INSTITUT FÜR INFORMATIK

# Simulation von biogeochemischen Prozessen in 3-D auf GPUs

Eike Siewertsen, Jaroslaw Piwonski, Thomas Slawig

Bericht Nr. 1204
May 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

# Simulation von biogeochemischen Prozessen in 3-D auf GPUs

Eike Siewertsen, Jaroslaw Piwonski, Thomas Slawig

Bericht Nr. 1204
May 2012
ISSN 2192-6247

e-mail: {esi,jpi,ts}@informatik.uni-kiel.de

Christian-Albrechts-Universität zu Kiel
Algorithmische Optimale Steuerung
Institut für Informatik

Bachelorarbeit Informatik

# Simulation von biogeochemischen Prozessen in 3-D auf GPUs

Eike Siewertsen

31. Mai 2012

**Erstgutachter:** Prof. Dr. Thomas Slawig

**Zweitgutachter:** Dipl.-Math. Jaroslaw Piwonski

## Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und in keinem anderen Prüfungsverfahren eingereicht habe.

Kiel,

_____

## Inhaltsverzeichnis

## Abbildungsverzeichnis

# 1 Einleitung

Der Themenbereich der Klimamodellierung spielt heutzutage eine wichtige Rolle im System der politischen Entscheidungsbildung: Der fortschreitende Klimawandel und die wissenschaftliche fundierte Bestimmung der Ursachen dieser führte dazu, dass Regierungen der Welt in Form der Vereinigten Nationen die Wichtigkeit einer nachhaltigen Entwicklung erkannten und sich für Lösungen interessierten. Aus diesem Grund wurde 1988 von den Vereinigten Nationen das „Intergovernmental Panel on Climate Change" (IPCC[1]) gegründet, welches als Aufgabe hatte, neben den Auswirkungen des Klimawandels auf die folgenden Jahrhunderte auch Möglichkeiten zur Eindämmung derselben zu finden. Da sich das Klima der Erde nur über lange Zeiträume (bis zu Jahrtausenden) ändert und die Auswirkungen von einzelnen Einflussfaktoren entweder noch nicht vollständig bekannt oder schwer bestimmbar sind, ist es nötig für eine zeitlich angemessene Analyse dieser Auswirkungen die Klimaeffekte zeitlich beschleunigt zu simulieren. Mittels Klimasimulationen werden Hypothesen in Form von Klimamodellen über lange Zeiträume simuliert, um zum einen ihre Übereinstimmung mit Messdaten zu überprüfen, zum anderen Klimaszenarien für die Zukunft zu bilden.

Ein wichtiger Teil dieser Simulation ist ihre Geschwindigkeit: Modelle müssen viele Male für bis zu mehrere tausend Jahre simuliert werden, während entweder manuell oder automatisch ihre Steuerparameter angepasst werden, bis die Simulationswerte der Realität am nähesten kommen. Deshalb ist es ein Bestreben der Wissenschaftler, diese Simulationen immer schneller und genauer durchzuführen. Da die Anzahl der Transistoren (und damit die Taktrate) von *einzelnen Kernen* der CPUs seit einigen Jahren nicht mehr nach Moore's Law zunehmen, dringen mehr und mehr Technologien und Ideen hervor welche darauf abzielen mehrere CPUs gleichzeitig zu verwenden - also die Berechnung zu parallelisieren. Da mit einer steigenden Anzahl von CPUs aber auch die Menge an Kommunikation zwischen den Komponenten zunimmt, nimmt der Geschwindigkeitsgewinn pro Einheit ab einer gewissen Anzahl an CPUs ab - es wird also eine Sättigung der Kommunikationsinfrastruktur erreicht, wodurch die Kosten zur Beschleunigung immer weiter steigen.

An dieser Stelle versucht diese Arbeit einzusteigen: Mittels eines Hardwaresystems - der Grafikkarte - welches für eine stark parallelisierte Berechnung spezialisiert ist und eine hohe Kommunikationsbandbreite zwischen den Rechenkomponenten bietet, wird ein bisher für die Berechnung auf CPU-Clustern ausgelegtes Softwaresystem konvertiert. Es sollen an-

[1] http://www.ipcc.ch

fangs grundlegende Techniken und teilweise frei verfügbare Bibliotheken vorgestellt werden, die den Entwickler dabei unterstützen existierenden Quellcode für die GPU anzupassen, auszuführen und Fehler in diesem zu finden und zu beheben. Es werden kurz die nötigen Änderungen am modifizierten System beschrieben und auf Besonderheiten eingegangen, und zum Schluss werden die Geschwindigkeitsgewinne über mehrere Simulationen hinweg analysiert und mit ähnlichen Simulationen auf einzelnen CPUs wie auch CPU-Clustern verglichen.

# 2 Grundlagen

Folgend soll eine kurze Übersicht über die in der Arbeit vorkommenden und verwendeten Technologien und Bibliotheken gegeben werden. Weiterführende Referenzen werden soweit verfügbar angegeben, da die Themen nicht erschöpfend erläutert werden können. Weiterhin soll darauf geachtet werden, dass sehr komplexe Technologien wie CUDA nicht in allen Details erläutert werden, da die Techniken zur Optimierung wie auf dem PC auch bis hinunter zur Manipulation von Maschinencode reichen. Stattdessen soll darauf abgezielt werden existierende Programmsysteme mit möglichst geringen Aufwand die Eigenschaften einer Grafikkarte ausnutzen zu lassen. Dazu sollen bereits existierende Bibliotheken wie CUSP und thrust beschrieben und verständlich gemacht werden.

## 2.1 CUDA

CUDA, kurz für „Compute Unified Device Architecture", ist NVIDIAs parallele Architektur zur Ausführung von rechenintensiven Programmcode auf der GPU. Durch die Ausnutzung der Architektur von Grafikkarten, sowie der erhöhten Speicherbandbreite ist es möglich eine weit höhere Anzahl von Fließpunktoperationen pro Sekunde (FLOPS) als auf CPUs zu erreichen. Während CPUs etwa ein bis acht Kerne mit jeweils maximal 4 GHz Taktrate haben, haben GPUs zwar eine geringere Taktrate, aber dafür mehrere hundert Kerne welche wiederum mehrere Threads gleichzeitig ausführen können. Der Grund für diesen Unterschied zwischen den Architekturen ist, dass die GPU, welche ihren Ursprung in der Computergrafik hat, auf rechenintensive aber stark parallel arbeitende Prozesse spezialisiert ist. Für die 3D-Computergrafik ist es nötig für jeden Pixelpunkt auf dem Bildschirm eine Reihe von Matrix-/Vektoroperationen zusätzlich zu weiteren Vertex- oder Pixeloperationen wie Licht- und Schattenberechnung durchzuführen. Dadurch, dass sich diese Operationen für jeden Pixel nur in den Parametern unterscheiden, der Programmcode aber der gleiche bleibt, hat sich die GPU zu der parallelen Architektur, die sie heute ist entwickelt.

Bevor NVIDIA CUDA entwickelte gab es nur klassische GPU APIs wie OpenGL und DirectX. Forscher mit Interessen in komplexen und zeitaufwendigen Prozessen waren dennoch daran interessiert ihre Berechnungen durch eine kostengünstige Parallelarchitektur zu beschleunigen. Deshalb war es nötig die existierenden Probleme auf Probleme der Grafikprogrammierung zu reduzieren. Da die GPU Farbwerte auf dem Bildschirm berechnet, mussten die Eingaben und Berechnungen so angepasst werden, dass die ausgegebenen Farbwerte als solche Daten interepretiert werden konnten. Während erste Ergebnisse in

die Richtung von bedeutenden Geschwindigkeitsgewinnen zeigten, führte das umständliche Programmiermodell dazu, dass sich nur wenige Gruppen damit beschäftigt haben.

Mit der Ankündigung der CUDA Architektur im November 2006 wurden die NVIDIA GPUs ab der GeForce 8800 GTX für die allgemeine Berechnung geöffnet und der Term der „General Purpose Graphics Processing Unit" (GPGPU) geformt. Es war nun möglich die Recheneinheiten (*arithmetic logic unit*, ALU) der GPU vollständig von Software zu kontrollieren. Weiterhin wurden wichtige Standards wie der IEEE Standard zu Fließkommazahlarithmetik eingehalten und zufällige Speicherzugriffe auf lokalem und globalem Speicher erlaubt.

Folgend sollen einige der Grundkonzepte des CUDA Programmiermodells im Vergleich zum typischen CPU Modell erläutert werden.

### Kernel

Der Grundbaustein der CUDA Architektur sind sogenannte CUDA Kernel. Ein Kernel ist ein Stück Programmcode der auf der Grafikkarte tausendfach in parallelen GPU Threads ausgeführt wird. Adressiert werden die einzelnen Threads über ein Gitternetz (*grid*) und Blöcke (Abbildung 2.1). Ein Kernelaufruft beinhaltet als Parameter eine Gitternetzgröße in drei Dimensionen, wobei jede Zelle des Gitternetzes aus einem Block besteht. Ein Block besteht aus Threads und seine Größe ist durch eine bis zu dreidimensionale Größenangabe festgelegt. Über das CUDA Sprachenkonstrukt `kernel<<<gridSize, blockSize>>>()` werden die Blöcke aus Threads erstellt und parallel von der GPU ausgeführt. Die Variable `threadIdx` gibt innerhalb des Kernels den Index des derzeit laufenden Threads innerhalb des Blockes an, den derzeitigen Block kennzeichnet die Variable `blockIdx`. Die maximale Anzahl an Threads entspricht also der Blockgröße mal der Anzahl von Blöcken.

