Diplomarbeit

Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fachbereich Informatik, Mathematik und Naturwissenschaften
Studiengang Informatik

Vergleich von APIs für Paralleles Rechnen auf Grafikkarten und Implementation auf Octave

Zur Erlangung des akademischen Grades
Diplominformatiker (FH)
vorgelegt von

Ronny Lindner

Prüfer HTWK-Leipzig: Herr Prof. Dr. rer. nat. Klaus Hering
Prüfer Max-Planck-Institut: Herr Dipl. Inf (FH) Andreas Romeyke

02.03.2009

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig und ohne unerlaubte Hilfe angefertigt, diese noch nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Application Programming Interface (API)
Schnittstelle für Programmierer zu einem Gerät oder einer Software.

arithmetisch-logische Einheit (ALU)
Arithmetisch-logische Einheiten sind Bereiche von Computerchips, die auf arithmetische Berechnungen spezialisiert sind. Beispiele sind Addition, Multiplikation und Vergleiche.

ATI
ATI war neben NVIDIA der größte Hersteller von Grafikkarten. Der Computerchip-Hersteller AMD kaufte die Firma 2006, der Name ATI wurde für Grafikkarten allerdings beibehalten.

Automatically Tuned Linear Algebra Software (ATLAS)
Eine alternative Implementation von BLAS mit automatischer Optimierung für die genutzte Hardware.

Basic Linear Algebra Subprograms (BLAS)
Referenzimplementation einer Bibliothek mit Funktionen der linearen Algebra.

bitonisch
Eine bitonische Folge besteht aus einer monoton steigenden und einer monoton fallenden Folge (oder umgekehrt).

Central Processing Unit (CPU)
Die CPU ist der Hauptprozessor des Computers, der für allgemeine Berechnungen zuständig ist und auf dem das Betriebssystem und alle Programme ausgeführt werden.

Datenabhängigkeit
Eine Datenabhängigkeit tritt auf, wenn beispielsweise das Ergebnis einer Berechnung bei einer nachfolgenden Berechnung genutzt wird. Beide Berechnungen können nicht parallel stattfinden, da der zweiten ein Eingabewert fehlt.

DirectX
API für Grafikanwendungen unter Microsoft Windows, vgl.: OpenGL.

dynamische Bibliothek
Maschinencode in einer Datei, der von anderen Programmen ausgeführt werden kann. Wird meist genutzt, wenn in mehreren Programmen dieselbe Funktionalität genutzt wird oder wenn ein Programm mit Zusatzfunktionalität ausgestattet werden soll. Unter Windows haben diese Dateien meist die Endung `.dll` (dynamically linked library) und unter UNIX-artigen Systemen `.so` (shared object).

Elektroenzephalografie (EEG)
Methode zur Messung der Hirnströme mittels auf dem Kopf angebrachter Elektroden.

`float`
Datenformat für eine Fließkommazahl.

`float4`
Ein Datenformat für vier zusammengefasste Fließkommazahlen. Es wird vor allem von Grafikkarten genutzt. Diese haben für die Grundrechenarten und einige weitere Befehle spezielle Versionen für je vier Ein- und Ausgabewerte.

Gaußhaftigkeit
Die Gaußhaftigkeit bezeichnet die Ähnlichkeit einer Datenmenge zur Normalverteilung, die auch Gauss-Verteilung genannt wird. Siehe Kurtosis.

General-Purpose computation on GPUs (GPGPU)
Bezeichnet das Rechnen mit allgemeinen Daten auf Grafikkarten. Hierbei werden die Vertex- und Pixelshader als Recheneinheiten genutzt, welche normalerweise die später auf dem Bildschirm angezeigten Bilder erzeugen.

GNU Compiler Collection (GCC)
Die GCC ist eine Sammlung von freien Compilern für viele Programmiersprachen (C, C++, Objective-C, Fortran, Java, Ada). GNU ist ein rekursives Akronym für "GNU's Not Unix". Das bezieht sich auf das GNU-System (GNU/Linux), welches als Alternative für UNIX entwickelt wurde.

Hyperthreading
Hyperthreading ist ein Konzept zum Beschleunigen von Berechnungen bei Intel-Prozessoren. Da nicht alle Teile des Prozessors gleichmäßig ausgelastet waren, wurde eine Technik hinzugefügt, die bei Wartezeiten eine andere Anweisung ausführt

und somit die Wartezeit überbrückt.

in-place
Beschreibung eines Algorithmus, der nur den Speicherplatz für die Eingangsdaten und eine konstante Menge Zwischenspeicher benötigt. Die Ergebnisse werden dabei auf den Speicherplatz geschrieben, welchen zuvor die Eingangsdaten belegten.

Independent Component Analysis (ICA)
Methode der Komponentenanalyse, siehe Abschnitt 4.2 auf Seite 17.

Knoten
Ein paralleles System besteht meist aus vielen parallel arbeitenden Einzeleinheiten, diese Einheiten werden wie in einem Netz(werk) als Knoten bezeichnet.

Kurtosis
Die Kurtosis ist eine Funktion zum Vergleichen von Daten mit der Normalverteilung. Die Normalverteilung ist bei einer Kurtosis von drei erreicht, größere Werte entstehen bei einer spitzeren Kurve, kleinere bei einer flacheren Kurve.

Mergesort
Mergesort ist ein schnelles Sortierverfahren. Eine Variante des Verfahrens wird in Abschnitt 4.4 auf Seite 20 vorgestellt.

Mischungsmatrix
Mit einer Mischungsmatrix können Eingabedaten gemischt werden, beispielsweise könnte man mit Hilfe eines Alphabets und mehreren Mischungsmatrizen Wörter bilden:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} A \\ C \\ D \\ C \end{pmatrix}$$

Für die Elemente der Mischungsmatrix sind beliebige Fließkommawerte möglich, wodurch Eingabewerte negiert, abgeschwächt oder verstärkt werden können. Wenn die Mischungsmatrix in einer Zeile mehrere Werte enthält, dann werden an dieser Stelle der Ausgabematrix mehrere Eingabeelemente gewichtet miteinander verknüpft, woher die Mischungsmatrix ihren Namen hat..

Multiple Instruction, Multiple Data (MIMD)
MIMD beschreibt eine Parallelität, wie sie beispielsweise mit Mehrkernprozessoren nutzbar ist: unterschiedliche Anweisungen können mit anderen Daten parallel

ausgeführt werden.

nicht-negative Matrizenfaktorisierung (NMF)
Algorithmus zum zerlegen von Matrizen (zum Beispiel Bilder) in einzelne Komponenten, siehe Abschnitt 4.3 auf Seite 19.

Groß-O-Notation ($\mathcal{O}$)
Die Groß-O-Notation ist eine obere Laufzeitabschätzung für Algorithmen. Die danach in Klammern angegebene Formel ist eine Obergrenze für die Anzahl Durchläufe, die der Algorithmus bei einer Datengröße von n hat. Das Suchen eines Wertes in einer ungeordneten Liste der Größe n hat beispielsweise eine Laufzeit $\mathcal{O}(n)$, da maximal alle n Elemente geprüft werden müssen.

Octave
Programmiersprache, siehe: Abschnitt 5.1.

Open Audio Library (OpenAL)
API für Soundeffekte, entwickelt von der Khronos-Group.

Open Computing Language (OpenCL)
GPGPU-Bibliothek der Khronos-Group.

Open Graphics Library (OpenGL)
API für Grafikanwendungen, siehe: Abschnitt 2.5.

paralleles Rechnersystem
Ein Rechnersystem besteht aus einer bestimmten Hardware und kompatibler Systemsoftware. Unterschiedliche Softwarekomponenten (etwa andere Laufzeitbibliotheken oder Betriebssysteme) führen zu unterschiedlichen parallelen Rechnersystemen.

PCI Express (PCIe)
PCI Express ist eine Punkt-zu-Punkt-Verbindung zwischen Computerkomponenten. Diese besteht aus bidirektionalen Leitungen, auch Lanes genannt, zwischen je einer Komponente und dem Motherboard. Die Übertragungsgeschwindgkeit pro Lane und Richtung beträgt maximal 250 MB/s bei PCIe 1.0 und 500 MB/s bei PCIe 2.0. Ein Gerät kann 1, 2, 4, 8, 16 oder 32 Lanes für eine schnellere Datenübertragung nutzen. Grafikkarten benutzen 16 Lanes, was bei neueren Karten mit PCIe 2.0 zu einer Geschwindigkeit von 8 GB/s führt. PCI Express wurde als Nachfolger von PCI-Bus und AGP entwickelt und hat inzwischen AGP (die bisherige

Anbindung für Grafikkarten) verdrängt.

Pixelshader
Ein Pixelshader ist eine programmierbare Einheit der Grafikkarte zum Modifizieren von Pixelfarben bei 3D-Darstellungen.

rendern
Das Rendern beschreibt das Zusammensetzen aller vorgegebenen Einzelteile zu einem Ganzen. Der Ausdruck wird hauptsächlich in der Grafikverarbeitung genutzt, wo er das Zusammensetzen aller (3D-)Objekte, Lichter und sonstiger Elemente zu einem fertigen Bild beschreibt.

