter Stelle.

Fortran erreicht nicht die vorher berechnete Zeit, da bei der ausgewählten Performanz-Modellierung davon ausgegangen wird, dass nur Multiplikationen und Additionen zusammen ausgeführt werden. In dem Freewake-Algorithmus sind allerdings auch Wurzeloperationen und Divisionen vorhanden. Um eine bessere Vorhersage der Laufzeit zu erzielen, ist daher ein genaueres Modell nötig. Cython benötigt etwa die doppelte

Implementierung	Zeit	Speed-Up
Cython	1,03s	5,3
Global Arrays	4,34s	3,7
Fortran	0,440s	5,4

Tabelle 2: Ergebnisse der Multi-Core-Implementierung

Laufzeit von Fortran. Eine Analyse mit *Likwid-perfctr* zeigt bei dieser Implementierung, dass keine SIMD-Operationen verwendet werden, was den Faktor 2 erklärt. Die Numpy-Implementierung benutzt, wie in Abbildung 19 dargestellt, SIMD-Operationen. Allerdings werden bei dieser Implementierung temporäre Arrays beim Aufruf der *wilin*-Funktion erstellt. Dadurch erhöht sich die Datenmenge signifikant, was nahelegt, dass hier der Performanz-Verlust aufgrund schlechteren Cache-Verhalten auftritt. Die *JIT*-Konvertierung von Numba ist eine Black-Box, bei welcher der Entwickler nicht verfolgen kann, welche Operationen ausgeführt werden. *Perf top* zeigt hierbei keine hilfreichen Informationen an. Eine Analyse mit *Likwid-perfctr* zeigt, dass neben den SIMD-Operationen auch einige skalare-Operationen ausgeführt werden, wodurch sich allerdings nur ein Teil des Performanz-Verlusts erklären lässt. Die Python-Implementierung verwendet dynamische Datentypen, die die Performanz drastisch reduzieren.

4.4.2 Multi-Core

Auf dem getesteten Multi-Core-System benötigt die mit OpenMP parallelisierte Cython-Implementierung 1,03 s. Die auf Global Arrays aufbauende Implementierung benötigt mit 4,34 s die vierfache Laufzeit. Die Fortran-Implementierung ist mit 0,44 s schneller als die beiden Python-Implementierungen. Die parallele Fortran-Implementierung erreicht, wie die serielle Implementierung, mit 13,64 GFLOPS ungefähr 50% der Peak-Performance (Abbildung 21). Die OpenMP-Parallelisierung sorgt in Cython für einen Anstieg der Performanz von 1,09 GFLOPS auf 5,78 GFLOPS. Die Global-Arrays-Version erreicht

eine geringere Performanz von 1,38 GFLOPS, behält aber die klare und übersichtliche Syntax von Python bei. In der dritten Spalte von Tabelle 2 wird der Speed-Up der

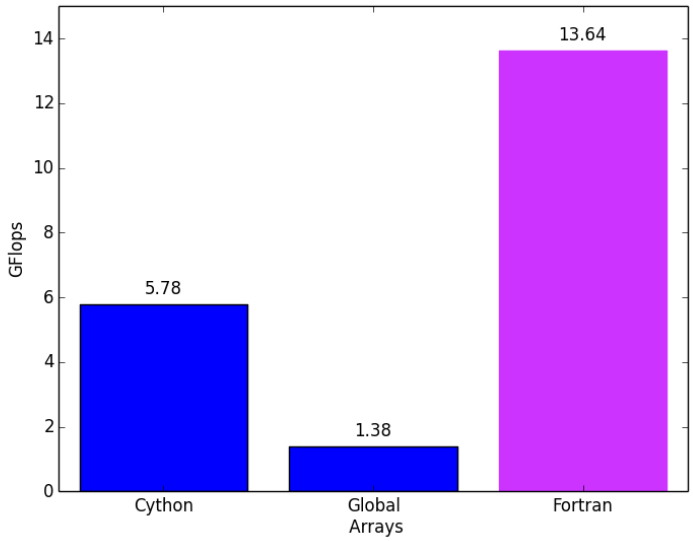


Abbildung 21: GFLOPS der parallelen Implementierungen

Implementierungen mit sechs Cores gegenüber der Implementierung mit einem Core aufgelistet. Der Speed-Up der Cython-Implementierung liegt mit 5,3 sehr nah an dem durch eine Leistungserhöhung um den Faktor sechs liegenden Maximum. Ebenso gut skaliert die Fortran-Implementierung mit einem Faktor von 5,4. Die Implementierung mit Global Arrays benötigt mit einem Core 16,4 s und skaliert mit einem Faktor von 3,7 nicht so gut wie die anderen zwei. Dies liegt vermutlich am Duplizieren der Daten und der Kommunikation zwischen lokalem Array und globalem Array.