Die Grafikkarte besteht dabei aus mehreren sogenannten *Streaming Multiprocessors* (SM) welche weiterhin jeweils aus mehreren CUDA Kernen bestehen. Jeder Multiprozessor besitzt seinen eigenen Zwischenspeicher, Register und eine Reihe von Kernen. Die Kerne besitzen ihre eigenen Rechenwerke für Ganzzahl- und Fließkommarechnung. Beispielsweise besitzt die GeForce GTX 480[2] 15 (die Architektur unterstützt bis zu sechzehn) Multiprozesoren mit jeweils 32 CUDA Kernen, insgesamt als 480 CUDA Kerne. Auf einem Kern ist die kleinste ausführbare Einheit ein sogenannter Warp, welcher aus 32 Threads be-

[2] http://www.geforce.com/Hardware/GPUs/geforce-gtx-480/architecture
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

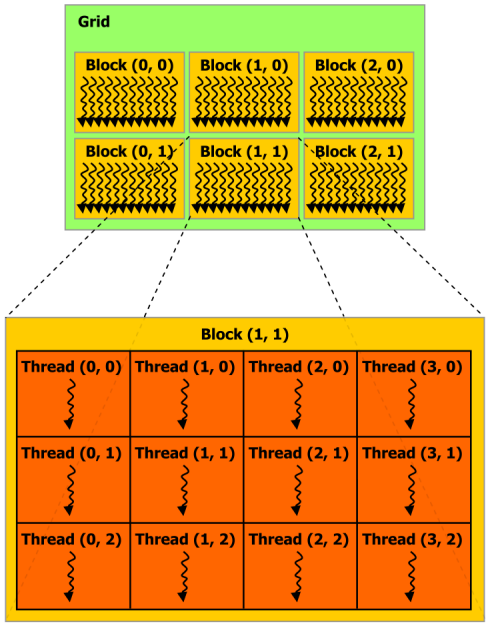


Fig. 2.1: CUDA Gitternetz und Blöcke [NVIDIA, 2011b, S. 9]

steht. Es ist weiterhin üblich das mehrere Warps eines Blockes gleichzeitig auf einem Kern ausgeführt werden, dann aber nebenläufig und nicht echt parallel. Wie viele Threads von einem Multiprozessor gleichzeitig ausgeführt werden können ist dabei abhängig von der von NVIDIA sogenannten *compute capability* (CC) des Grafikchips. Die CC beschreibt die unterstützten Eigenschaften und den Instruktionssatz der GPU und legt unter anderem die maximale Anzahl von Threads innerhalb eines Multiprozessors fest. Für die GTX 480, welche eine CC von 2.0 hat, ist das Limit 1536 Threads [NVIDIA, 2011b, S. 159] und es ergibt sich damit eine maximale Anzahl von gleichzeitig laufenden Threads für die gesamte Grafikkarte von $15 * 1536 = 23040$.

### Speichertypen

Während der Arbeitsspeicher auf PCs im allgemeinen ein homogener Speicherbereich ist, wird der RAM, der der GPU zur Verfügung steht, in mehrere einzelne physische als auch virtuelle Bereiche eingeteilt, wobei jeder von diesen spezielle Eigenschaften und Nutzen hat. Während seiner Ausführungszeit hat ein Thread auf mehrere Speicherbereiche Zugriff: Jeder Thread hat seinen privaten lokalen Speicher für den Stack und Variablen. Abhängig

von der CC beträgt dieser maximal 16 KB oder 512 KB [NVIDIA, 2011b, S. 159]. Weiterhin können Threads innerhalb eines Blockes über den *shared memory* von 16 oder 48 KB zusammenarbeiten, dieser existiert solange wie der Block und kann von allen Threads gelesen und beschrieben werden. Zuletzt haben alle Threads auf den gleichen, geteilten globalen Speicher Zugriff, dieser enthält unter anderem auch die beiden nur lesbaren Konstanten- und Texturenspeicher. Dessen Größe wird dabei nur von der Arbeitsspeichergröße der Grafikkarte beschränkt, allerdings ist er von den verfügbaren Speichern auch der langsamste.

Wichtig für die Arbeit mit CUDA ist die Unterscheidung zwischen dem CPU-Speicher und dem GPU-Speicher. Damit Kernelcode auf Datenstrukturen arbeiten kann, müssen diese zuerst vom RAM des sogenannten Hostsystems (*host memory*, auf der CPU) zum Gerätespeicher (*device memory*, auf der GPU) kopiert werden. Über die CUDA Laufzeitbibliothek werden Funktionen zur Verfügung gestellt, welche Allokationen, Deallokationen und Kopieren von Daten zwischen den Systemen ermöglichen. Auf den oben beschriebenen *device memory* kann über ähnliche Speicherverwaltungsfunktionen wie aus der C-Standardbibliothek zugegriffen werden: `cudaMalloc()`, `cudaFree()` und `cudaMemcpy()`. Der einzige Unterschied bei `cudaMemcpy()` ist, dass ein zusätzlicher Parameter benötigt wird um die Kopierrichtung anzugeben, da CUDA nicht anhand der Zeiger die Speicherbereiche ermitteln kann, in welchen diese sich befinden. Zugriff auf den *shared memory* erfolgt über die `__shared__` Attribute von lokalen Variablen im Kernelcode. Eine Variable welche als *shared* markiert wird, teilt ihren Speicherbereich mit allen anderen Threads des Kernels. *Shared memory* wird von NVIDIA als sehr viel schneller als globaler Speicher beschrieben, insofern ist es auch ein Ziel so viele Speicherzugriffe wie möglich in diesen auszulagern. Eine weiter Variante der üblichen Speicherbereiche ist der *page-locked* oder auch *pinned* Speicher. Je nach Unterstützung (CC) des NVIDIA Grafikchips erlaubt *pinned* Speicher asynchrones gleichzeitiges Kopieren und Kernelausführung [siehe dazu NVIDIA, 2011b, Kapitel 3.2.5] und das direkte Legen von Speicherseiten des Hostspeichers in den Gerätespeicher, wodurch das manuelle Kopieren zwischen den Speichern unnötig wird [NVIDIA, 2011b, Kapitel 3.2.4.3].

### Programmmodell und API

Um über NVIDIA CUDA mit der GPU zu interagieren besteht ein CUDA Programm aus zwei Komponenten: Zum einen dem Kernelcode bestehend aus dem CUDA Instruktionssatz (PTX), zum anderen dem C Code, welcher diesen Kernel aufruft. Während es möglich

ist den PTX Assemblercode per Hand zu schreiben, bietet NVIDIA einen Compiler, *nvcc*, welcher C-Code in PTX-Code übersetzt. *nvcc* verhält sich dabei ähnlich wie der GCC indem es ähnliche Parameter annimmt und kompatible Objektdateien ausgibt. Die Endung für CUDA Code ist dabei überlicherweise `.cu`. Der Compiler unterteilt die Arbeit, um von einer .cu Quellcodedatei auf eine Binärdatei zu kommen, in mehrere Schritte: Als erstes wird der Teil der Datei, welcher auf der GPU ausgeführt werden soll, in das PTX Assemblerformat oder das *cubin* Format übersetzt. Dabei ist der PTX Code eine Zwischensprache (*Intermediary Language*), welche vom Grafikkartentreiber beim ersten Aufrufen speziell für die GPU Version zu Binärcode kompiliert und ausgeführt wird. Dadurch wird die Kompatibilität von CUDA-Anwendungen zu mehreren Grafikkartentypen mit unterschiedlichen Instruktionssätzen gehalten. Das *cubin* Format wird dagegen speziell für einen Instruktionssatz kompiliert und kann demnach auch nur auf Grafikkarten die diesen verwenden funktionieren. Welche Funktionen für die GPU kompiliert werden sollen, wird dabei über die Funktionsattribute `__device__`, `__global__` und `__host__` angegeben. Funktionen ohne Attribute werden standardgemäß nur für das Hostsystem kompiliert. Im zweiten Schritt wird der Hostcode Teil der Datei modifiziert, indem Kernelaufrufe über die `<<<...>>>` Syntax durch korrekte Aufrufe der CUDA API ersetzt werden. Diese Aufrufe laden dann den erstellten PTX/cubin Code und führen ihn aus.

### Debugging von Kernelcode

Ein weiterer wichtiger Teil von CUDA, der in dieser Arbeit angesprochen werden, soll ist die Unterstützung des CUDA Debuggers CUDA-GDB [NVIDIA, 2008]. Da es nicht möglich ist, Kernelcode auf der GPU mit üblichen Debuggern (GNU GDB[3], OllyDBG[4], WinDBG[5], etc.) zu debuggen, stellt NVIDIA eine Portierung des GNU GDB Debuggers zur Verfügung. Das Ziel ist es dem Programmierer eine einzige Umgebung zu geben in der sowohl nativer Hostcode, als auch Devicecode debuggt werden kann. Damit ein CUDA Programm untersucht werden kann, müssen dem Compiler *nvcc* die beiden extra Parameter `-g -G` übergeben werden. Dadurch werden zum einen Optimierungen des Compilers ausgeschaltet, als auch extra Debuginformationen generiert und in der Binärdatei abgelegt.

Während die Menge der Features für GPU-Debugging sehr viel eingeschränkter ist, als die auf dem Hostsystem verfügbaren Techniken, unterstützt CUDA-GDB dennoch die klas-

[3] http://www.gnu.org/software/gdb/
[4] http://www.ollydbg.de/
[5] http://msdn.microsoft.com/en-us/windows/hardware/gg463009

sichen Features eines Debuggers: Mittels breakpoints kann jede Funktion oder Zeile eines Kernels bei Aufruf unterbrochen werden. Dabei werden alle ausführenden Threads unterbrochen. Mittels des neuen GDB-Befehls `thread <<<block, thread>>>` kann zwischen den blockierten Threads und Blocks gewechselt werden. Die aktiven Threads und Blöcke können über den Befehl `info cuda threads` angezeigt werden. Sobald ein breakpoint erreicht wurde, kann die Ausführung mit den GDB Befehlen `next` und `step` weitergeführt werden. Der GDB-Befehl `print` wurde erweitert, um Gerätespeicher auszulesen und anzuzeigen und implizite Laufzeitvariablen wie `threadIdx` und `blockIdx` auszugeben. Weiterhin können allgemeine Informationen über den Zustand von CUDA - wie Hardwaredaten und Speichernutzung - über `info cuda state` angezeigt werden.

Insgesamt erleichtert CUDA-GDB die Erstellung von parallelem Programmcode auf der GPU enorm, allerdings ändert dies nichts an der dazugehörigen Schwierigkeit mit spezialisierten und stark parallelisiertem Code umzugehen. Weiterhin ist CUDA-GDB selbst in Version 4.0 noch relativ instabil und kann dazu führen, dass die gesamte Grafikkarte blockiert wird. Deshalb ist es angebracht selbst für die Entwicklung von CUDA-Kernelcode immer auf einer zweiten Grafikkarte zu arbeiten, die nicht für die Anzeige der Benutzeroberfläche zuständig ist.