Runtime
API zur Anbindung eines Programms an verschiedene Hardwareschnittstellen, bei BrookGPU etwa zur Nutzung von OpenGL, DirectX oder CPU.

Shader
Ein Shader bezeichnet einerseits ein kurzes Programm, welches auf der Grafikkarte ausgeführt wird, andererseits wird er auch als Oberbegriff für die Hardwareeinheit genutzt, welche die vorgenannten Programme ausführt. Beispiele sind Pixelshader und Unified Shader.

Single Instruction, Multiple Data (SIMD)
SIMD ist eine Parallelisierungsart, bei der dieselbe Anweisung parallel mit vielen verschiedenen Daten ausgeführt wird.

Single Program, Multiple Data (SPMD)
SPMD ist eine Stufe zwischen SIMD und MIMD, bei der Verzweigungen unterstützt werden, so dass ganze Programme parallel laufen können. Es ist im Gegensatz zu MIMD nicht möglich, verschiedene Programme gleichzeitig auszuführen.

Skalarprodukt
Das Skalarprodukt, auch Punktprodukt genannt, ist eine mathematische Operation, mit der man unter anderem bestimmen kann, ob zwei Vektoren orthogonal zueinander sind, also senkrecht aufeinander stehen. Dafür wird jedes Element eines Vektors mit dem dazugehörigen Element des anderen Vektors multipliziert und die Ergebnisse summiert:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = 1 * 4 + 2 * 5 + 3 * 6 = 32$$

Das Verfahren wird auch beim Multiplizieren von Matrizen angewandt.

Speedup

Maßangabe für die Beschleunigung eines Algorithmus bei Nutzung mehrerer Recheneinheiten p und der bei n Recheneinheiten benötigten Zeit T_n, berechnet über die Formel $S_p = \frac{T_1}{T_p}$. Der Speedup ist nach dem Amdahlschen Gesetz (siehe Abschnitt 2.1 auf Seite 3) durch die Größe des sequentiellen Anteils im Algorithmus begrenzt.

Standardabweichung

Ein Maß für die Schwankung von Messergebnissen. Etwa 68 % aller gemessenen Werte liegen innerhalb der einfachen Standardabweichung. Aus einer hohen Standardabweichung folgt eine starke Streuung der Messwerte.

Stream

Ein Stream ist eine listenähnliche Datenstruktur, in der beliebig viele Daten — bei den meisten Grafikkarten aktuell auf 4096 Elemente pro Dimension des Streams beschränkt — gleichen Typs gespeichert werden können. In Grafikkarten werden meist zweidimensionale Streams genutzt, da sich hiermit zu berechnende Bilder leicht abbilden lassen, indem ein Datum des Streams dem rot-, grün- und blau-Anteil eines Pixels entspricht.

Streaming SIMD Extensions (SSE)

SSE ist eine Befehlssatzerweiterung von Prozessoren, mit der eine SIMD-Parallelisierung im Prozessor ermöglicht wird. Die Register für Ein- und Ausgabe sind jeweils 128 Bit (16 Byte) breit und erlauben je nach Datentyp bis zu 16 parallele Berechnungen.

Subversion (SVN)

Subversion ist ein Versionsverwaltungssystem für Software. Es verwaltet den Quelltext aller Versionen eines Programmes und bietet so einerseits Einblick in die Evolution der Software und andererseits die Möglichkeit, alte Versionen wiederherzustellen.

Textur

Eine Textur ist ein Bild, das ähnlich einer Folie um ein 3D-Objekt gelegt wird. Somit entsteht am Computer beispielsweise aus einem Zylinder und dem Bild einer Baumrinde ein Baumstamm.

Texture Fetch

Als Texture Fetch wird das Lesen von Farbdaten aus einer Textur innerhalb der

Grafikkarte bezeichnet.

Thread

Ein Thread wird allgemein als Teilablauf eines Programmes gesehen (wörtlich übersetzt: Faden → roter Faden durch das Programm). Bei Grafikkarten wird damit speziell die Ausführung eines Shader-Programmes mit einem Datum des Eingabestreams bezeichnet.

Topologie

Die Topologie beschreibt den Aufbau eines Netzwerks. Ein Beispiel ist die Sternstruktur. In diesem Fall werden Geräte direkt an einem zentralen Verteiler angeschlossen. Eine weitere Möglichkeit ist die Busstruktur, bei der alle Geräte an einem Kabel angeschlossen sind und Daten bei allen Geräten eintreffen.

Unified Shader

Grafikkarten bestehen aus mehreren Recheneinheiten mit unterschiedlichen Einsatzgebieten: Pixelshader, Vertexshader und ab DirectX 10 auch Geometrieshader. Seit DirectX 10 unterstützen alle Einheiten die gleichen Befehle, weshalb die hardwareseitige Unterteilung in unterschiedliche Recheneinheiten nicht mehr nötig ist. Ein Unified Shader kann je nach anstehender Aufgabe die Arbeit der vorher genannten Einheiten ausführen. Dies führt zu einer besseren Auslastung der Grafikkarte.

Vertex (Plural: Vertices)

3D-Objekte im Computer bestehen meist aus Dreiecken, die aus je drei Vertices, den Eckpunkten des Dreiecks, zusammengesetzt sind. Jeder Vertex hat eine Position im Raum, eine Farbe an dieser Position und Texturkoordinaten für eine auf das Dreieck gelegte Textur [1].

1 Einleitung

1.1 Motivation

Computer werden immer leistungsstärker, dennoch benötigen viele Programme Stunden oder Tage, um ein Ergebnis zu berechnen. Diese Algorithmen bestehen häufig aus Operationen, die parallel ausgeführt werden könnten, allerdings unterstützen selbst aktuelle Central Processing Units (CPUs) eine parallele Ausführung nur sehr begrenzt.

Grafikkarten bieten durch ihren ursprünglichen Einsatzzweck, der Berechnung von vielen Bildpunkten in möglichst kurzer Zeit, ein hohes Parallelitätspotential. Aufgrund der immer höheren Anforderungen von Computerspielen wurde die Leistung der Grafikkarten in den letzten Jahren beträchtlich erhöht. Die Entwicklungen in der Vergangenheit haben gezeigt, dass die Grafikkarte auch für allgemeine Berechnungen gut geeignet ist und diese schneller ablaufen können als auf der CPU [2]. Wäre es möglich, Algorithmen einfach für die Grafikkarte zu portieren, könnte mit niedrigem Budget das Ergebnis für zeitkritische Berechnungen schneller zur Verfügung stehen.

1.2 Aufgabenstellung und Ziele

In dieser Arbeit soll die Kompatibilität der Programmiersprache Octave mit verschiedenen Bibliotheken zum Rechnen auf der Grafikkarte untersucht werden. Octave wurde gewählt, da diese Programmiersprache die Beschreibung von parallelen Berechnungen vereinfacht und weil sie häufig für wissenschaftliche Zwecke verwandt wird. Weiterhin soll geprüft werden, ob sich die Nutzung der Grafikkarte bei wissenschaftlichen Berechnungen lohnt. Es soll dennoch keinen Vergleich zwischen mehreren verschiedenen Grafikkarten geben, da derartige Tests über den Umfang der Arbeit hinausgehen würden und zudem bereits in Fachzeitschriften und dem Internet verfügbar sind.

1.3 Vorgehen

Nach einer gewissen Vorarbeit werden einige Algorithmen mit der ausgewählten Bibliothek implementiert und auf der Grundlage der gemachten Erfahrungen entschieden, ob sich die Bibliothek für den produktiven Einsatz mit Octave eignet. Dafür werden sowohl

die komfortable Benutzung als auch Geschwindigkeitsaspekte betrachtet. Eventuell auftretende Bugs werden behoben, sofern dies im gegebenen Zeitrahmen möglich ist.

1.4 Resultate

Es hat sich gezeigt, dass die Programmiersprache von BrookGPU zwar komfortables Arbeiten in einer gewohnten C-Umgebung ermöglicht, aber leider noch zu viele Bugs enthält. Einer dieser Fehler ist gar dafür ausschlaggebend, dass eine Verwendung von Octave in Verbindung mit BrookGPU aktuell nicht empfohlen wird. Beendet sich das Programm aufgrund des Problems, so gehen alle bisher in dieser Octave-Session berechneten Daten verloren. Das Zwischenspeichern der Werte ist unüblich, da die Berechnungen in dieser Programmiersprache sehr zuverlässig sind und nur sehr selten abstürzen. Die Geschwindigkeitstests haben dennoch gute Ergebnisse gezeigt. Es wäre sinnvoll, die noch bestehenden Bugs zu beheben, denn dann könnte man die Kombination aus BrookGPU und Octave gut für aufwändige Berechnungen verwenden.