Implementierung	Zeit
Numba	0,770s
Fortran + OpenACC	0,086s

Tabelle 3: Ergebnisse der GPU-Implementierungen

4.4.3 GPU

Aus der Multiplikation von numberOfBlades (nB), dimensionInRadialDirection (dR), dimensionInAzimutalDirection (dA) und numberOfTurns (nT) ergibt sich die maximale Anzahl von unabhängigen Threads (Formel 4.13).

$$\mathrm{maxThreads} = \mathrm{nB} \cdot \mathrm{dR} \cdot \mathrm{dA} \cdot \mathrm{nT} \tag{4.13}$$

$$\mathrm{maxThreads} = 4 \cdot 12 \cdot 36 \cdot 4 = 6912 \tag{4.14}$$

Jeder dieser Threads berechnet die Summe der induzierten Geschwindigkeiten von allen Wirbelsegmenten für einen Gitterpunkt. Dies ermöglicht es, jeden dieser Threads in weitere circa 6000 Threads aufzuteilen und die Anteile von allen Wirbelsegmenten dann erst am Ende aufzusummieren. Das Ausführen einer arithmetischen Operation benötigt 22 Taktzyklen [Cud14b]. Um diese Latenz zu vermeiden, sollte zu jedem Taktzyklus eine Operation zum Ausführen vorhanden sein. Bei 488 Cores ergibt sich eine geforderte Anzahl an Threads von ungefähr 10.000 Threads. Durch das Aufteilen der Threads erhöht sich die maximale Anzahl an Threads auf ungefähr 6000^2, wodurch das Minimum an Threads zur Vermeidung der Latenzen erreicht wird. Dieses Aufsummieren macht Numba nicht automatisch. Daher erfordert es ein deutliches Umstrukturieren des Codes. Dies passt nicht nicht zu dem Ziel einer einfachen Implementierung passt. Daher wurde in Numba mit den in Gleichung 4.14 berechneten 6912 unabhängigen Threads gerechnet. Insgesamt ist das Problem zu klein, um die GPU vollständig auszulasten.

Auf der GPU kann durch die hohe Anzahl an Threads eine bessere Performanz als auf der CPU erreicht werden. Die Numba-Implementierung erreicht mit 7,79 GFLOPS eine Zeit von 0,77 s. Die Fortran-Implementierung mit OpenACC [Ope14a] benötigt 0,086 s und erzielt 69,77 GFLOPS (siehe Tabelle 3 und Abbildung 22).

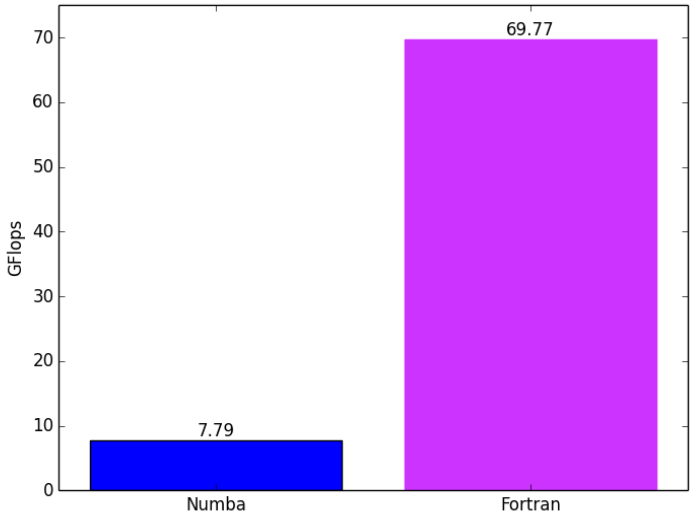


Abbildung 22: GFLOPS der GPU-Implementierungen

5 Fazit und Ausblick

In diesem Kapitel wird das Fazit der Arbeit vorgestellt. Abgeschlossen wird das Kapitel mit einem Ausblick auf mögliche Entwicklungen in der Zukunft.