### 2.1.1 Thrust

Thrust [Hoberock and Bell, 2010] ist eine C++ Bibliothek zur Unterstützung der effizienten Entwicklung von GPU Algorithmen für CUDA. Dadurch, dass die Bibliothek komplett in der C++ Template-Sprache geschrieben ist, muss sie nicht extra kompiliert werden. Seit CUDA 4.0 ist Thrust außerdem im CUDA 4.0 Toolkit enthalten.

Ein bedeutender Teil von Thrust ist die große Sammlung an generischen Algorithmen, die meist transparent die Parallelität der GPU ausnutzen, um auf auf ihr vorliegenden Datenstrukturen zu arbeiten. Dabei wird Wert darauf gelegt, dass die klassischen C++ Konzepte aus der Standard Template Library (STL), wie Iteratoren, verwendet werden. Viele Probleme können dadurch bereits gelöst werden, ohne überhaupt Code für die GPU zu schreiben. Während CUDA dem Entwickler eine große Freiheit bei der detaillierten Entwicklung von GPU-Algorithmen einräumt, bleibt es dennoch eine aufwendige Arbeit den parallelisierten Algorithmus zu entwickeln und implementieren. Thrust löst dabei das Problem von Algorithmen, die entweder keinen großen Geschwindigkeitsvorteil durch eine Parallelisierung erlangen würden, oder die gar nicht Ziel einer Parallelisierungsarbeit sind. Wegen der relativ hohen Kosten an Zeit zwischen dem Host- und Devicesystem zu kopieren, müssen diese aber dennoch in einer Form auf die GPU portiert werden. Thrusts abstraktes Interface zur CUDA API erlaubt dem Entwickler dagegen, diese Algorithmen in einer abstrakten Form zu beschreiben und die genaue Implementierung des Kernelcodes der Bibliothek zu überlassen.

Listing 1 zeigt eine Beispieloperation mit Thrust: Hier wird zuerst auf dem Hostsystem ein (großes) Array mit zufällig gewählten Zahlen erstellt (Zeile 10-11), welche folgend (Zeile 13) in einem Zug auf die GPU kopiert und dort sortiert (Zeile 15) werden. Zum Abschluss werden diese wieder zurück in den Hostspeicher kopiert. Thrust stellt in diesem Stil viele verschiedene abstrakte Schnittstellen zu grundlegenden Algorithmen - wie sortieren, scannen oder reduzieren - zur Verfügung[6].

### 2.1.2 CUSP

CUSP ist eine Bibliothek zur Unterstützung von Operationen der Linearen Algebra mittels CUDA. Dabei werden ausschließlich Strukturen für dünnbesetzte Matrizen verwendet und Algorithmen zur Lösung von Gleichungssystemen aufbauend auf diesen Strukturen zur

[6] http://docs.thrust.googlecode.com/hg/modules.html

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/generate.h>
4  #include <thrust/sort.h>
5  #include <thrust/copy.h>
6  #include <cstdlib>
7
8  int main(void) {
9    // generate 16M random numbers on the host
10   thrust::host_vector<int> h_vec(1 << 24);
11   thrust::generate(h_vec.begin(), h_vec.end(), rand);
12   // transfer data to the device
13   thrust::device_vector<int> d_vec = h_vec;
14   // sort data on the device
15   thrust::sort(d_vec.begin(), d_vec.end());
16   // transfer data back to host
17   thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
18   return 0;
19 }
```

Listing 1: Thrust Beispiel

Verfügung gestellt. Besonders ist dabei das stark von CUDA abstrahierte Interface welches geläufige Operationen auf Matrizen erleichtert:

Das Beispiel in Listing 2 zeigt wie eine Matrix von der Festplatte geladen, in das *sparse-storage-format* HYB konvertiert und dann transparent in den Grafikkartenspeicher geladen wird. Zum Schluss wird das lineare Gleichungssystem mit einem von CUSPs Lösungsverfahren auf der GPU gelöst.

Alle Datenstrukturen in CUSP (`cusp::*_matrix`, `cusp::array*`) besitzen einen Templateparameter, der aussagt, ob die Struktur im Arbeitsspeicher oder Grafikkartenspeicher gelagert wird. Alle Operationen auf den Strukturen finden dann in den ihr zugehörigen Speicherbereichen statt. Interessant für diese Arbeit ist insbesondere `cusp::multiply` als Matrix-Vektor-Multiplikation sowie die in PETSc verwendete CUSP-Matrixstruktur `cusp::csr_matrix` welche die Besetzungsdaten der Matrix im *compressed sparse row* Format speichert. Als *sparse matrix-vector multiplication* wird der von Bell and Garland [2008, 2009] speziell für die GPU entwickelte Algorithmus verwendet.

Eine vorherige Arbeit von Sören Mahmens am Lehrstuhl „Algorithmische Optimale Steuerung - CO2-Aufnahme des Meeres" von Herrn Prof. Dr. Slawig hat bereits ausgiebig Matrixformate sowie Matrix-Vektor Multiplikationsalgorithmen für die GPU untersucht.