1.5 Übersicht

Zu Beginn werden einige Grundlagen zum parallelen Rechnen geklärt. Unter anderem werden auch Grafikkarten allgemein betrachtet und deren Leistungsmerkmale beleuchtet. Dann werden einige Bibliotheken kurz vorgestellt, die das Rechnen auf der Grafikkarte vereinfachen sollen. Eine davon wird ausgewählt und genauer beschrieben. Daraufhin folgt die Erläuterung einiger Algorithmen, die eventuell auf der Grafikkarte schneller ausgeführt werden können. Deren Implementation wird erklärt und es wird grafisch dargestellt, wie die Berechnungszeit sich im Vergleich zur klassischen Implementation auf der CPU verändert. Die bei der Programmierung gewonnenen Kenntnisse werden im darauffolgenden Kapitel erläutert und geklärt, ob die ausgewählte Bibliothek für eine Verwendung mit Octave geeignet ist. Den Schluss bildet ein Kapitel mit der Zusammenfassung und dem Ausblick auf zukünftige Entwicklungen. Für das bessere Verständnis der Arbeit werden allgemeine Informatik- und Programmierkenntnisse vorausgesetzt.

1.6 Vereinbarung zu Schreibweisen

In der Arbeit werden Teile von Quelltexten vorkommen, die für eine bessere Unterscheidung zum normalen Text in einer `Monospace-Schriftart` geschrieben sind, selbiges gilt auch für `Dateinamen` und `Pfade`. Mathematische Symbole und Gleichungen werden in einer speziellen *Mathematikschreibweise* dargestellt. Die Arbeit enthält zu Beginn ein Glossar und ein Abkürzungsverzeichnis. Weiterführende Literatur wird in Klammern [3] angegeben.

2 Grundlagen

2.1 Parallelität in Hardware und Software

Bei der Parallelverarbeitung ist der Speedup ein wichtiger Aspekt. Das Amdahlsche Gesetz bietet eine grobe Abschätzung zum maximal möglichen Speedup eines Algorithmus [4]. In [5] wird es folgendermaßen beschrieben:

> Wenn bei einer parallelen Implementierung ein (konstanter) Bruchteil $f(0 \leq f \leq 1)$ sequentiell ausgeführt werden muss, setzt sich die Laufzeit der parallelen Implementierung aus der Laufzeit $f * T_1(n)$ des sequentiellen Teils und der Laufzeit des parallelen Teils, die mindestens $(1-f)/p * T_1(n)$ beträgt, zusammen. Für den erreichbaren Speedup gilt deshalb
>
> $$S_p(n) = \frac{T_1(n)}{f * T_1(n) + \frac{1-f}{p} T_1(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Wenn 10 % ($f = 0.1$) des Programmes sequentiell berechnet werden müssen, dann ergibt sich nach der obigen Abschätzung ein maximal möglicher Speedup von $1/0.1 = 10$ für den Fall, dass der parallele Teil unendlich schnell ausgeführt wird und folglich keine Zeit mehr benötigt.

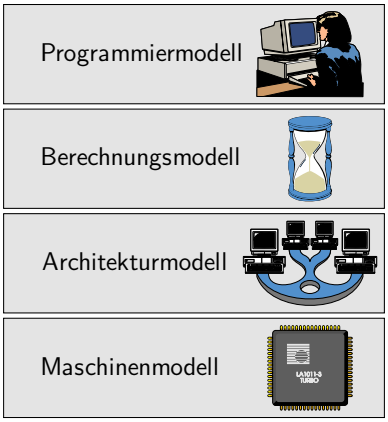

Abbildung 2.1: *Abstraktionsebenen paralleler Rechnersysteme*

Parallele Rechnersysteme können in unterschiedlichen Abstraktionsebenen betrachtet werden. In [6] werden vier Modelle vorgestellt. Die unterste Ebene enthält das Maschinenmodell, mit dem das Rechnersystem hardwarenah beschrieben wird. Es umfasst etwa

die Erläuterung der Register und Datenpfade eines Prozessors, die Eingabe- und Ausgabepuffer sowie die Datenverbindungen innerhalb eines Knotens. Im Architekturmodell wird die Topologie des Verbindungsnetzwerks und die Speicherorganisation beschrieben. Bei Letzterer wird geklärt, ob alle Knoten Zugriff auf einen großen Speicher haben oder ob es für jeden Knoten einen exklusiven Speicher gibt. Weiterhin wird in dieser Ebene die Art des parallelen Rechnersystems behandelt, wie beispielsweise Single Instruction, Multiple Data (SIMD) und Multiple Instruction, Multiple Data (MIMD). Das Berechnungsmodell umfasst die im System verfügbaren parallelen Operationen und die für die Berechnung benötigte Zeit. Schlussendlich wird im Programmiermodell die Sicht des Programmierers auf das System beschrieben, vor allem auf Schnittstellen, die dem Programmierer zur Verfügung stehen.

Zusammenfassend beschreiben diese Modelle, welche der oben genannten Parallelitätsebenen das System unterstützt, wie Informationen ausgetauscht werden und ob der Programmierer den Ablauf der Parallelität explizit festlegen kann. Außerdem werden mit den Modellen Synchronisationsmöglichkeiten beschrieben und geklärt, ob die Recheneinheiten zueinander synchron laufen.

Je nach Rechnersystem unterscheidet sich die Granularität der parallelen Bereiche. Es wird zwischen Parallelität auf Instruktionsebene, Anweisungsebene, Schleifenebene und Prozedurebene unterschieden. Bei der Parallelität auf Instruktionsebene werden verschiedene Anweisungen ohne Datenabhängigkeit gleichzeitig ausgeführt. Parallele Berechnungen auf Anweisungsebene arbeiten nach dem Prinzip von SIMD, wobei eine Anweisung für mehrere Daten gleichzeitig ausgeführt wird und erst am Schluss alle Ergebnisse abgespeichert werden. Die Parallelität auf Schleifenebene hat eine grobkörnigere Granularität als die beiden vorher genannten Ebenen. Hier finden die Durchläufe durch die Schleife parallel statt, wobei von Seiten des Programmierers auf Datenunabhängigkeit geachtet werden muss. Die Parallelität auf Prozedurebene wird auch Funktions- oder Taskparallelität genannt. Sie macht sich zunutze, dass häufig voneinander unabhängige Programmteile existieren, die parallel ausgeführt werden können.

2.2 Aufbau von Grafikkarten

Grafikkarten werden üblicherweise auf einem Board ausgeliefert, auf dem neben dem Grafikchip zusätzlich noch der Arbeitsspeicher und diverse kleinere Chips für die Kommunikation untergebracht sind. Aktuelle Grafikkarten sind per PCI Express (PCIe) mit den anderen Komponenten des Rechners verbunden, der bei der aktuellen Version 2.0 eine maximale Übertragungsgeschwindigkeit von 8 GB/s erreicht. Für den Arbeitsspeicher werden Chips mit besonders hoher Übertragungsrate hergestellt, da bei Grafikberechnungen große Datenmengen transferiert werden müssen.

In Abbildung 2.2 ist die Aufteilung des Grafikchip von dem Hersteller ATI gut sichtbar. Er besteht zu einem großen Teil aus Recheneinheiten, den Unified Shadern. Diese sind

für das Ausführen von Vertexshadern, Geometrieshadern und Pixelshadern zuständig. Dies sind die Aufgaben, die den größten Teil der Rechenzeit benötigen und von deren Beschleunigung der Gesamtprozess der Grafikberechnung sehr profitiert. Auf dem Grafikchip befinden sich zudem noch Kontrollbereiche für die Anbindung an den Arbeitsspeicher. Ein Bereich des Grafikchips ist für die Verbindung der Grafikkarte per PCIe reserviert. Ein weiterer wichtiger Teil des Grafikchips sind die Textur-Einheiten. Mit ihnen werden Texturen vorverarbeitet, bevor sie für Berechnungen an die Shader weitergegeben werden. Der letzte wichtige Bereich, der hier erläutert werden soll, ist für die Ausgabe des Bildes an einen angeschlossenen Monitor zuständig. Er übernimmt bei neueren Grafikkarten zusätzlich Kontrollaufgaben für das schnellere Anzeigen von hochauflösenden Videos. In den oben genannten Berechnungs- und Textureinheiten befinden sich mehrere Cache-Ebenen. Diese ermöglichen einen schnelleren Zugriff auf Daten, die im Voraus aus dem Arbeitsspeicher geladen wurden.

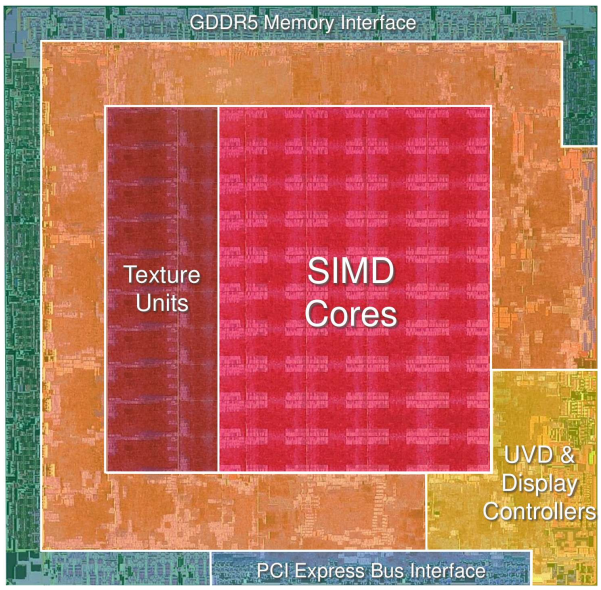

Abbildung 2.2: *Grafikchip der ATI Radeon HD 4800 Serie (Quelle: [7])*

Für einen Vergleich zwischen Grafikkarte und CPU wird der Aufbau der CPU hier kurz erklärt. In dieser werden Befehle nahezu sequentiell ausgeführt. Das Abarbeiten eines Befehls wird in mehrere Schritte wie beispielsweise Dekodierung, Berechnung und Speicherung unterteilt, die in mehreren Hardwarebereichen durchgeführt werden. Dies ermöglicht die parallele Verarbeitung mehrerer Befehle, so dass bei obigem Beispiel mit drei Stufen ein Speedup von drei möglich ist. Dieses Prinzip wird Pipelining genannt, da mehrere Befehle nacheinander wie in eine große Pipeline hineingeschoben werden, ohne dass alle vorhergehenden Befehle fertig bearbeitet wurden.