5.1 Fazit

Diese Arbeit hat bestätigt, dass Python ohne Zusatzbibliotheken deutlich langsamer ist als hardware-nahe Programmiersprachen wie C oder Fortran. Bei dem implementierten Algorithmus erreicht Python ein Dreihundertstel der Performanz von Fortran. Bei Single-Core-Implementierungen erhöht die Numpy-Bibliothek durch im Hintergrund arbeitende C-Routinen die Performanz von Python auf ein Sechstel von Fortran. Schneller als Numpy ist abgesehen von Fortran nur die Cython-Implementierung. Diese erzeugt C-Funktionen, besitzt jedoch nicht mehr die übersichtlichen Syntax von Python.

Auf Multi-Core-Systemen ist die Fortran-Variante wie auf Single-Core-Systemen am schnellsten. Auch mit Cython kann man Anwendungen parallel auf gemeinsamen-Speicher-Systemen ausführen und erreicht hier ungefähr die halbe Performanz von Fortran. Die einzige Implementierung, die eine übersichtliche Python-Syntax besitzt, verwendet *Global Arrays* und erreicht ein Zehntel der Performanz von Fortran.

Die Numba-Implementierung auf der GPU ist schneller als alle anderen Implementierungen auf der CPU mit Python. Dabei erreicht diese Implementierung nicht das Maximum der GPU-Performanz. Fortran ist mit OpenACC auf der GPU fast zehnmal schneller.

Mit Rückblick auf die Ausgangshypothese, nach der es mit der Programmiersprache Python und dazu erhältlichen Bibliotheken möglich ist, eine ähnliche Performanz wie mit C oder Fortran bei komplexen Anwendungskernels zu erreichen, lässt sich sagen, dass Python je nach System zwischen der Hälfte und einem Zehntel der Performanz erreicht. Derartige komplexe Anwendungskernels werden in der Regel auf Multi-Core-Systemen ausgeführt. Hierbei erreicht die Implementierung mit *Global Arrays* ein Zehntel der Performanz von Fortran, überzeugt allerdings durch eine besonders schöne Implementierung. Bei Anwendungen, die eine recht kurze Laufzeit besitzen und bei denen ein Performanzverlust von Zehn keinen zu großen Zeitverlust bedeutet, ist eine solche Implementierung zu empfehlen. Bei zeitkritischen Anwendungen oder Anwendungen, die mehrere Tage laufen, rechtfertigt eine schnellere und einfachere Implementierung den Zeitverlust bei der Programmausführung nicht. Die Verwendung der GPU beschleunigt den Code deutlich. Hierbei zeichnet sich die Numba-Implementierung wie die Global-Arrays-Implementierung bei Multi-Core-Systemen durch einen einfachen, kompakten und übersichtlichen Programmierstil aus. Allerdings ist zu beachten, dass der Entwickler Kenntnisse über die Programmierung der GPU besitzen muss. Eine Portierung der Multi-Core-Implementierung auf die GPU ist hier nicht ohne weiteres möglich.

5.2 Ausblick

Während es unwahrscheinlich erscheint, dass Python ohne Zusatzbibliotheken in Zukunft eine bessere Performanz erreichen wird, bieten Zusatzbibliotheken einen deutlichen Performanz-Gewinn. Die Community hinter den Numpy-, Numba-, Cython- und Global-Array-Projekten arbeiten aktiv an der Weiterentwicklung, wodurch sich die Performanz weiter erhöhen wird. Ein Erreichen oder gar ein Überbieten der Performanz einer Implementierung mit C oder Fortran wird trotzdem kaum möglich sein. Es ist allerdings vorstellbar, dass die Performanz von Python nahe genug an die von Fortran heran

kommen könnte, um einen generellen Einsatz im HPC-Bereich zu rechtfertigen, da die Entwicklung eines Programms in Python weniger Zeit benötigt als in Fortran und einen sowohl übersichtlicheren als auch besser wartbaren Code produziert. Alternativ können Performanz-kritische Funktionen auch in C oder Fortran geschrieben und von Python aufgerufen werden.