```
1  #include <cusp/hyb_matrix.h>
2  #include <cusp/io/matrix_market.h>
3  #include <cusp/krylov/cg.h>
4  int main(void) {
5      // create an empty sparse matrix structure (HYB format)
6      cusp::hyb_matrix<int, float, cusp::device_memory> A;
7      // load a matrix stored in MatrixMarket format
8      cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");
9      // allocate storage for solution (x) and right hand side (b)
10     cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
11     cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);
12     // solve the linear system A * x = b with the Conjugate Gradient method
13     cusp::krylov::cg(A, x, b);
14     return 0;
15 }
```

Listing 2: CUSP Beispiel

## 2.2 PETSc

Die PETSc[7] Bibliothek, kurz für „Portable, Extensible Toolkit for Scientific Computation", ist eine Sammlung von Datenstrukturen und Algorithmen zur skalierbaren (parallelen) Lösung von numerischen Problemen aus der Wissenschaft. Insbesondere versucht das Toolkit die Repräsentation und Lösung von linearen Gleichungssystemen mit Hilfe von verteilbaren Vektor- und Matrixstrukturen zu vereinfachen. Mit OpenMPI[8] wird der MPI-2[9] *Message Passing Interface* Standard implementiert und unterstützt damit das Skalieren von Rechenproblemen auf mehrere hundert CPUs. PETSc bietet dabei APIs zu einer Reihe von Programmiersprachen und Tools an, unter anderem Fortran, C, C++, Python und MATLAB.

Über ein C-Objektmodell wird für alle Datenstrukturen ein einheitliches Interface unabhängig von der Implementierung deklariert. Insbesondere für Matrizen ist es möglich, über Kommandozeilenparameter oder Konfigurationsdateien zwischen den angebotenen *sparse-storage-formats* frei zu wechseln, ohne den Programmcode verändern zu müssen.

[7] http://www.mcs.anl.gov/petsc/
[8] http://www.open-mpi.org/
[9] http://www.mpi-forum.org/

### 2.2.1 PETSc GPU

In der Arbeit von Minden et al. [2010] wird eine vorläufige Implementierung der PETSc Klassenbibliothek für die NVIDIA CUDA Architektur auf NVIDIA GPUs vorgestellt. Mit Hilfe der CUSP und thrust Bibliotheken wurden große Teile der Vector-Klasse sowie einige Teile der Matrix-Klasse bereits implementiert. Die grundlegenden Probleme zur Interaktion von PETSc mit der GPU wurden gelöst, allerdings wurden Funktionen meist nur so weit implementiert, wie sie zur fehlerfreien und schnellen Ausführung von in Minden et al. [2010] behandeltem Beispielcode nötig waren.

Grundlegend erweitert die GPU Implementierung die PETSc Strukturen um einen Wert, der für die Struktur angibt, in welchem Speicher die Daten am aktuellsten sind. Wenn nun also ein Zugriff auf die Werte eines Vektors von der CPU ausgeführt wird, die Daten aber nur auf der GPU vorhanden sind, werden diese transparent in den Arbeitsspeicher kopiert. Dazu verwendet die Implementierung eine Reihe von Funktionen, welche die Präsenz der Daten in einem bestimmten Speicher garantieren und diese - wenn nötig - kopieren.

PETSc GPU erweitert den Konfigurations- und Bauprozess der PETSc Bibliothek um einige wichtige Details: Bei der Ausführung des *configure* Skriptes ist es nötig, wenn CUDA verwendet werden soll, die Kommandozeile um folgende drei Parameter zu erweitern: `--with-cuda=1 --with-cusp=1 --with-thrust=1`. Mit ihnen wird das Bauskript instruiert, die CUDA, CUSP und thrust Bibliotheken mitzubauen. Weiterhin ist es nötig, für Grafikkarten, welche lediglich einfache Prezision für Fließkommarechnung unterstützen, den Parameter `--with-precision=single` anzugeben. Wenn PETSc auf einem 64-bit System kompiliert und verwendet wird, ist es außerdem nötig mit `--with-nvcc=''nvcc -m64''` zusätzliche Parameter an den CUDA Übersetzer zu übergeben, damit dieser ebenfalls für 64-bit Systeme übersetzt.

## 2.3 Metos3D

Metos3D, kurz für „Marine Ecosystem Toolkit for Optimization and Simulation in 3D", ist ein in der Arbeitsgruppe Algorithmische Optimale Steuerung des Instituts für Informatik der Christian-Albrechts-Universität zu Kiel entwickeltes Softwarepaket zur Simulation und Parameteroptimierung von gekoppelten Ozeanzirkulations- und Ökosystemmodellen. Bei den Modellen handelt es sich dabei um Klimamodelle spezifisch für den Ozean, welche über mehrere tausend Jahre globale Meeresströmungen und biogeochemische Prozesse innerhalb

dieser simulieren sollen. Objekt der Simulation sind dabei Spurenstoffe - auch *Tracer* genannt - welche zum einen durch die Meeresströmungen transportiert werden, zum anderen Bestandteil der angemerkten biogeochemischen Prozesse sind.

In Metos3D wird der Transportprozess von Stoffen im Meer durch die sogenannte „Transport Matrix Method" simuliert, welche insbesondere einen guten Ausgleich zwischen Geschwindigkeit und Genauigkeit trifft [Piwonski and Slawig, 2010, Khatiwala et al., 2005, Khatiwala, 2007]. Dabei wird davon ausgegangen, dass die Wassermassen der Erde in Form von Wassersäulen diskretisiert werden. Die in dieser Arbeit verwendete Ausdehnung der Säule beträgt 2.8125° in Breiten- und Längengrad und bis zu fünfzehn Schichten in der Vertikalen, wobei die Höhe einer Schicht mit der Tiefe zunimmt. Dabei ist die Anzahl der verwendeten Schichten der einzelnen Säulen von der tatsächlichen geographischen Tiefe abhängig: Während sie am Land null beträgt, werden zum Beispiel zur Darstellung des Milwaukeetiefs alle fünfzehn Schichten benötigt. Der Einfachheit halber werden die Stoffkonzentrationen in einem eindimensionalen Vektor gespeichert, welches für die obige Auflösung mit den unterschiedlichen Tiefen der Säule eine Länge von 52749 Stoffkonzentrationswerten besitzt.

Für das Verfahren von Khatiwala et al. werden von den Transportmatrizen sowohl explizite als auch implizite Varianten benötigt. Die explizite Variante behandelt dabei die Verteilung von Stoffen in den zwei Dimensionen der Längen- und Breitengrade, während die implizite Variante die Verteilung in der Tiefe behandelt. Da es aus Effizienzgründen nicht möglich ist, für jeden Zeitpunkt einer üblichen zeitlichen Diskretisierung eines Jahres (in Metos3D meist 2880 Zeitschritte) zwei Matrizen zu speichern, werden stattdessen zwölf monatlich gemittelte Matrizen linear interpoliert.

Insgesamt beschreibt folgende Gleichung die für einen Zeitschritt durchgeführte Rechnung zur Simulation der Meeresströmungen:

$$y_{i+1} = A_{imp,i}(A_{exp,i}\, y_i + q_i(y_i, u))$$

Dabei beschreibt $y_i$ die Stoffkonzentration zum Zeitschritt $i$; $q_i$ den Algorithmus des biogeochemischen Modells, $u$ seine Parameter und $A_{imp,i}$, $A_{exp,i}$ die oben beschriebenen interpolierten Transportmatrizen.

Der zweite Teile der Simulation, die Anwendung des biogeochemischen Modells, wird von Metos3D über eine Schnittstelle zu einem Fortran Programm übernommen. Über drei Funktionen wird das Modell initialisiert, ausgeführt und deinitializiert. Die Anwendung

erfolgt dabei über eine Funktion, welche für jede Wassersäule der Simulation pro Zeitschritt aufgerufen wird. Innerhalb dieser kann der Modellierer die biogeochemischen Prozesse in den einzelnen Wasserschichten modellieren. Die genauen Details und Signaturen der Schnittstelle werden in Piwonski and Slawig [2010] beschrieben. Solange das Modellprogramm diese Schnittstelle implementiert und damit die übergebenen Eingaben annimt, sowie die erwarteten Ausgaben in Form der modifizierten Stoffkonzentrationswerte in den Schichten ausgibt.

Zum Zeitpunkt dieser Arbeit befinden sich zwei Beispielmodelle[10] im Metos3D Simulationspaket, welche in Kapitel 4 zur Laufzeitanalyse verwendet werden: Das I-Cs Modell ist ein leichtgewichtiges Modell, welches den Zerfall von Jod- und Cäsiumisotopen über zwei Differentialgleichungen beschreibt. Das N-DOP Modell ist dagegen ein Beispiel eines komplexeren Modells und wird genauer in Piwonski and Slawig [2010, S. 7] beschrieben.

## 2.4 PGI CUDA-Fortran

Fortran ist eine Sprache mit einer Vielfalt an verfügbaren Übersetzern für diverse Systeme. Für diese Arbeit ist es nötig gewesen, Fortran-Programme für die GPU, insbesondere die NVIDIA CUDA Architektur, zu übersetzen. Damit wird zum einen die Eigenschaften der GPU auch für diesen Teil ausgenutzt werden können und zum anderen das Kopieren der Stoffkonzentrationsdaten zwischen dem Arbeitspeicher der CPU und GPU innerhalb eines Zykluses vermieden wird.

Mit dem kommerziellen PGI CUDA-Fortran Compiler[11] stellt „The Portland Group" einen Fortran Übersetzer für die CUDA Architektur zur Verfügung, welcher die Fortran-Sprache um die in Kapitel 2.1 erwähnten sprachlichen Konstrukte zum Aufruf von Kernelfunktionen, als auch die CUDA API-Funktionen erweitert. Die Syntax eines Kernelaufrufs ist mit `call kernel<<<...>>>()` ähnlich wie in CUDA C++ und Variablenattribute werden in der üblichen Fortran Syntax angegeben [The Portland Group, 2011b, S. 5]. Die Menge der möglichen Attribute deckt sich dabei weitestgehend mit denen aus CUDA C. CUDA-Fortran bietet aber zusätzlich die Möglichkeit über die `device` Attribute automatisch Speicherplatz im Gerätespeicher zu reservieren, ohne `cudaMalloc` verwenden zu müssen. CUDA-Fortran Prozeduren und Funktionen erhalten dabei wie in CUDA C auch extra Attribute, die bestimmen für welche Architektur der Quellcode übersetzt werden soll.

[10] http://www.informatik.uni-kiel.de/~algopt/metos3d/models/
[11] http://www.pgroup.com/resources/cudafortran.htm

Für Fortran Quellcode, der für die NVIDIA GPU übersetzt wird gibt es allerdings einige Einschränkungen: Kernelaufrufe, und Aufrufe von anderen Funktionen, welche mit dem `device` Attribut übersetzt wurden, sind nur innerhalb eines Moduls möglich. Weiterhin ist es nicht möglich in Kernel- und Gerätefunktionen lokale Felder mit einer dynamischen Größe zu definieren. Die Größe des Stacks einer solchen Funktion muss zum Zeitpunkt der Übersetzung bestimmbar sein. Weitere Beschränkungen werden im PGI CUDA-Fortran Handbuch aufgeführt [The Portland Group, 2011b, S. 14].

# 3 Implementierung

Nach der Untersuchung der genannten Technologien und weiterer Bibliotheken zur GPU-Berechnung wurde die Entscheidung getroffen, die anfängliche PETSc GPU Implementierung als Ausgangsbasis zur Modifikation von Metos3D zu verwenden. Wie bereits in Kapitel 2.2 beschrieben baut PETSc auf einem objektorientierten Modell in der Sprache C auf. Dadurch lassen sich die nötigen Änderungen an Metos3D auf direkte Manipulationen der Profilvektoren beschränken. Die restlichen Änderungen werden größtenteils durch die von PETSc definierte Schnittstelle verdeckt.

Ein Hauptziel der Arbeit ist es soweit wie möglich die Kompatibilität zu den von Modellierern entwickelten biogeochemischen Fortran-Modellen zu wahren, solange diese die in Piwonski and Slawig [2010] beschriebene Schnittstelle implementieren. Deshalb mussten teilweise etwas umständliche Wege genommen werden um den PGI CUDA-Fortran Übersetzer einzubinden.

## 3.1 Modifikationen an PETSc

Die PETSc GPU Implementierung ist wie beschrieben eine vorläufige Fassung: Die Autoren Minden et al. [2010] hatten lediglich das Ziel die Löser für Gleichungssysteme auf der GPU auszuführen. Aus diesem Grund waren zum Zeitpunkt dieser Arbeit noch nicht alle nötigen Funktionen vollständig für die GPU implementiert. Dies machte es nötig, innerhalb eines Metos3D Simulationszyklus die Profildaten mehrmals aufwendig von der GPU auf die CPU oder umgekehrt zu kopieren, wodurch der Beschleunigungseffekt noch minimal war. Es war also außerdem nötig die PETSc GPU Bibliothek zu erweitern.

PETScs C Objektmodell implementiert eine Art Virtual Method Table, wie sie in C++ Programmen verwendet wird, um das OOP Prinzip der Funktionsüberladung in Vererbten Klassen umzusetzen. Dazu besitzt jede Instanz einer PETSc Klasse einen Zeiger *ops* auf eine Tabelle, welche den möglichen Operationen einen Zeiger auf eine Funktion zuordnet. Dadurch werden über das gleiche Interface die *sparse-storage-formats* als auch die verwendete *AIJCUSP* GPU Implementierung abstrahiert.

PETSc setzt vorraus, dass C-Funktionen ein gewisses Format einhalten, damit bei Fehlern korrekte Stacktrace-Informationen über den Ursprung des Fehlers gegeben werden können. Listing 3 zeigt die nötigen Komponenten einer Funktion: Zeile eins bis zwei definieren für folgende Makros den Funktionskontext und sorgen unter anderem dafür, dass bei Fehleranalysen (z.B. mit dem Makro CHKERRQ) die Urpsprungsfunktion angegeben