Im Gegensatz dazu arbeitet die Grafikkarte massiv parallel. Für eine Gruppe aus Unified Shadern wird ein Pool von zu berechnenden Threads bereit gehalten. Stockt die Verarbeitung einer Aufgabe, weil beispielsweise ein Texture Fetch nötig ist, so wird der

aktuelle Thread in den Cache verdrängt und der Nächste aus dem Cache geladen und gestartet. Sind die angeforderten Daten nach einer Zeit verfügbar, kann der alte Thread weiter berechnet werden.

Spezielle Verfahren wie das Pipelining und eine Neuanordnung der Befehle für eine optimalere Ausnutzung der Hardware — "out-of-order execution" genannt — sorgen für eine schnelle Abarbeitung mit Hilfe der CPU. Die Grafikkarte hat zwar wegen der vielen Recheneinheiten theoretisch eine höhere Leistung, der auszuführende Algorithmus muss allerdings extra für die vielen Recheneinheiten optimiert werden. Dieses Thema wird in Kapitel 5 behandelt.

2.3 Stand der Entwicklung

Die letzten Jahre über wurden Grafikkarten immer leistungsfähiger, aber verbrauchten dementsprechend mehr Strom. Über den PCIe-Steckplatz wird die Grafikkarte mit 75 Watt versorgt. Außerdem können je nach Bedarf ein oder zwei 6-Pin-Stecker an die Grafikkarte angeschlossen werden, womit zusätzlich je 75 Watt zur Verfügung stehen. Selbst 225 Watt reichten bald nicht mehr aus, weshalb in der PCIe-2.0-Spezifikation einer der 6-Pin-Stecker durch einen 8-Pin-Stecker mit 150 Watt ersetzt wurde. Dies führt zu einem maximal möglichen Verbrauch von 300 Watt [8]. Wegen einer vom Hersteller angegebenen maximalen Leistungsaufnahme von 236 Watt [9] benötigt eine NVIDIA GeForce GTX280 den 6-Pin- und den 8-Pin-Stecker des Netzteiles. Eine etwas langsamere ATI Radeon HD 4870 kommt mit zwei 6-Pin-Steckern aus und hat unter Last einen gemessenen Verbrauch von 170 Watt [10]. Ein kompletter Rechner mit den oben genannten Karten benötigt 360 bzw. 333 Watt, wovon die Grafikkarten über die Hälfte verbrauchen [11].

Derzeit dominieren die zwei Grafikkartenhersteller ATI und NVIDIA mit einer ähnlichen Produktpalette den Markt. Beide Hersteller bieten die Möglichkeit, dass mehrere Grafikkarten gleichzeitig genutzt werden. Bei ATI nennt sich diese Technik "CrossFire", bei NVIDIA handelt sich um "SLI". Hiermit ist es möglich, mehrere in einem Rechner verbaute Grafikkarten zu verbinden und deren gemeinsames Berechnungspotenzial für Grafiken zu nutzen. Außerdem existieren Lösungen mit zwei Grafikchips auf dem Board einer Grafikkarte. Verbindet man zwei dieser Doppel-Grafikkarten mit "SLI" oder "CrossFire", erhält man theoretisch die vierfache Leistung einer einzigen Grafikkarte. Nicht alle der in Kapitel 3 vorgestellten Bibliotheken unterstützen allerdings die gleichzeitige Berechnung mit mehreren Grafikkarten.

ATI und NVIDIA verfolgen unterschiedliche Strategien beim Aufbau ihrer Grafikkarten. Auf dem Chip der ATI Radeon HD 4870 sind zehn SIMD-Einheiten mit je 16 superskalaren Recheneinheiten untergebracht. Jede der 160 unabhängigen Recheneinheiten enthält fünf Arithmetisch-logische Einheiten, die je eine unabhängige Anweisung desselben Threads ausführen können. Bei Programmen mit wenigen unabhängigen Anweisungen

kann es daher vorkommen, dass nur ein Fünftel der 800 Unified Shader eine Berechnung durchführen. Die Gruppierung der Shader wird in der offiziellen Präsentation der Grafikkarte [7] auf Folie 7 dargestellt. Die 240 Unified Shader der NVIDIA GeForce GTX280 sind in 30 SIMD-Einheiten mit je 8 ALUs gruppiert. Die gruppierten ALUs bearbeiten bei NVIDIA aber nicht unabhängige Befehle desselben Threads, sondern unterschiedliche Threads. Dies führt bei genügend großen Streams zu einer besseren Auslastung der Shader [12].

In den letzten Jahren sank der Preis für große Monitore, deren Nachfrage deshalb stark anstieg. Die hohe Auflösung solcher Bildschirme verlangt den Grafikkarten immer mehr Leistung ab. Dafür muss nicht nur die Rechenkraft von Grafikkarten gesteigert, sondern in gleicher Weise der Arbeitsspeicher auf der Karte optimiert werden. Dies ist nötig, da Spielehersteller immer größere Texturen einsetzen und wegen den angesprochenen größeren Auflösungen, die pro berechnetem Bild mehr Speicher erfordern. Im Jahr 2006 wurden Grafikkarten meist mit 256 bis 512 Mebibyte Speicher ausgeliefert. Aktuelle enthalten sie üblicherweise ein Gibibyte Arbeitsspeicher. Auf Grafikkarten mit zwei Grafikchips ist die doppelte Menge Speicher verbaut, da jeder Chip eine eigene Kopie der Daten benötigt.

2.4 Potenzial für Parallelverarbeitung

Sehr gut für Parallelverarbeitung eignen sich Grafikkarten mit vielen Shadereinheiten und einem hohen Shadertakt und deshalb einer hohen Geschwindigkeit pro Einheit [12]. Diese Eigenschaften haben vor allem Karten, die für Computerspiele entwickelt wurden. Inzwischen gibt es auch spezielle GPGPU-Karten, die nur für Berechnungen gedacht sind und bis auf den fehlenden Anschluss für einen Monitor sehr ähnlich zu normalen Grafikkarten sind. Office-Grafikkarten oder besonders stromsparende Lösungen für mobile Geräte sind weniger gut geeignet. Sie erkaufen meist den geringeren Strombedarf mit einer niedrigen Leistung. Gänzlich ungeeignet sind ältere Grafikkarten ohne programmierbare Pixelshader, da diese für die Berechnung zuständig sind. Minimal sollte die Grafikkarte DirectX 9 unterstützen, denn ab dieser Version ist eine ausreichende Funktionalität sichergestellt. Ältere Grafikkarten unterstützen meist nur herstellerspezifische Programmiersprachen zum Ansteuern der Berechnungseinheiten. DirectX 10-kompatible Grafikkarten unterstützen einige für allgemeine Berechnungen hilfreiche Funktionen wie Ganzzahlberechnungen. Aus diesem Grund wird die Verwendung einer Grafikkarte mit mindestens DirectX 10-Unterstützung empfohlen.

Abgesehen von leistungsstarken Spiele-Grafikkarten stellen die Firmen ATI und NVIDIA seit 2006 GPGPU-Karten her, die nur für das Rechnen genutzt werden können. Intel, der dritte große Hersteller von Grafik-Lösungen, stellt nur Grafikchips her, die in das Motherboard integriert werden. Sie sind hauptsächlich für den Einsatz in mobilen Rechnern gedacht und bieten nur eine niedrige Leistung. Somit beschränkt sich das Angebot an Hochleistungsgrafikkarten auf die Produkte der ersten zwei Anbieter.

Wie oben bereits erwähnt, ist die Anzahl der Shader und deren Takt wichtig für die Gesamtgeschwindigkeit. Ein kleiner Vergleich soll die theoretische Leistung von Grafikkarte und Hauptprozessor gegenüberstellen. Der aktuell[1] schnellste Prozessor ist der Intel Core i7 965 XE [13] mit 3200 MHz Taktrate, vier Prozessorkernen und Hyperthreading. Obwohl mit der Nutzung von Hyperthreading keine einhundertprozentige Geschwindigkeitssteigerung erreicht werden kann, wird dies hier der Einfachheit halber angenommen. Die Taktrate aller Kerne zusammen beträgt somit 2 * 4 * 3,2 GHz = 25,6 GHz bei einem Preis von aktuell etwa 1000 €[2].