Literaturverzeichnis

[Ada14] Adams, N. A.: *Wirbelströmungen - Gesetz von Biot-Savart*. http://www.aer.mw.tum.de/fileadmin/tumwaer/www/pdf/lehre/fluidmechanik2/VL3_mit.pdf, 2014. – Abruf: 17. Juli 2014

[BRZH14] Basermann, Achim ; Röhrig-Zöllner, Melven ; Hoffmann, Johannes: *Porting a parallel rotor wake simulation to GPGPU accelerators using OpenACC*. http://www.t-systems-sfr.com/e/deu/abstract.2014_7.php, 2014. – Abruf: 17. Juli 2014

[BSSS87] Boxwell, D. A. ; Schmitz, F. H. ; Splettstoesser, W. R. ; Schultz, K. J.: Helicopter Model Rotor-Blade Vortex Interaction Impulsive Noise: Scalability and Parametric Variations. In: *Journal of the American Helicopter Society* 32 (1. Januar 1987), Nr. 1, 3-12. http://dx.doi.org/doi:10.4050/JAHS.32.3. – DOI doi:10.4050/JAHS.32.3

[CUD14a] *CUDA parallel computing*. http://www.nvidia.de/object/cuda-parallel-computing-de.html, 2014. – Abruf: 21. Juli 2014

[Cud14b] *Cuda Toolkit Documantation - Multiprocessor Level*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level, 2014. – Abruf: 16. September 2014

[Cyt14a] *The Cython compiler for writing C extensions for the Python language*. https://pypi.python.org/pypi/Cython/, 2014. – Abruf: 03. September 2014

[Cyt14b] *Cython Compiler-Direktiven*. http://docs.cython.org/src/reference/compilation.html#compiler-directives, 2014. – Abruf: 18. August 2014

[Cyt14c] *Using the Cython Compiler to write fast Python code*. http://www.behnel.de/cython200910/talk.html, 2014. – Abruf: 03. September 2014

[DLR14] *DLR Portal*. http://www.dlr.de/dlr/desktopdefault.aspx/tabid-10443/637_read-251/#/gallery/8570, 2014. – Abruf: 25. Juni 2014

[DSP+11] Daily, Jeff ; Saddayappan, P. ; Palmer, Bruce ; Manojkumar Krishnan, Sriram K. ; Vishnu, Abhinav ; Chavarría, Daniel ; Nichols, Patrick: *High Performance Computing in Python using NumPy and the Global Arrays Toolkit*. http://hpc.pnl.gov/globalarrays/tutorials/GA_SciPy2011_Tutorial.pdf. Version: 08 2011. – Remarks by Chairman Alan Greenspan at the Annual Dinner and Francis Boyer Lecture of The American Enterprise Institute for Public Policy Research, Washington, D.C. [Accessed: 2013 06 20]

[E5614] *Intel Xeon Processor E5645 Spezifikationen*. http://ark.intel.com/de/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI?q=e5645, 2014. – Abruf: 30. Juli 2014

[Fun14] *Funktionsweise von Numba*. http://on-demand.gputechconf.com/supercomputing/2013/presentation/SC3121-Programming-GPU-Python-Using-NumbaPro.pdf, 2014. – Abruf: 04. September 2014

[Guo14] Guo, Philip: *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities*. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-t fulltext, 2014. – Abruf: 22. Juli 2014

[HW10] Hager, G. ; Wellein, G.: *Introduction to High Performance Computing for Scientists and Engineers*. Taylor & Francis, 2010 (Chapman & Hall/CRC Computational Science). http://books.google.de/books?id=rkWPojgfeM8C. – ISBN 9781439811931

[Ill14] Illmer, Joachim: *Effziente Python-Programmierung auf Multi-Core-Architekturen und Grafikkarten für numerische Probleme aus der linearen Algebra*. 2014. – 28. April 2014

[Int14] *Intel Xeon Processor 5600 Series*. http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf, 2014. – Abruf: 30. Juli 2014

[Kep14] *Kepler Computing-Architektur-Whitepaper*. http://www.nvidia.de/object/nvidia-kepler-de.html, 2014. – Abruf: 21. Juli 2014