```
1 #undef __FUNCT__
2 #define __FUNCT__ "MatScale_SeqAIJCUSP"
3 PetscErrorCode MatScale_SeqAIJCUSP(Mat inA, PetscScalar alpha) {
4         // ... Funktionskörper ...
5         PetscFunctionReturn(0);
6 }
```

Listing 3: PETSc Funktionsaufbau

werden kann. Die PETSc User Reference [Balay et al., 2011a] legt außerdem fest, dass jede Funktion einen `PetscErrorCode` zurückgibt, welcher den Erfolg oder Misserfolg einer Funktion beschreibt. Anstelle des üblichen `return`s wird in Zeile fünf ein `PetscFunctionReturn` verwendet um den Fehlercode zurückzugeben. Ein Fehlercode von Null sagt dabei aus, dass kein Fehler aufgetreten ist.

### 3.1.1 MatCopy

Die Wrapper-Funktion MatCopy ist in PETSc dafür verantwortlich, die Werte einer Matrix in die eine andere Matrix zu kopieren. Zusätzlich zu der Quell- sowie Zielmatrix informiert ein dritter Parameter die Funktion darüber, ob die Zielmatrix im Falle einer sparse-matrix die gleiche Anzahl nicht-Nullwerte hat wie die Quellmatrix. Dadurch können unnötige Speicherreservierungen vermieden werden.

Für die GPU Variante (`MatCopy_SeqAIJCUSP`) ist es für beide Eingabematrizen möglich, dass sie entweder im GPU-Speicher, im CPU-Speicher oder in beiden aktuell sind. Für eine vollständige, korrekte Implementierung wäre es nötig gewesen alle diese Fälle abzudecken und entsprechend auszuwählen innerhalb welchen Speicher die Matrizen tatsächlich kopiert werden. Für Metos3D reichte es allerdings aus, nur den GPU-GPU Fall abzudecken: Innerhalb eines Zyklus' sollen sich die Daten vollständig auf der GPU befinden. Deshalb wird mit `MatCUSPCopyToGPU` für beide Matrizen sichergestellt, dass diese sich im Grafikkartenspeicher befinden.

Da die GPU-Varianten der Matrix-Klasse durch CUSP-Strukturen repräsentiert werden, ist es möglich die Kopierroutinen von CUSP zu verwenden.

### 3.1.2 MatScale und MatAXPY

`MatScale` und `MatAXPY` sind beide typische Funktionen in Bibliotheken zur Bearbeitung von Problemen der linearen Algebra und können aus diesem Grund fast eins-zu-eins durch

die CUSP BLAS Bibliothek auf die GPU portiert werden. Dazu müssen die Funktionen auf den Vektor der nicht-null Elemente der Matrix angewendet werden. Die CUSP Matrixklasse stellt dafür ein Feld in der Klasse zur Verfügung, welches einen Zeiger auf die Datenstruktur im GPU Speicher zurückgibt.

## 3.2 Metos3D Erweiterungen

Wie in Kapitel 2.3 erwähnt, besteht Metos3D aus etwa zwei Teilen: Der Simulationssoftware in C und dem biogeochemischen Modell in Fortran. Während die Erweiterungen an PETSc dazu dienten die in Metos3D verwendeten und bereits in Mahmens [2011] untersuchten Algorithmen für Matrix-Vektor-Multiplikation und Interpolation auf die GPU auszulagern, verbleiben kleinere in Metos3D laufende Algorithmen und der Teil der biogeochemischen Simulation des Fortran-Modells um die gesamte Simulation auf die GPU zu übertragen.

Die in Metos3D simulierten Stoffkonzentrationen werden aus Kompatibilitätsgründen mit dem PETSc Löser für nichtlineare Gleichungssysteme (SNES, Balay et al., 2011a, S. 91) sowie aus Effizienzgründen bei einer Vielzahl von Spurenstoffen in mehreren Formaten abgelegt. Für die in Kapitel 2.3 erwähnte Transportgleichung von Stoffen werden die Vektoren SEPARAT gehalten, das heißt, für jeden Stoff sind die Konzentrationsdaten für die einzelnen Profile und Tiefen in einem eigenen Vektor gehalten. Es gibt also so viele Vektoren wie Stoffe in der Simulation. Dadurch ist es möglich die Transportgleichung mehrmals auszuführen, einmal auf jeden Stoff, anstatt für die verwendete Anzahl an Gleichungen die Transportmatrizen zu vergrößern und zu kopieren. Wäre dies nicht möglich, würde die Größe der Transportmatrix linear mit der Anzahl der Stoffe wachsen, welches selbst bei effizienter Datenhaltung zu zu großen Datenmengen führt. Für die Anwendung des biogeochemischen Modells werden die Daten dagegen in einen einzigen Vektor kopiert (DIAGONAL), da jeder Modellaufruf Zugriff auf alle vorhandenen Stoffkonzentrationen braucht.

Die Algorithmen, welche zwischen diesen Formaten konvertieren, wurden mit Hilfe der Thrust Bibliothek (Kapitel 2.1.1) effizient auf die NVIDIA Architektur konvertiert. Konkret werden Iteratoren des Typs `thrust::counting_iterator<int>` aufgesetzt, um alle Profile der Simulation durchlaufen zu können. Mittels `thrust::for_each` und den erstellten Iteratoren wird dann über Thrust ein sogenannter *functor* auf die GPU kopiert und ausgeführt. Ein *functor* ist dabei ein C++-Konstrukt, welches eine Instanz einer Struktur und deren Zustand in Form von Variablen, sowie einer durch den `operator()`-Operator definierten Funktion ist (also eine Funktion mit Zustand). Diese Funktionen sind dabei ab-

hängig davon, zwischen welchen Formaten auf der GPU konvertiert werden soll. Es wurden `Metos3DUtilVecCopySeparateToDiagonal`, `Metos3DUtilVecCopyDiagonalToSeparate` und `Metos3DUtilVecCopySeparateToDiagonalProfile` zu Thrust portiert und implementiert, welche in der Datei `metos3d_util_gpu.cu` zu finden sind.

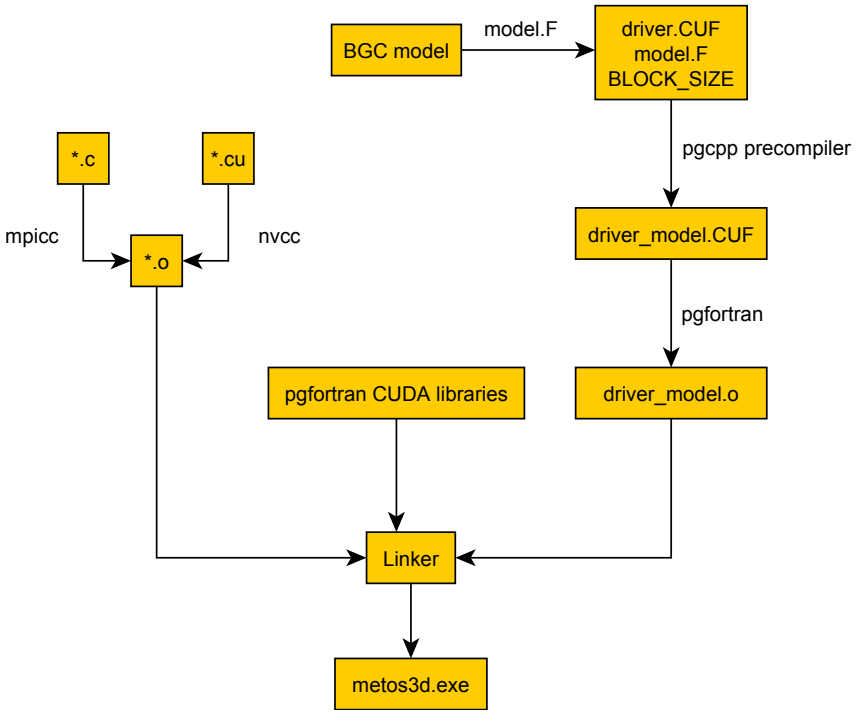


Fig. 3.1: Ablauf der Übersetzung von Metos3D und dem BGC Modell

Es verbleibt die tatsächliche Simulation der biogeochemischen Effekte. Dazu ist es zuerst nötig den Fortran-Quellcode dieser Modelle in die vom NVIDIA CUDA Treiber verständliche Zwischensprache des PTX-Assemblers zu übersetzen. Diese Arbeit wird vom PGI CUDA-Fortran Compiler (Kapitel 2.4) übernommen: Wie in Abbildung 3.1 zu sehen, wird zu Anfang mit Hilfe eines Vorübersetzers (*precompiler*) aus dem vom Modellierer erstellten biogeochemischen Fortranmodell und einem „Treiber" das Modell in eine Form gebracht, welche ohne große Änderungen und Umstände in den Metos3D C-Teil integriert werden kann. Die Kombination aus dem Modell und dem Treiber wird dann mit dem PGI CUDA-Fortran-Compiler zu einer Objektdatei übersetzt. Parallel dazu erfolgt die Übersetzung des Metos3D Programmcodes: Der Originalcode der C-Dateien wird wie vorher mit dem OpenMPI-Wrapper des GNU C Compilers `mpicc` übersetzt, während Erweiterungen in

Form von CUDA Kerneln und ihre Wrapperfunktionen mit dem NVIDIA CUDA Compiler übersetzt werden.

Vom PGI CUDA-Fortran Compiler übersetzte Programme sind von einer Reihe von extra Bibliotheken abhängig, welche vom Hersteller mitgeliefert werden. Diese werden standardgemäß vom PGI CUDA-Fortran Linker bei der Erstellung der endgültigen ausführbaren Datei mit eingebunden. Da Metos3D aus verschiedenen Gründen abhängig vom OpenMPI Linker ist, müssen diese in einem dritten, parallelen Schritt zum Zeitpunkt der Erstellung der ausführbaren Datei mit eingebunden werden.

Der Treiber (`models/driver.CUF`) hat hauptsächlich zwei Probleme zu lösen:

Zum einen erfordert der PGI CUDA-Fortran Übersetzer, dass alle Funktionen welche, für die GPU kompiliert werden und auf dieser laufen sollen, mit dem `device` Attribut markiert werden [The Portland Group, 2011b]. Da der Übersetzer keine Möglichkeit bietet die Standardattribute für alle Funktionen zu setzen, ist es nötig diese über eine Precompiler Definition einzubinden. Das Schlüsselwort `subroutine,` welches zur Deklaration einer Funktion in Fortran verwendet wird, wird mit `attributes(device) subroutine` ersetzt. Da weiterhin der PGI Fortran Precompiler [The Portland Group, 2011c,a] noch in der Entwicklungsphase ist und bei dieser Definition in eine Endlosschleife läuft (durch das Vorkommen von `subroutine` in dem zu ersetzenden Wort und der Ersetzung), ist es nötig dazu den PGI C++ Precompiler zu verwenden. Der Fortran Quellcode des biogeochemischen Modells wird daraufhin innerhalb eines Fortran Moduls eingebunden, da CUDA-Fortran keine globalen Funktionsdeklarationen von `device`-Funktionen zulässt.

Zum anderen muss der Treiber Hilfsfunktionen für die drei Einsprungspunkte in das Fortranmodell zur Verfügung stellen: `metos3dbgc`, `metos3dbgcinit` und `metos3dbgcfinal`, also die Hauptsimulationsfunktion sowie eine Initialisierungs- und Deinitialisierungsfunktion. Zusätzlich dazu müssen für alle drei Funktionen Kernelfunktionen definiert werden, welche die enstprechend dazugehörige Funktion des Modells auf der GPU aufrufen. Der Grund dafür ist, dass Kernelaufrufe eine feste Menge von Parametern haben, welche für jeden Thread auf der GPU die gleiche ist. Da die Simulationsfunktionen des Modells aber einen Zeiger auf den Speicherbereich für genau ein Profil erwarten, ist es nötig über die Kernelvariablen `blockidx` und `threadidx` das im aktiven Thread zu bearbeitende Profil zu berechnen und den dazugehörigen Zeiger an das Modell zu übergeben. Die drei Einsprungsfunktionen sind dafür zuständig sowohl die Anzahl der Blöcke in Abhängigkeit von der zum Zeitpunkt der Kompilation gewählten Blockgröße zu berechnen, als auch die internen Kernelfunktionen mit den übergebenen Parametern aufzurufen.

# 4 Auswertung

In diesem Kapitel werden die Auswirkungen der oben beschriebenen Änderungen und Bibliotheken auf die Laufzeit von zwei Metos3D Beispielmodellen beschrieben, und die Ergebnisse erläutert. Dazu wird zuerst der Aufbau der Testumgebung beschrieben, welche zum einen das PC-System auf dem die Tests durchgeführt werden enthält, zum anderen die Konfigurationsdateien für Metos3D. Daraufhin wird der genaue Ablauf der Tests, sowie die genauen Details der untersuchten Funktionen und Parameter beschrieben, um zum Schluss Ergebnisse im Vergleich betrachten zu können.

## 4.1 Testaufbau

Für alle Tests und Simulationen wurde ein vom Lehrstuhl Kommunikationssysteme[12] der CAU Kiel zur Verfügung gestelltes System verwendet.

| Komponente | Details |
|---|---|
| Prozessor | 2x Intel(R) Xeon(R) CPU E5520 @ 2.27GHz je 4 hyperthreaded Cores |
| Arbeitsspeicher | 40 GB |
| Grafikprozessor | 2x GeForce GTX 480 je 1.5 GB RAM und<br>15 Multiprocessors, insgesamt 480 CUDA Kerne |

Tab. 1: Technische Details des Testsystems

Tabelle 1 beschreibt den Aufbau des Testsystems. Anzumerken ist, dass für die folgenden Testaufbauten und Ergebnisse nur eine Grafikkarte verwendet wurde. Bei der Reproduktion der Tests ist darauf zu achten, dass die verwendete Grafikkarte nicht von einem etwaigen grafischen Betriebssystem belastet wird. Viele grafische Benutzeroberflächen sind mittlerweile GPU-Beschleunigt, wodurch die Ergebnisse verfälscht werden können. Insbesondere ist es möglich, dass auf der Grafikkarte nicht mehr ausreichend Speicherplatz für die Transportmatrizen verfügbar ist. Diese nehmen mit einer Auflösung von 2.8° für ein ganzes Jahr etwa 1 GB Grafikkartenspeicher ein.

Die GeForce GTX 480 ist bereits ein älteres Modell der GTX Baureihe. Zum Zeitpunkt des Schreibens besitzt die GTX 580 mit 512 CUDA Cores bereits mehr als ihr Vorgänger. Weiterhin besitzt die GeForce GTX 590 als Zusammenschluss zweier GTX 580 Grafikkarten mit 1024 CUDA Kernen bedeutend mehr Kerne und es ist damit ein weit größerer Beschleunigungseffekt zu erwarten.

[12] http://www.informatik.uni-kiel.de/comsys/

### 4.1.1 Konfiguration

Ausschlaggebend für die Dauer und den Aufwand der BGC-Simulation in Metos3D sind die verwendeten Konfigurationsdateien. Diese erlauben dem Benutzer eine Anpassung einer Vielzahl von Simulationsparametern. Die vollständigen verwendeten Parameter sind im Anhang zu finden, folgend nur eine kurze Nennung und Erklärung der wichtigsten, sowie ihre Auswirkung auf die Laufzeit.

#### Metos3DMatrixCount

Die Anzahl der zu verwendenden Transportmatrizen (siehe Kapitel 2.3), zwischen denen interpoliert wird, beeinflusst direkt drei Merkmale: Die Qualität der Simulation ist abhängig von der Genauigkeit der Transportmatrizen. Da für jeden Zeitschritt eine Transportmatrix benötigt wird, es aber nicht möglich ist 2880 (gegeben durch `Metos3DTimeStepCount`, die verwendete Anzahl von Zeitschritten pro Jahr) von diesen in den Speicher zu laden, werden die beiden jeweils zeitlich am nahesten beieinander liegenden linear interpoliert. Mit einer Auflösung von etwa 2.8° passen die zwölf verwendeten Matrizen in etwa in den Arbeitsspeicher der verwendeten Grafikkarte (1.5 GB). Für die genauere, auch verfügbare, Auflösung von 1° ist es nötig, die Rechenarbeit auf mehrere Grafikkarten aufzuteilen: Wegen ihrer Größe passen die Matrizen nicht mehr vollständig in den Speicher, was es ansonsten nötig machen würde innerhalb eines Zyklus' wiederholt Matrizen vom Arbeitsspeicher des Hostsystems in den der Grafikkarte zu kopieren.

#### Metos3DSpinupCount

Als Simulationsmodus wurde der Spinup-Löser von Metos3D verwendet, da dieser eine durchgehende Simulation einer beliebigen Anzahl von Zyklen erlaubt und sich damit für eine Geschwindigkeitsanalyse am besten eignet. Als Laufzeit wurde für die GPU- und CPU-Varianten eine Simulationsdauer von 100 Jahren verwendet. Damit bleibt die Gesamtlaufzeit eines Tests innerhalb erträglicher Größen und es werden mit insgesamt $100 * 2880$ Zeitschritten die gröbsten Messfehler ausgeschlossen.

#### Metos3DTracerCount

Ein weiterer wichtiger Parameter ist die Anzahl der Spurenstoffe. Abgesehen von der biogeochemischen Simulation in Fortran muss für jeden Stoff die Transportgleichung aus Kapi-

tel 2.3 berechnet werden. Das beinhaltet zum größten Teil die Matrix-Vektor Multiplikation und ist damit neben dem Modell der Parameter mit dem größten Einfluss auf die gesamte Laufzeit der Simulation. Die verwendeten Modelle besitzen jeweils zwei Stoffe. In komplexeren Modellen ist es aber möglich, dass diese aus mehreren Dutzend Stoffen bestehen.