Die NVIDIA GeForce GTX 295 ist momentan eine der besten Grafikkarten auf dem Markt. Dies ist eine Karte mit zwei Grafikchips, einer Taktrate von 1242 MHz und je 240 Recheneinheiten. Das ergibt insgesamt eine theoretische Geschwindigkeit von 2 * 240 * 1,242 GHz = 596,16 GHz. Der aktuelle Preis beträgt knapp 500 €[3]. Die theoretische Geschwindigkeit der Grafikkarte ist über 23 mal so hoch wie beim aktuell schnellsten Hauptprozessor und die Hardware kostet nur die Hälfte.

Die Berechnung ist sehr einfach gehalten und bietet nur eine grobe Abschätzung der erreichbaren Geschwindigkeit. Weder wurde beim Hyperthreading mit den korrekten niedrigeren Werten gerechnet, noch wurde der Verwaltungsaufwand bei Berechnungen auf der Grafikkarte berücksichtigt. Normalerweise müsste sich solch ein Vergleich hauptsächlich auf die Fließkommarechenleistung stützen, die hier nicht beachtet wurde. Weiterhin ist zu bedenken, dass der Befehlssatz der CPU größer ist als der einer Grafikkarte, so dass manche Befehle auf dem Hauptprozessor in einem Schritt berechnet werden können, während die Grafikkarte mehrere Anweisungen ausführen muss. Wie schnell beide Recheneinheiten wirklich sind, wird Kapitel 5 zeigen.

Die Grafikkarte ist eine spezialisierte Hardware mit Parallelität auf Anweisungsebene und einem enormen Parallelisierungspotenzial. Allerdings bietet sie nur wenige Ein- und Ausgabemöglichkeiten. Beim normalen Ablauf werden Grafiken (Texturen) und Objektdaten an die Grafikkarte übergeben, dann werden Programme zum Modifizieren der Daten auf dieser ausgeführt (gerendert) und das Ergebnis von der Karte an den Monitor weitergegeben. Während dieser Zeit liegen die Daten durchgängig auf der Grafikkarte und müssen nicht zum Hauptprozessor übertragen werden. Es existieren dennoch Möglichkeiten, die berechneten Daten ohne Anzeige auf dem Monitor wieder für die CPU verfügbar zu machen. Dafür beschränkt man die Objekte meist auf ein Rechteck, das den gesamten sichtbaren Bereich der virtuellen 3D-Welt in der Grafikkarte einnimmt und legt eine Textur darüber. Diese Textur besteht aus den zu berechnenden Daten, die vorher an die Grafikkarte übergeben wurden. Das Ergebnis der Berechnung wird nun in den Backbuffer — einen Bildspeicher in der Grafikkarte — geschrieben. Normalerweise würde dieser im nächsten Schritt mit dem Frontbuffer getauscht, der das gerade auf dem Monitor sichtbare Bild enthält. Diesen Schritt verhindert man, indem stattdessen der Backbuffer gesperrt wird und die CPU die Ergebnisse ausliest.

[1] Stand: Februar 2009
[2] Alternate.de, 02.02.2009: 979 €
[3] Alternate.de, 02.02.1009: 444 € bis 539 €

2.5 OpenGL und DirectX

Die Open Graphics Library (OpenGL) bietet eine herstellerunabhängige Schnittstelle zur Grafikkarte. Ohne diese Abstraktionsschicht müsste der Grafikkartentreiber direkt angesprochen werden, was bei den vielen verschiedenen Grafikkarten schwierig wäre. Es ist mit einfachen Mitteln möglich, auf verschiedenen Betriebssystemen wie zum Beispiel Windows oder Linux 3D-Objekte hardwarebeschleunigt darzustellen. OpenGL ist der Nachfolger von Grafikbibliotheken, die in SGI Workstations genutzt wurden. Aus diesem Grund wird es auch heute noch häufig im professionellen Sektor benutzt, zum Beispiel bei Animationsprogrammen wie Blender [14].

Eine weitere Schnittstelle zur Grafikkarte bietet DirectX, ein Application Programming Interface (API) von Microsoft, das speziell für die Grafikdarstellung in Computerspielen entwickelt wurde. Es ist nur für Windows verfügbar und wird bei der Programmierung von Computerspielen aktuell sehr häufig eingesetzt. Bei einigen Spielen kann man wählen, welche API für die Grafikberechnungen eingesetzt werden soll, da dies die Portierung der Programme auf andere Betriebssysteme erleichtert. Aktuelle Grafikkarten unterstützen OpenGL und DirectX gleichermaßen.

2.6 Eingesetzte Grafikkarten

Für die Experimente in dieser Arbeit wurden zwei verschiedene Grafikkarten eingesetzt. Zum einen die NVIDIA GeForce 7900GTX, die Anfang 2006 als Oberklassemodell erschien und maximal DirectX 9 unterstützt. Diese wird hier für die meisten Berechnungen verwendet. Laut [15] hat sie 24 Pixelshader und acht Vertexshader, die zur Berechnung genutzt werden können. Die Leistung dieser Karte befindet sich, verglichen mit aktuell verkauften Grafikkarten, im Mittelfeld. Sie hat genügend Leistungsreserven für neuere Computerspiele, sofern diese nicht DirectX 10 benötigen. Die zweite Grafikkarte ist eine NVIDIA GeForce GTX280, die freundlicherweise vom Max-Planck-Institut für Kognitions- und Neurowissenschaften in Leipzig zur Verfügung gestellt wurde. Der Marktstart dieser Grafikkartengeneration war im Juni 2008. Sie ist nach wie vor eine der leistungsstärksten Grafikkarten und unterstützt DirectX 10. Auf der offiziellen Webseite des Herstellers [9] erfährt man, dass diese Karte über 240 Unified Shader verfügt, was im Vergleich zu den oben genannten 32 Shadern eine enorme Steigerung ist.

3 Bibliotheken

In diesem Kapitel werden Softwarelösungen besprochen, mit denen die Grafikkarte als Rechenwerkzeug genutzt werden kann. Ein Hauptkriterium für die Wahl der hier später genauer betrachteten Bibliotheken ist deren Lizenz. Die Programmiersprache Octave steht unter der GPL [16], einer strikten Lizenz, die besagt, dass eine GPL-lizenzierte Programmbibliothek nur in Programmen genutzt werden darf, wenn diese unter einer GPL-kompatiblen [3] Lizenz stehen. Da Octave gemeinsam mit einer der Bibliotheken ausgeführt werden soll, ist die Lizenz ein wichtiges Ausschlusskriterium.

3.1 Übersicht

Für die Berechnung von Algorithmen mit der Grafikkarte werden Funktionen genutzt, die eigentlich für die Beschleunigung von Grafikberechnungen entwickelt wurden. Häufig werden die APIs OpenGL und DirectX für die Ansteuerung der Grafikkarte genutzt. Möchte ein Programmierer die Beschleunigung der Grafikkarte für einen Algorithmus nutzen, benötigt er nicht nur allgemeine Programmierkenntnisse und das Wissen zu dem Fachgebiet, aus dem der Algorithmus stammt, sondern muss sich zusätzlich mit einer der oben genannten APIs auskennen. Das umfangreiche benötigte Wissen erschwert den Einstieg in die GPGPU-Programmierung. Deshalb wurden in den letzten Jahren einige Bibliotheken geschrieben, welche die grafikkartenspezifischen Befehle in sich kapseln. Hiermit ist es dem Programmierer einfacher möglich, Algorithmen mit Hilfe der Grafikkarte zu berechnen.

Die Bibliothek BrookGPU ist ein Forschungsprojekt des Stanford University Graphics Lab. Es befindet sich aktuell[1] im Beta-Stadium, wobei die aktuellste Version im November 2007 veröffentlicht wurde. Ein Teil des Projektes steht unter der modifizierten BSD-Lizenz und der Rest unter GPLv1+[2]. In BrookGPU kann man vor dem Ausführen des Programmes wählen, welche der vorgenannten APIs genutzt werden soll. Weiterhin ist es möglich, das Programm ohne Grafikkartenunterstützung auszuführen, was aber wegen der häufig langen Ausführungszeit nur für Testzwecke empfohlen wird.

Ein weiteres Forschungsprojekt ist Sh [17]. Es wurde an der Universität von Waterloo entwickelt und steht unter der LGPLv2.1+[3]. Es unterstützt das Rechnen auf der Grafik-

[1] Stand: Februar 2009
[2] GPL Version 1 oder neuer
[3] LGPL Version 2.1 oder neuer

karte und zusätzlich Funktionen für die Grafikverarbeitung, der ursprünglichen Aufgabe der Grafikkarte. Das Entwicklerteam von Sh entschied sich 2004 für den Ausbau des Programmes unter einer proprietären Lizenz. Sie gründeten eine Firma für die Weiterentwicklung und den Vertrieb der jetzt RapidMind [18] genannten Software. Seitdem wurde die Software um Komponenten wie etwa die Unterstützung für Cell-Prozessoren erweitert [19]. Im Jahr 2006 wurde die Weiterentwicklung von Sh eingestellt, kann aber weiterhin heruntergeladen werden.