[lik14a] *likwid - Lightweight performance tools*. https://code.google.com/p/likwid/, 2014. – Abruf: 12. September 2014

[Lik14b] *LikwidBench Wiki*. https://code.google.com/p/likwid/wiki/LikwidBench, 2014. – Abruf: 18. Juli 2014

[Lut13] Lutz, M.: *Learning Python*. O'Reilly Media, 2013 (Safari Books Online). http://books.google.de/books?id=4pgQfXQvekcC. – ISBN 9781449355692

[Mat14] *Homepage von Matlab*. http://www.mathworks.de/products/matlab/, 2014. – Abruf: 22. Juli 2014

[MKL14] *Intel Math Kernel Library*. https://software.intel.com/en-us/intel-mkl, 2014. – Abruf: 28. Juli 2014

[MPI14] *The Message Passing Interface (MPI) standard*. http://www.mcs.anl.gov/research/projects/mpi/, 2014. – Abruf: 12. September 2014

[Num14a] *Homepage von Numba*. http://numba.pydata.org/, 2014. – Abruf: 03.

September 2014

[Num14b] *Ways to parallelize - Numba-Users Mailinglist*. https://groups.google.com/a/continuum.io/forum/#!topic/numba-users/UN4sDSr8Iew, 2014. – Abruf: 01. August 2014

[Num14c] *Numba Mailinglist*. https://groups.google.com/a/continuum.io/forum/#!topic/numba-users/iOnkSJTcF0A, 2014. – Abruf: 12. September 2014

[Num14d] *Homepage von Numpy*. http://www.numpy.org/, 2014. – Abruf: 22. Juli 2014

[Nvi14a] *Nvidia CUDA*. https://developer.nvidia.com/about-cuda, 2014. – Abruf: 19. August 2014

[Nvi14b] *Nvidia Tesla C2075 Companion Processor*. http://www.nvidia.de/content/PDF/data-sheet/NV_DS_Tesla_C2075_Sept11_US_HR.pdf, 2014. – Abruf: 12. September 2014

[Ope14a] *Homepage der OpenACC API*. http://www.openacc-standard.org/, 2014. – Abruf: 15. September 2014

[Ope14b] *OpenCL - The open standard for parallel programming of heterogeneous systems*. http://openmp.org/wp/, 2014. – Abruf: 12. September 2014

[Ope14c] *OpenMP Specification for parallel Programming*. http://openmp.org/wp/, 2014. – Abruf: 12. September 2014

[Par14] *Parallele Berechnungen mit CUDA*. http://www.nvidia.de/object/cuda-parallel-computing-de.html, 2014. – Abruf: 04. September 2014

[Per14] *Live analysis with perf top*. https://perf.wiki.kernel.org/index.php/Tutorial#Live_analysis_with_perf_top, 2014. – Abruf: 15. September 2014

[PyU14] *PyUnit - the standard unit testing framework for Python*. http://pyunit.sourceforge.net/, 2014. – Abruf: 05. September 2014

[SC 13] *Simulations- und Softwaretechnik (SC)*. http://www.dlr.de/sc/, 2013. – Abruf: 25. Juni 2014

[SKS+14] Splettstösser, W.R. ; Kube, R. ; Seelhorst, U. ; Wagner, W. ; Boutier, A. ; Micheli, F. ; Pengel, K.: *Higher harmonic Control Aeroacustic Rotor Test (HART) - Test Documentation and Representative Results*. http://www.t-systems-sfr.com/e/deu/abstract.2014_7.php, 2014. – Abruf: 17. Juli 2014

[Tes14] *Nvidia Tesla Koprozessoren*. http://www.nvidia.de/object/tesla-workstation-graphics-de.html, 2014. – Abruf: 18. Juli 2014

[Tio14] *TIOBE Index for July 2014*. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htmll, 2014. – Abruf: 22. Juli 2014

[Top14] *Top500 List - Juni 2014*. http://www.top500.org/list/2014/06/, 2014. – Abruf: 26. Juni 2014

[Tra14] *The abstraction-optimization tradeoff*. http://blog.vivekhaldar.com/post/12785508353/the-abstraction-optimization-tradeoff, 2014. – Abruf: 03. September 2014

[WWP09] Williams, Samuel ; Waterman, Andrew ; Patterson, David: Roofline: An Insightful Visual Performance Model for Multicore Architectures. In: *Commun. ACM* 52 (2009), April, Nr. 4, 65–76. http://dx.doi.org/10.1145/1498765.1498785. – DOI 10.1145/1498765.1498785. – ISSN 0001–0782

A Freewake in Fortran

Die vorhandene Freewake-Funktion in Fortran.