### 4.1.2 Ablauf

Interessant für diese Arbeit ist der Vergleich der Zeiten der einzelnen Operationen innerhalb eines Jahreszyklus'. Das nötige Zeitschrittverfahren dafür findet im Programm innerhalb der `Metos3DTimeStepPhi` Funktion statt. Tabelle 2 zeigt die Funktionen, die innerhalb des Kontextes eines Zyklus' betrachtet werden. Die Funktionen wurden auf der Erkenntnis basierend gewählt, dass sie in vorangehenden Laufzeittests in der CPU-Variante die meiste Laufzeit einnahmen.

| Funktion | Beschreibung |
|---|---|
| Metos3DTimeStepPhi | Gesamtdauer des Zyklus' |
| Metos3DBGCStep | Ausführung des biogeochemischen Modells für einen Zeitschritt |
| MatScale, MatAXPY, MatCopy | Interpolation zwischen zwei Transportmatrizen |
| MatMult | Multiplikation der Transportmatrizen mit dem Tracervektor |

Tab. 2: Betrachtete Funktionen der Simulation

Zur Durchführung einer exakten Zeitmessung ist ein genaues und solides Profilingsystem nötig, welches PETSc mit PETSc Profiling[13] bietet. Es wird ein Profiling Objekt mittels `PetscLogEventRegister` erstellt und mit `PetscLogEventBegin` und `PetscLogEventEnd` der Bereich, für den die Zeit gemessen werden soll, markiert. Über die PETSc Option `-log_summary`, welche als Kommandozeilenparameter übergeben werden kann, wird der Profiler aktiviert und gibt nach Programmende eine Zusammenfassung aller erstellten und ausgeführten Profilingblöcke aus.

Mit den Konfigurationseinstellungen aus 4.1.1 wurden auf dem System aus 4.1 die Simulationen auf der CPU und der GPU mit variierenden Blockgrößen für den CUDA Kernel des biogeochemischen Modells durchgeführt. Aufrufe von CUDA Kernelfunktionen blockieren standardgemäß nicht bis dieser Aufruf fertig ist: Erst wenn Daten von der GPU kopiert werden sollen oder bei einem Aufruf von `cudaSynchronize()` wartet CUDA, bis alle Kernelaufrufe beendet sind. Deshalb ist es wichtig für eine genaue Zeitmessung gerade dies zu tun. PETSc integriert in alle seine Funktionsaufrufe von CUSP und CUDA

[13] http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Profiling/index.html

eine Abfrage einer globalen Variable, welche aussagt, ob gewartet werden soll, bis die Funktionen auf der Grafikkarte beendet sind. Über den PETSc Einstellungsparameter `-cusp_synchronize` oder den entsprechenden Kommandozeilenparameter lässt sich diese Funktion kontrollieren. Bei einer Angabe von `-log_summary` wird diese Funktion außerdem zusätzlich aktiviert, da eine Zeitmessung mit dem PETSc Profiling Modul ansonsten keinen Sinn ergeben würde.

## 4.2 Ergebnisse

Folgend sollen die Ausgaben des PETSc Profilers analysiert und visualisiert werden, um sowohl den Beschleunigungseffekt zu messen, als auch Angriffspunkte für weitere Optimierungen zu finden. Es werden dabei sowohl das I-Cs als auch N-DOP Modell betrachtet, da sich beide in der Komplexität des Fortrancodes und der Anzahl der Spurenstoffe unterscheiden.

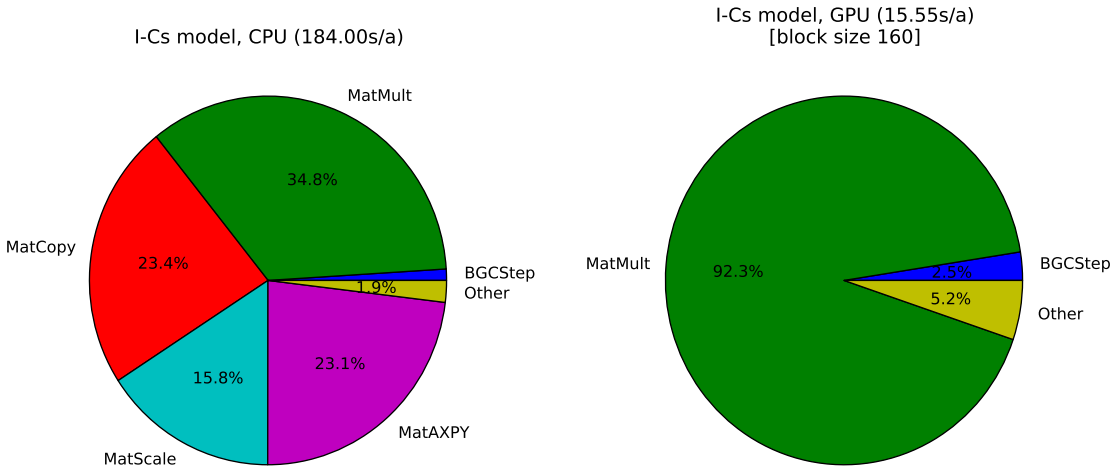


Fig. 4.1: I-Cs Modell CPU/GPU

Abbildungen 4.1 und 4.2 zeigen in Form von Tortendiagrammen die Anteile der in 4.1.2 beschriebenen Funktionen an der gesamten Simulationsdauer eines Zykluses. Beide Paare lassen erkennen, dass während der Anteil der `MatCopy`-, `MatScale`- und `MatAXPY`-Funktionen auf der CPU noch relativ groß ist, er auf der GPU so gering wird, dass er

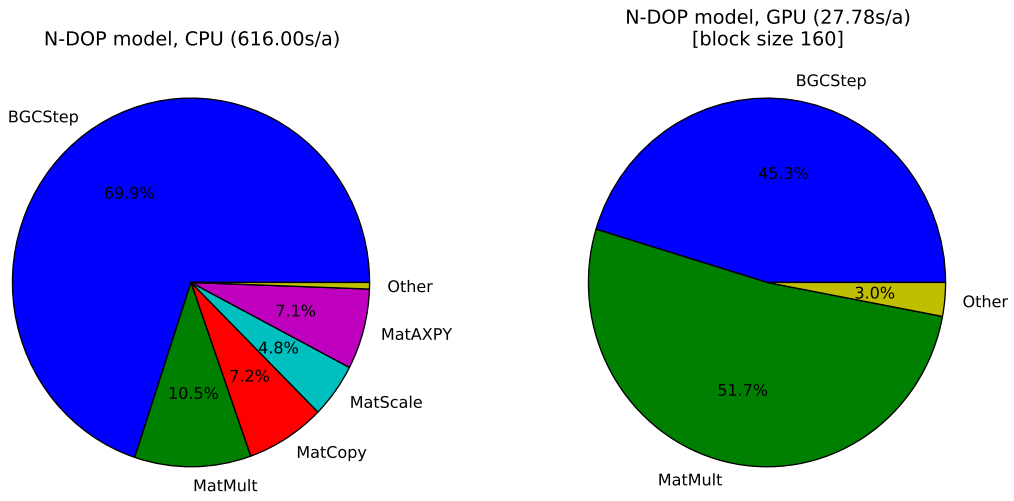


Fig. 4.2: N-DOP Modell CPU/GPU

im Diagramm nicht mehr darstellbar ist. Dies lässt sich dadurch erklären, dass alle drei Operationen genau die Hauptfunktion der GPU treffen: Viele unabhängige und einfache Operationen die sich leicht parallelisieren lassen. `MatScale` und `MatAXPY` sind für jede Zelle der Matrix simple Multiplikationen und Additionen ohne Konditionalblöcke, welche durch die schlechte Branch-Prediction auf der GPU verlangsamt werden könnten. `MatCopy` nutzt die sehr viel höhere Bandbreite der Speicherarchitektur, welche in den dreistelligen Gigabyte pro Sekunde-Bereich liegt [NVIDIA, 2011a, S. 17].

Grundsätzlich verteilt sich auf der GPU die Laufzeit auf die Matrix-Vektor-Multiplikation (`MatMult`) und die Simulation des BGC-Modells (`BGCStep`). Während die absolute, mit der Anzahl der Spurenstoffe normalisierte Laufzeit von `MatMult` für alle BGC-Modelle wegen gleicher verwendeter Matrizen gleich bleibt, ist `BGCStep` stark von der Komplexität des Modells abhängig. Das I-Cs Modell ist ein rechnerisch eingeschränktes Modell, bestehend aus der Anwendung zweier Differentialgleichungen auf jeden Layer eines jeden Profils. Das macht sich durch seinen verhältnismäßig geringen Anteil an der Rechenzeit auf beiden Recheneinheiten bemerkbar. Das N-DOP Modell ist dagegen bedeutend komplexer und ineffizienter: Es besteht aus mehreren verschachtelten Schleifen und vielen Konditionalblöcken, welche die Rechenzeit stark erhöhen. Dadurch ist in Abbildung 4.2 der Anteil von `BGCStep` an der Berechnung sehr viel größer.

Insgesamt beschleunigt sich die gesamte Simulation mit dem I-Cs Modell um einen Faktor von 11.8, während der Beschleunigungsfaktor vom N-DOP Modell mit 20.8 sehr viel größer ist. Es lässt sich daraus also schliessen, dass insbesondere komplexere Modelle von der Portierung auf die GPU den größten Beschleunigungseffekt erfahren.

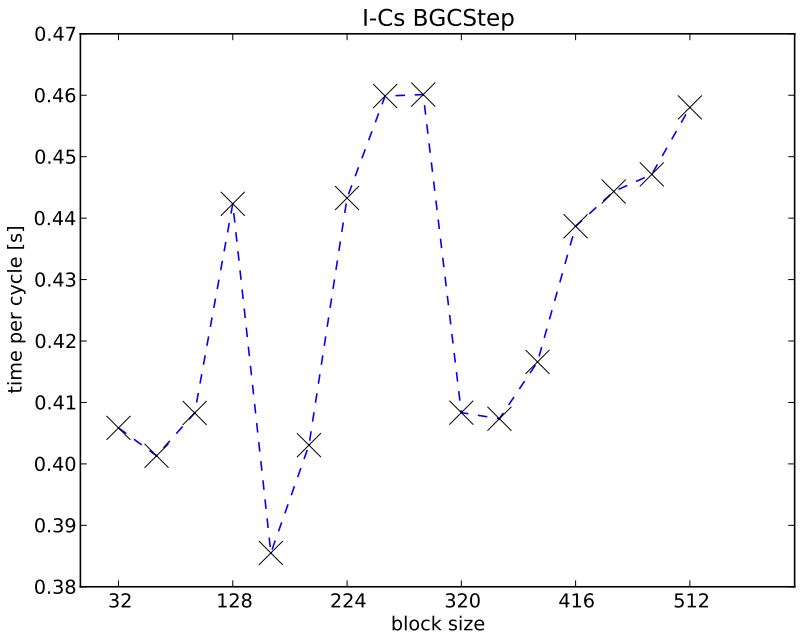


Fig. 4.3: Simulationsdauer verschiedener Blockgrößen, I-Cs Modell

Ein weiterer zu untersuchender Parameter ist die verwendete Blockgröße für den PGI-CUDA-Fortran Kernelaufruf des BGC Modells. Die Blockgröße beschreibt für das Modell die Anzahl der Profile, die innerhalb eines Blockes vom Modell bearbeitet werden. In Abbildung 4.3 und 4.4 werden die getesteten Blockgrößen für beide Beispielmodelle von Metos3D gezeigt. Die Graphen geben dabei die Laufzeit des biogeochemischen Modells über einen Zyklus (also 2880 Zeitschritte) an. Bei beiden Modellen gibt es starke Schwankungen - bis zu 100% der Laufzeit in Abhängigkeit von der Blockgröße - und bei beiden Modellen ist das absolute Minimum der Laufzeit bei einer Blockgröße von 160 (I-Cs: ~0.35s, N-DOP: ~13s). Weiterhin lassen beide Graphen für die gemessenen Blockgrößen eine gemeinsame Struktur mit ähnlichen Minima und Maxima erkennen. Die Struktur des Graphen der N-DOP Simulation ist ausgeprägter als die des I-Cs-Modells, da der Anteil des BGC Modells an

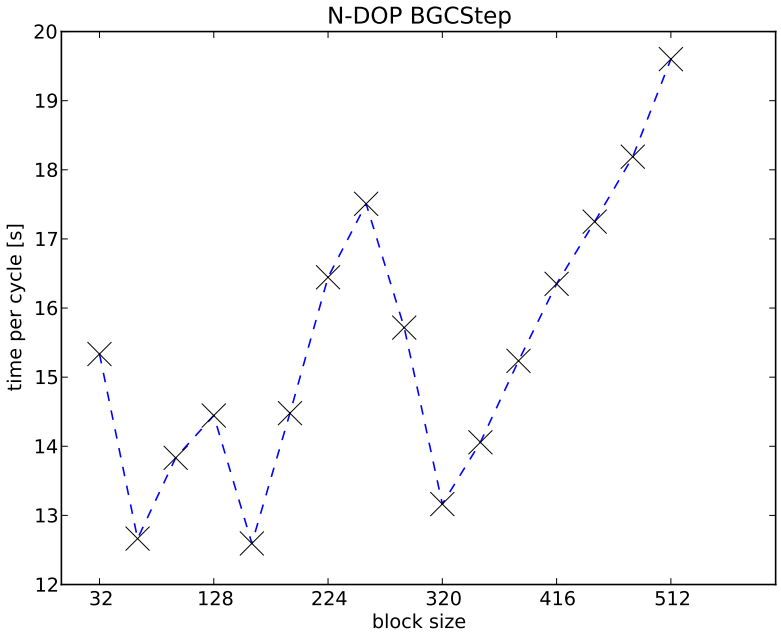


Fig. 4.4: Simulationsdauer verschiedener Blockgrößen, N-DOP Modell

der Gesamtrechenzeit von Metos3D im ersten Fall bedeutend höher ist, und zudem weniger von Caching oder anderen schwer berechenbaren Effekten beeinflusst wird.