Die Grafikkartenhersteller ATI und NVIDIA haben je eine Bibliothek erstellt, die nur mit den eigenen Grafikkarten funktioniert. Brook+ von ATI ist eine Weiterentwicklung von BrookGPU für Grafikkarten ab der mit DirectX 10 kompatiblen ATI Radeon HD2x00 [20]. Die Lizenz wurde von BrookGPU übernommen, demzufolge besteht Brook+ ebenfalls aus einer Mischung von BSD- und GPL-Lizenz. Im Gegensatz zu BrookGPU ist Brook+ nicht auf OpenGL oder DirectX angewiesen, da es direkt den Grafiktreiber anspricht und somit eine Abstraktionsschicht wegfällt.

Die Rechenbibliothek des Konkurrenten NVIDIA nennt sich CUDA [21]. Auch hier entfällt die Abstraktionsschicht der obigen Grafik-APIs und es werden alle mit DirectX 10 kompatiblen Grafikkarten von NVIDIA unterstützt. CUDA nutzt eine eigene Lizenz zur kostenfreien Weitergabe der Entwicklungsumgebung, ohne aber die Quelltexte offen zu legen.

CGiS [22] ist eine Software der Universität Saarbrücken. Sie ist momentan nicht im Internet verfügbar, da einige Programmteile noch auf einem frühen Entwicklungsstand sind. CGiS soll unter einer Open-Source-Lizenz erscheinen und dann parallele Berechnungen auf der Grafikkarte und spezielle CPU-Anweisungen wie Streaming SIMD Extensions (SSE) unterstützen.

BrookGPU unterstützt die Grafikkarten von den beiden großen Herstellern, ist recht aktuell und besitzt ein zu Octave passendes Lizenzmodell. Deswegen fällt die Entscheidung zu Gunsten dieser Bibliothek aus, die im nächsten Abschnitt genauer betrachtet wird. Danach folgt ein näherer Blick auf CGiS, das aber nicht ausgiebig getestet wurde, da es sich noch in Entwicklung befindet und aktuell nicht frei verfügbar ist.

3.2 BrookGPU

An der Stanford Universität wird seit 2003 BrookGPU entwickelt [23]. Diese Sprache basiert auf der Datenparallelisierung mittels Streams, die einem Vektor oder eine Matrix ähneln und deshalb mehrere Elemente beinhalten. Bei der Übergabe eines Streams an eine. Funktion wird die Funktion mit jedem Element des Streams parallel abgearbeitet. Dabei ist es nicht möglich, einzelne Elemente des Streams von der Berechnung auszuschließen. Es lassen sich aber Bereiche festlegen, in denen gerechnet werden soll. BrookGPU besteht aus zwei Teilen, einem Compiler und verschiedenen Runtimes. Der Compiler wandelt die Shader-Definitionen aus einer C-ähnlichen Sprache in C++ um.

Der Quelltext zum Starten der Anwendung und Bereitstellen der Daten kann entweder auch in dieser Sprache geschrieben werden oder er wird in einer externen C++-Datei abgelegt. Beide C++-Dateien werden dann beispielsweise mit der GNU Compiler Collection (GCC) kompiliert und ergeben eine ausführbare Datei. Setzt man die Umgebungsvariable `BRT_RUNTIME` auf einen der Werte `ogl`, `dx9`, `ctm` oder `cpu`, dann wird das BrookGPU-Programm mit eben dieser Schnittstelle ausgeführt. Bei `ogl` oder `dx9` wird OpenGL beziehungsweise DirectX genutzt, `ctm` ist die veraltete ATI-spezifische Version und mit `cpu` wird das Programm auf der CPU ausgeführt, was aber nur zu Testzwecken gedacht ist. Brook+ unterstützt die Runtimes `cpu` und `cal`, wobei Letzteres nur von DirectX 10-Karten von ATI unterstützt wird. Der Programmierer muss die Initialisierung von BrookGPU nicht explizit starten. Sie findet dann statt, wenn ein BrookGPU-Aufruf erfolgt. In einem Programm müssen zuerst Variablen für einen oder mehrere Streams definiert werden. Danach bindet man die Daten aus einem Feld mit Hilfe einer speziellen Kopierfunktion an den Stream. Der mit Daten gefüllte Stream kann nun an einen Shader übergeben werden, wobei der Shader als normale Funktion in C++ aufgerufen wird. Wahlweise kann der Stream, der als Ausgabestream eines Shaders genutzt wurde, als Eingabe für den Nächsten dienen. Sind alle Berechnungen abgeschlossen, so werden die Ergebnisse aus den Streams wieder in Felder kopiert. Wann die Daten wirklich zur Grafikkarte übertragen und die Ergebnisse berechnet werden, entscheiden das BrookGPU-Runtime und der Grafikkartentreiber. Dies erfolgt aber spätestens, wenn die Ergebnisdaten vom Programmierer angefordert werden. Es ist außerdem auch nicht erforderlich, dass die Daten erst zum Ende aller Berechnungen zum Hauptspeicher übertragen werden. Je nach Algorithmus kann es nötig sein, Zwischenergebnisse zu begutachten und beispielsweise beim Überschreiten eines Grenzwertes die Berechnung abzubrechen.

Der Shader in Listing 3.1 beschreibt die Multiplikation aller Elemente eines Streams mit einer Konstante, die vom Hauptprogramm aus übergeben wird. Im Funktionskopf sind die Variablen `f1` und `produkt` mit `<>` markiert, was sie als Streams kennzeichnet. `f2` ist kein Stream, sondern eine konstante Zahl, weswegen die Markierung an dieser Stelle fehlt. Listing 3.2 ist das Hauptprogramm, das die Daten verwaltet und BrookGPU-Funktionen aufruft. In den ersten zwei Zeilen werden Felder für Ein- und Ausgabedaten erzeugt. Das Schlüsselwort `BRTALIGNED` wird verwendet, damit Variablen an 16-Byte-Grenzen ausgerichtet werden. Dies ist für bestimmte Befehle der Runtimes nötig. In der fünften und sechsten Zeile werden die Streams definiert und in Zeile acht mit Daten gefüllt. Die Funktion `mult` ist ein Aufruf, der alle Daten mit dem obigen Algorithmus auf der Grafikkarte verarbeitet. Das Ergebnis liegt nun in `sfErgebnis` und wird in der letzten Zeile in den Hauptspeicher zurückgeschrieben. Die einzige Möglichkeit, die Parallelität der Grafikkarte auszunutzen besteht demnach darin, einen Algorithmus auf alle Elemente eines Streams gleichzeitig anzuwenden. Da von der Grafikkarte auch Verzweigungen und Schleifen unterstützt werden, handelt es sich hier um eine Single Program, Multiple Data (SPMD)-Parallelität [24].

In Abbildung 3.1 werden die Zeiten für den Transfer von Daten zwischen Grafikkarte und Hauptspeicher dargestellt. Dafür wurden zehn quadratische Matrizen gleicher Größe zur Grafikkarte übertragen, in einen anderen Stream kopiert und wieder zum Hauptspeicher

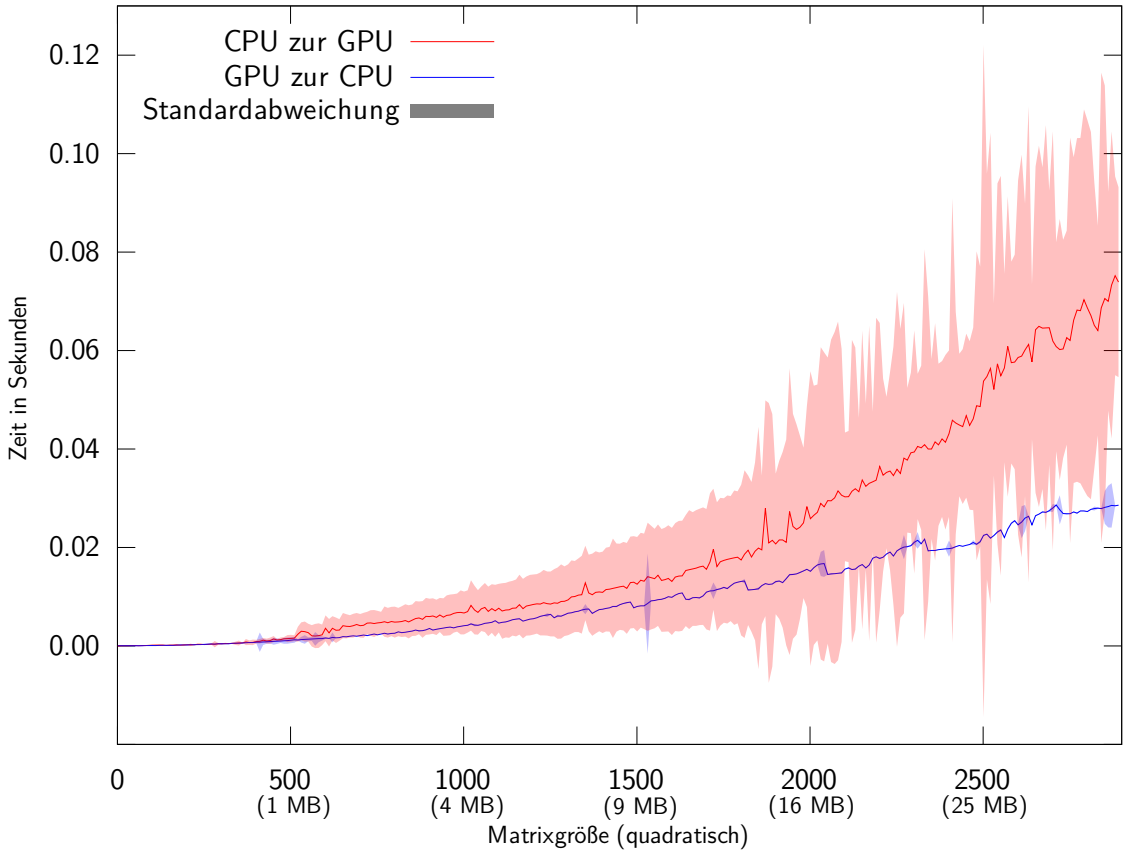

Abbildung 3.1: *Zeitmessung bei Datentransfers zwischen der CPU und der Grafikkarte mit Hilfe von quadratischen Matrizen (Median aus 10 Messungen)*