A.1 Spezifikation des Algorithmus

```
1  elemental subroutine wilin(dax,day,daz,dex,dey,dez,ga,ge,wl,vx,vy,vz)
2    !---------------------------------------------------------------------
3    real, intent(in) :: dax, day, daz
4    real, intent(in) :: dex, dey, dez
5    real, intent(in) :: ga, ge, wl
6    real, intent(inout) :: vx, vy, vz
7    !---------------------------------------------------------------------
8    real, parameter :: rcq = 0.1
9    real, parameter :: eps = 1.e-8
10   !---------------------------------------------------------------------
11   real :: daq, deq, da, de, dae, sqa, sqe, sq, rmq, fak
12   !---------------------------------------------------------------------
13   daq = dax**2+day**2+daz**2
14   deq = dex**2+dey**2+dez**2
15   da = sqrt(daq)
16   de = sqrt(deq)
17   dae = dax*dex+day*dey+daz*dez
18   sqa = daq - dae
19   sqe = deq - dae
20   sq = sqa + sqe
21   rmq = daq * deq - dae**2
22   fak = ( (da+de)*(da*de - dae) * (ga*sqe + ge*sqa) &
23     & + (ga-ge)*(da-de)*rmq ) / ( sq*(da*de+eps)*(rmq+rcq*sq) )
24
25   fak = fak * wl / sqrt( sq )
26   vx = vx + fak * ( day*dez - daz*dey )
27   vy = vy + fak * ( daz*dex - dax*dez )
28   vz = vz + fak * ( dax*dey - day*dex )
29 end subroutine wilin
```

Listing 14: Funktion zur Berechnung der vom Wirbel induzierten Geschwindigkeit in Fortran

```
!$ACC LOOP collapse(3) gang reduction(+:vx)
!$OMP DO collapse(3) reduction(+:vx)
    do ib = 1, nb
      do ir = 2, nr
        do ia = 1, nanu
          call wilin(x(:,:,:,1)-x(ia,ir,ib,1), x(:,:,:,2)-x(ia,ir,ib,2), &
            & x(:,:,:,3)-x(ia,ir,ib,3), x(:,:,:,1)-x(ia,ir-1,ib,1), &
            & x(:,:,:,2)-x(ia,ir-1,ib,2),x(:,:,:,3)-x(ia,ir-1,ib,3), &
            & gams(ia,ir,ib),gams(ia,ir-1,ib), wlongs(ia,nr,nb), &
            & vx(:,:,:,1),vx(:,:,:,2),vx(:,:,:,3))
        end do
      end do
    end do
!$ACC WAIT
!$ACC LOOP collapse(3) gang reduction(+:vx)
!$OMP DO collapse(3) reduction(+:vx)
    do ib = 1, nb
      do ir = 1, nr
        do ia = 2, nanu
          call wilin(x(:,:,:,1)-x(ia,ir,ib,1), x(:,:,:,2)-x(ia,ir,ib,2), &
            & x(:,:,:,3)-x(ia,ir,ib,3), x(:,:,:,1)-x(ia-1,ir,ib,1), &
            & x(:,:,:,2)-x(ia-1,ir,ib,2),x(:,:,:,3)-x(ia-1,ir,ib,3), &
            & gamt(ia,ir,ib),gams(ia-1,ir,ib), wlongt(ia,nr,nb), &
            & vx(:,:,:,1),vx(:,:,:,2),vx(:,:,:,3))
        end do
      end do
    end do
!$ACC WAIT
    do iDir = 1, 3
!$OMP WORKSHARE
      x(:,:,:,iDir) = x(:,:,:,iDir) + dt*vx(:,:,:,iDir)
!$OMP END WORKSHARE
```

Listing 15: Benchmark für Freewake in Fortran