## 4.3 Ausblick

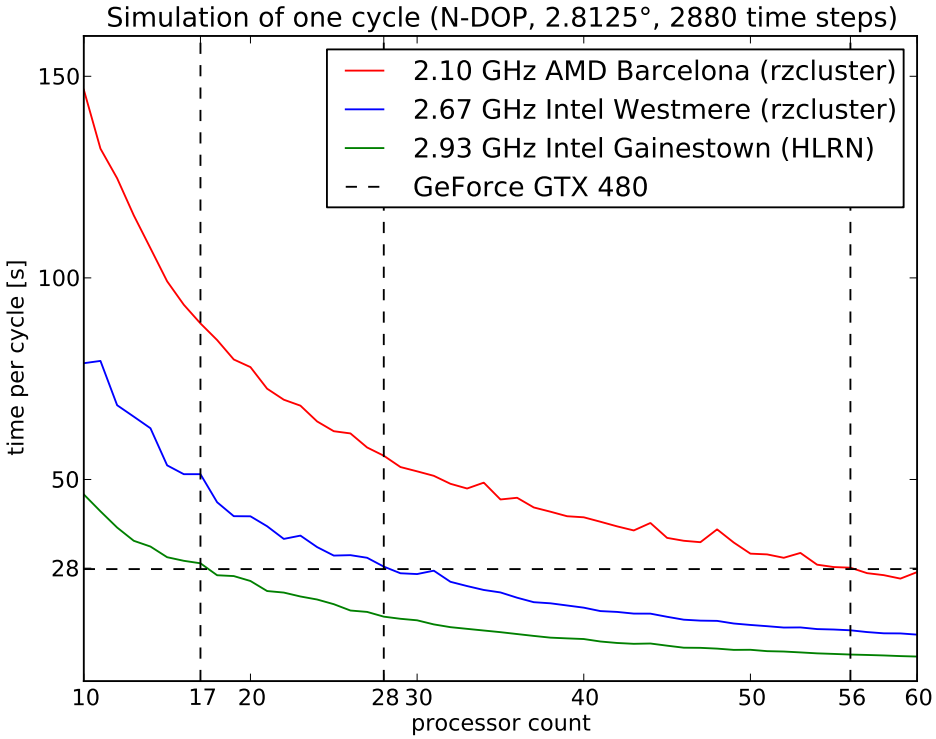


Fig. 4.5: Geschwindigkeitsvergleich von CPU Cluster und einer GPU [CPU Messdaten von Jaroslaw Piwonski]

In dieser Arbeit wurde als erster vollständigen Schritt auf die Grafikkarte ein Simulationsprogramm portiert, sodass es komplett auf der GPU läuft. Dadurch wurden bereits enorme Geschwindigkeitsgewinne erzielt, welche sonst erst mit mehreren Dutzend CPUs erreicht werden können. Während der Algorithmus zur Matrix-Vektor-Multiplikation [Bell and Garland, 2008, 2009] bereits als hochoptimiert zu bezeichnen ist, lässt die Ausführung des biogeochemischen Modells noch viele weitere Optimierungen zu: Im Moment erfolgt die Verteilung der Profile auf die einzelnen CUDA Blöcke noch gleichmäßig und unabhängig von der Länge des Profils. Da die Profile sich aber teilweise stark in ihrere Länge unterscheiden, ergibt dies eine inhomogene Verteilung der Rechenlast auf die einzelnen Threads (wobei ein Thread genau ein Profil bearbeitet) innerhalb der Blöcke. Dadurch ist

es möglich, dass ein Multiprozessor der Grafikkarte für einige Zeit unterbenutzt sein kann, während er auf den Abschluss der Berechnung eines längeren Profils wartet. Eine mögliche Optimierung an dieser Stelle wäre das Zusammenlegen von mehreren kleinen Profilen zur Bearbeitung von einem Thread. Durch eine geeignete Kombination wäre es möglich, die Laufzeit der Threads innerhalb eines Blockes anzugleichen, um damit Fälle wie den oben genannten auszuschließen. Allerdings muss dafür der Scheduler, welcher von CUDA verwendet wird, weiter untersuchen werden, da dieser bereits versucht, innerhalb eines Multiprozessors durch gleichzeitige Ausführung mehrerer Warps für eine volle Auslastung zu sorgen. Eine naive Herangehensweise kann hier leicht zu einer Verlangsamung statt der erwünschten Beschleunigung führen.

Weiterhin wird bisher nur die langsamste Variante der Grafikspeicher - der globale Speicher - ausgenutzt. Indem der geteilte Speicher (Kapitel 2.1) dazu verwendet wird, die Speicherzugriffe innerhalb des biogeochemischen Modells auf den schnellen geteilten Speicher zu beschränken, wäre es möglich, bei komplexen Modellen weiterhin große Geschwindigkeitsgewinne zu erzielen, da dieser Speicher im Vergleich zum globalen Speicher um mehrere Faktoren schneller ist. Da die Größe dieses Speichers aber limitiert ist, würde diese Änderung unter Umständen weitere Limitierungen für das Fortranmodell erzwingen.

Bisher findet die gesamte Simulation nur auf einer Grafikkarte statt. Allerdings spricht nichts dagegen die Simulation auf zwei Grafikkarten auszudehnen. PETSc unterstützt über OpenMPI bereits die Zusammenarbeit mehrerer Prozesse, und PETSc GPU implementiert weiterhin bereits einige nötige Funktionen zur Segmentierung der CUSP Matrix- und Vektorimplementierungen. Was in diesem Fall zu untersuchen wäre, ist die genaue Funktionsweise von OpenMPI und die Anforderungen die es an die Strukturen (Matrix, Vektor) stellt. Die Angabe der zu verwendenen Grafikkarte erfolgt in PETSc GPU über den Parameter `-cuda_set_device`. Dieser müsste also beim Aufruf von OpenMPI für jeden Prozess einzeln angegeben werden.

Die Migration auf die Grafikkarte zeigt, dass der Sättigungseffekt, welcher bei CPU Clustern zu erkennen ist, mit spezialisierter, aber vergleichbar günstiger Hardware umgangen werden kann, und es dennoch möglich ist, die in Metos3D verwendbaren Klimamodelle mit immer höheren Geschwindigkeiten zu simulieren. Selbst mit einer vergleichbar günstigen, handelsüblichen Grafikkarte konnte eine Geschwindigkeit erreicht werden, welche bei einem CPU Cluster erst mit fast 60 CPUs erreicht wird (siehe Abbildung 4.5). Da bei mehreren GPUs diese aber über die CPU kommunizieren müssen, ist bei der Verwendung mehrerer Grafikkarten allerdings ein geringerer Beschleunigungseffekt zu erwarten. Direkte

Kommunikation zwischen Grafikkarten (z.B. SLI) wäre eine mögliche Lösung dieses Problems, erfordert aber womöglich tiefgreifendere Änderungen und Erweiterungen an der PETSc Bibliothek.

## Danksagung

Im Abschluss dieser Arbeit möchte ich mich insbesondere bei meinem Betreuer Jaroslaw Piwonski bedanken, welcher mir während der Arbeit mit Hilfestellungen und Orientierung Richtung Ziel zur Seite stand. Weiterhin bedanke ich mich bei Alexander Koch für das Korrekturlesen der Arbeit.

Einen herzlichen Dank auch an die Mitarbeiter des Lehrstuhls Kommunikationssysteme von Prof. Dr.-Ing. Norbert Luttenberger, insbesondere Matthias Westphal und Hagen Peters für die Bereitstellung eines Rechnerplatzes mit der nötigen Hardware.

## Literatur

Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

Satish Balay, Jed Brown, , Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011a.

Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2011b. http://www.mcs.anl.gov/petsc.

Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: http://doi.acm.org/10.1145/1654059.1654078.

Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. URL `http://www.meganewtons.com/`. Version 1.3.0.

Samar Khatiwala. A computational framework for simulation of biogeochemical tracers in the ocean. *Global Biogeochemical Cycles*, 21(3):1–14, July 2007. ISSN 0886-6236. doi: 10.1029/2007GB002923. URL `http://www.agu.org/pubs/crossref/2007/2007GB002923.shtml`.

Samar Khatiwala, Martin Visbeck, and Mark a. Cane. Accelerated simulation of passive tracers in ocean circulation models. *Ocean Modelling*, 9(1):51–69, January 2005. ISSN 14635003. doi: 10.1016/j.ocemod.2004.04.002. URL `http://linkinghub.elsevier.com/retrieve/pii/S1463500304000307`.

Sören Mahmens. Implementierung von 3D-Simulationsalgorithmen auf Grafikkarten. 2011.

V. Minden, B.F. Smith, and M.G. Knepley. Preliminary implementation of PETSc using GPUs. *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, 2010.

NVIDIA. CUDA C Best Practices Guide. 2011a. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf`.

NVIDIA. CUDA C Programming Guide. 2011b. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

NVIDIA. CUDA-GDB: The NVIDIA CUDA Debugger. 2008. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/cuda-gdb.pdf`.

Jaroslaw Piwonski and Thomas Slawig. The Idea and Concept of Metos3D - A Marine Ecosystem Toolkit for Optimization and Simulation in 3-D. Technical report, CAU Kiel, 2010.

The Portland Group. PGI Compiler Reference Manual. 2011a. URL `http://www.pgroup.com/doc/pgiref.pdf`.

The Portland Group. CUDA Fortran Progamming Guide and Reference. 2011b. URL `http://www.pgroup.com/doc/pgicudafortug.pdf`.

The Portland Group. PGI Fortran Reference. 2011c. URL `http://www.pgroup.com/doc/pgifortref.pdf`.