gesendet. Dabei wurde vor und nach den Befehlen `streamread` und `streamwrite` die Zeit gemessen und die Differenz gebildet. Es ist fraglich, ob die ermittelten Zeiten tatsächlich die reinen Datentransferzeiten sind, da mit PCIe 1.0 bis zu 4000 MB/s in jeder Richtung übertragen werden können. Das entspricht dem vier- bis achtfachen der gemessenen Werte. Es ist möglich, dass ein Teil der Initialisierung zu diesem Zeitpunkt stattfindet, was auch die hohe Standardabweichung der Übertragungszeit beim Transport zur Grafikkarte erklären könnte. Eine weitere Erklärung wären nötige Konvertierungen der Daten in das grafikkarteninterne Fließkommaformat.

In Anhang A ist eine Installations- und Einrichtungsanleitung für BrookGPU, da zu der im Internet verfügbaren Version kein Installationsprogramm mitgeliefert wird und einige Kompilierfehler mit neueren Compilern noch nicht behoben sind.

3.3 CGiS

CGiS ist ein Forschungsprojekt des Compiler Design Lab an der Universität Saarbrücken. Es ist aktuell in Entwicklung, so dass die in dieser Arbeit gezeigten Geschwindigkeitswerte sich bis zur Veröffentlichung des Programmes noch ändern können. CGiS stellt eine eigene Programmiersprache für die Nutzung der Grafikkarte zur Verfügung, die in drei Bereiche unterteilt ist. Listing 3.3 zeigt ein kleines CGiS-Programm, das zwei Vektoren addiert. Im Bereich `INTERFACE` werden die Streams deklariert, wobei festgelegt wird, welchen Datentyp die Elemente haben sollen und ob der Stream zum lesen, schreiben oder für beides verwendet werden soll. Im `CODE`-Abschnitt wird die eigentliche parallele Operation beschrieben, in diesem Fall das Addieren zweier Fließkommazahlen. Hier ist es egal, ob mit Streams oder Konstanten gerechnet werden soll. Die Funktionen beziehen sich auf einzelne Werte. Der letzte Bereich wird mit dem Schlüsselwort `CONTROL` eingeleitet. Hier teilt der Programmierer die Streams in einzelne Elemente und gibt diese an die Shader weiter. Die `forall`-Anweisung iteriert über alle Argumente und führt die nachfolgenden Funktionen mit allen Elementen der Vektoren aus. Das Programm wird von C++ aufgerufen. Dort findet zuerst eine explizite Initialisierung von CGiS statt, in welcher unter anderem die Größe der Vektoren festgelegt wird. Außerdem werden an dieser Stelle auch die Streams fest an Felder gebunden, so dass für Datenübertragungen nur noch die Richtung und der zu übertragende Stream genannt werden müssen. Ein Entwickler von CGiS hat im Internet eine sehr ausführliche Funktionsreferenz bereitgestellt, die den genauen Aufbau der Sprache mit allen Befehlen beschreibt [25].

Listing 3.1: *ein Shaderprogramm in BrookGPU*

```
kernel void mult( float f1<>, float f2, out float produkt<> ){
        produkt = f1 * f2;
}
```

Listing 3.2: *Hauptprogramm in C++ mit Aufrufen von BrookGPU*

```
1  BRTALIGNED float* afDaten = new float( anzahl );
2  BRTALIGNED float* afAusgabe = new float( anzahl );
3  // afDaten hier mit Daten fuellen
4
5  BRTALIGNED stream sfEingabe = stream::create<float>( anzahl );
6  BRTALIGNED stream sfErgebnis = stream::create<float>( anzahl );
7
8  streamRead( sfEingabe, afDaten );
9  BRTALIGNED float fKonstante = 3;
10 mult( sfEingabe, fKonstante, sfErgebnis ); // Berechnung auf GPU
11 streamWrite( sfErgebnis, afAusgabe );
```

Listing 3.3: *Stream-Definitionsbereich von CGiS*

```
PROGRAM vektor_add;

INTERFACE

extern inout float vektor1<SIZE>;
extern in float vektor2<SIZE>;

CODE

procedure add( in float summand1, in float summand2,
               out float summe ){
    summe = summand1 + summand2;
}

CONTROL

forall( v1 in vektor1, v2 in vektor2 ){
    add( v1, v2, v1 );
}
```

4 Algorithmen

Es gibt viele Probleme, die sich mit Hilfe der Grafikkarte lösen lassen, allerdings sind dazu spezielle Algorithmen erforderlich. Die einzelnen Recheneinheiten der Grafikkarte sind nicht so leistungsfähig wie ein Hauptprozessor. Es müssen demnach möglichst viele Berechnungen gleichzeitig durchgeführt werden, was vor allem bei Algorithmen mit Datenabhängigkeiten meist schwierig ist. Zurzeit ist es mit den in dieser Arbeit betrachteten Bibliotheken außerdem nicht möglich, ein Berechnungsergebnis an eine beliebige Position eines Streams zu schreiben, da sie von der Grafikkarte festgelegt wird. Wegen diesen Schwierigkeiten ist es manchmal nötig, Algorithmen mit schlechteren Laufzeiteigenschaften[1] auszuwählen.

4.1 Matrizenmultiplikation

Die Multiplikation von zwei Matrizen ist ein elementarer Bestandteil vieler Algorithmen. Die Laufzeit der Multiplikation ist bei Matrizen mit $n \times n$ Elementen $\mathcal{O}(n^3)$, was bei größeren Matrizen durchaus zu einer Rechenzeit von einigen Sekunden bis zu mehreren Minuten führen kann. Es existieren noch weitere Möglichkeiten für die Berechnung der Matrizenmultiplikation, beispielsweise der Algorithmus von Strassen mit einer Laufzeit von $\mathcal{O}(n^{2.8})$. Hier soll aber nur die einfache Variante behandelt werden, da beim Algorithmus von Strassen die Komplexität im Verhältnis zum Gewinn zu groß ist.

Eine Matrix A mit m Zeilen und n Spalten wird hier in der Kurzform $A^{m \times n}$ dargestellt. Für die Multiplikation werden zwei Matrizen $A^{m \times n}$ und $B^{n \times p}$ benötigt und das Ergebnis wird in $C^{m \times p}$ gespeichert. Das Skalarprodukt des i-ten Zeilenvektors von A und des k-ten Spaltenvektors von B wird in Zeile i, Spalte k von C geschrieben. Die Berechnung der Ergebniswerte von C ist unabhängig voneinander, was eine gute Voraussetzung für eine Parallelisierung schafft.

[1] siehe Groß-O-Notation → $\mathcal{O}$

4.2 Independent Component Analysis

Die Independent Component Analysis (ICA) wird in vielen Bereichen der Signalverarbeitung angewandt, etwa bei der Auswertung einer Elektroenzephalografie (EEG) oder für das Trennen von Audiosignalen. Die lange Berechnungszeit und die großen Datenmengen machen die ICA zu einem guten Kandidaten für die Berechnung auf der Grafikkarte.

Der ICA-Algorithmus bietet eine Möglichkeit, eine Matrix in bestimmte Bestandteile zu zerlegen. Das Cocktailpartyproblem liefert eine nichtwissenschaftliche Erklärung des generellen Problems: In einem Raum befinden sich einige sprechende Personen und mehrere Mikrofone. Jedes Mikrofon nimmt die Stimmen aller Personen auf. Ein Computer kann die in den Aufnahmen enthaltenen Stimmen nicht einfach voneinander trennen. Mit der ICA ist es möglich, die einzelnen Stimmen aus den Mikrofonsignalen wiederherzustellen.

Bei einer EEG nehmen am Kopf angebrachte Elektroden elektrische Signale aus dem Gehirn auf, wobei aber die Quelle eines Signals nicht genau lokalisiert werden kann. Mit Hilfe der ICA ist es möglich, das aufgezeichnete Gemisch der Signale zu trennen und eventuell vorkommende Störsignale wie das Netzbrummen einer nahen Stromleitung zu entfernen.

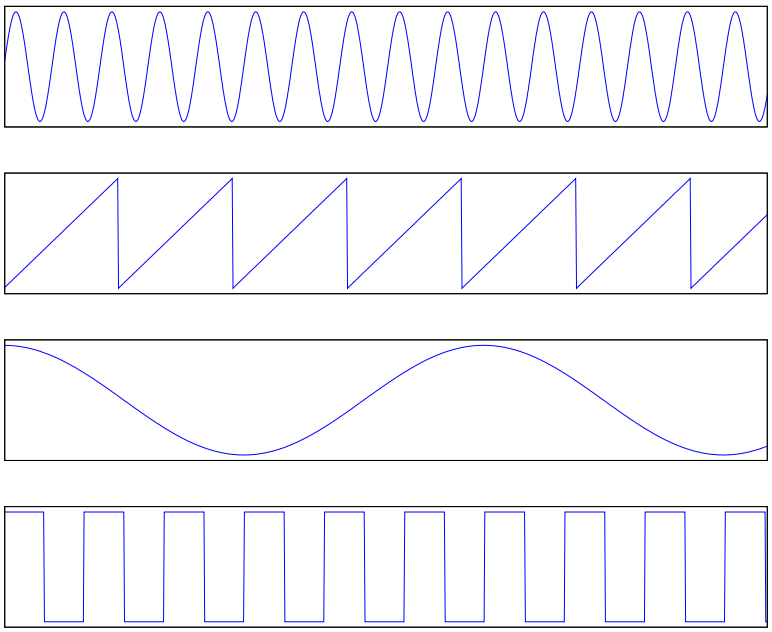

Abbildung 4.1: *Sinus, Sägezahnschwingung, Cosinus, Rechteckschwingung*

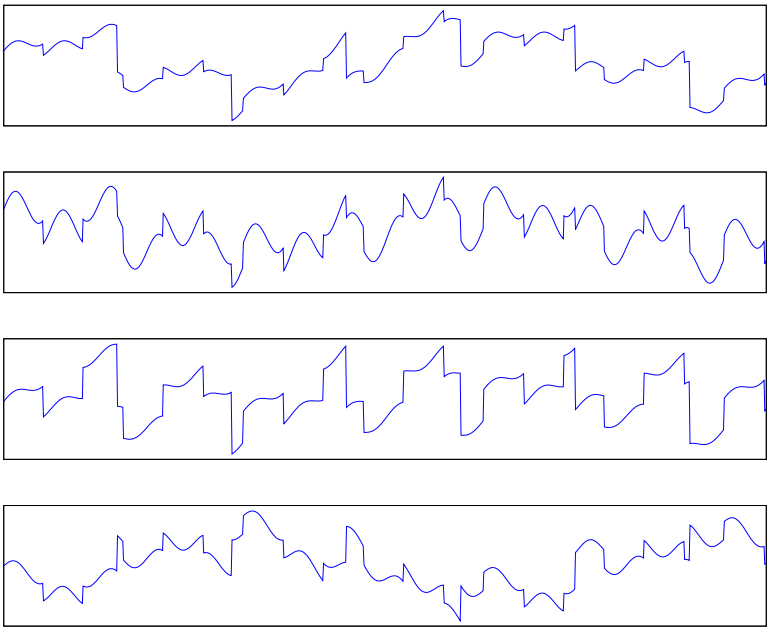

Abbildung 4.2: *gemischte Signale als Eingabe für den ICA-Algorithmus*

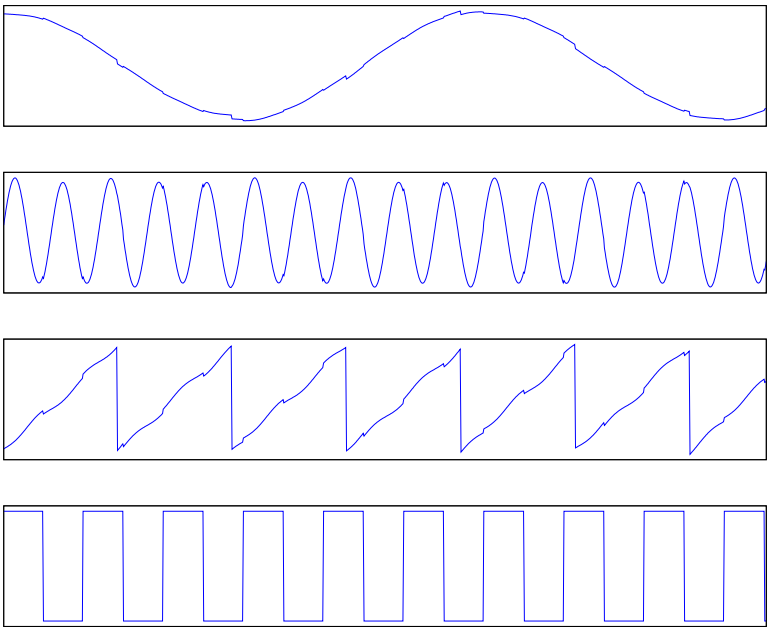

Abbildung 4.3: *Ausgabesignale des ICA-Algorithmus (100 Iterationen,* $\varepsilon = 10^{-6}$*, Optimierungsfunktion: Kurtosis)*

Zu Testzwecken werden die vier in Abbildung 4.1 dargestellten Ursprungssignale aus je 1000 Datenpunkten erzeugt. Weiterhin wird die in Formel 4.1 sichtbare Mischungsmatrix konstruiert. Sie bestimmt, zu welchem Teil die Ursprungssignale in die Mischsignale eingehen. Nach der Multiplikation von Mischungsmatrix und Ursprungssignalen entstehen die gemischten Signale, die man in Abbildung 4.2 sehen kann.

$$\begin{pmatrix} 1 & 2 & 3 & 1 \\ 3 & 1 & 2 & 2 \\ 1 & 3 & 1 & 3 \\ 1 & -1 & -2 & 1 \end{pmatrix} \tag{4.1}$$

Diese Signale entsprechen den Informationen, die Elektroden bei der EEG aufzeichnen. Die Ursprungssignale sind hier normalerweise nicht mehr zu erkennen, da aber in diesem Beispiel vier gut zu unterscheidende Signale genutzt werden, kann man auch mit bloßem Auge aus den Mischsignalen einen Teil der Ursprungssignale wiederherstellen. Bei echten Signalen ist dies schon alleine wegen der großen Signalanzahl unmöglich, eine EEG wird beispielsweise mit 32–128 Elektroden durchgeführt.

$$\begin{pmatrix} 3.27 & 0.87 & 2.21 & 0.86 \\ 2.18 & 2.22 & 1.10 & 1.84 \\ 1.13 & 0.93 & 3.42 & 2.75 \\ -2.17 & 0.63 & -1.08 & 1.00 \end{pmatrix} \Rightarrow \begin{pmatrix} 0.87 & 2.21 & 3.27 & 0.86 \\ 2.22 & 1.10 & 2.18 & 1.84 \\ 0.93 & 3.42 & 1.13 & 2.75 \\ 0.63 & -1.08 & -2.17 & 1.00 \end{pmatrix} \tag{4.2}$$

Je nach Algorithmus und Eingabeparametern unterscheiden sich die Ausgaben, insbesondere können die Ausgabesignale negiert oder anders skaliert sein als die Ursprungssignale. Beim Vergleich der Abbildungen 4.1 und 4.3 sieht man, dass die Reihenfolge der einzelnen Signale sich auch ändern kann. Eine Beispielausgabe des Algorithmus ist die erste Matrix aus Formel 4.2, welche zu Abbildung 4.3 führt. In der zweiten Matrix der Formel 4.2 sind die Spalten vertauscht und entsprechen der Anordnung in Formel 4.1. Im Vergleich dieser beiden Matrizen ist die Ähnlichkeit ersichtlich. Die Abweichungen führen zu den in Abbildung 4.3 sichtbaren Unterschieden gegenüber den Ursprungssignalen. Eine sehr ausführliche Beschreibung des ICA-Algorithmus und eine Implementation ist unter [26] zu finden.

4.3 Nicht-negative Matrizenfaktorisierung

Im Gegensatz zur ICA bestehen die Daten bei der nicht-negativen Matrizenfaktorisierung (NMF) nicht aus mehreren Einzelsignalen sondern sind eine Einheit. Somit kann die NMF beispielsweise genutzt werden, um Fotos von Gesichtern in einzelne Gesichtszüge zu zerlegen. Mit Hilfe einer Mischungsmatrix ist es dann möglich, beliebige Gesichter zu erzeugen, die auf den Gesichtszügen der Ursprungsfotos basieren.

Der Kern des Algorithmus besteht aus einer Schleife mit nur sieben Matrizenmultiplikationen, einigen elementweisen Operationen und der Summe aller Elemente einer Matrix. Alle Operationen lassen sich sehr gut parallelisieren, weswegen der Algorithmus für diese Arbeit ausgewählt wurde. Der Algorithmus wird in Abschnitt 5.4 auf Seite 31 genauer erklärt.

4.4 Sortieren

Die meisten Sortieralgorithmen erfassen die zu sortierenden Elemente sequentiell und schreiben das Element je nach Sortierkriterien an eine andere Stelle. Die Parallelisierung mit BrookGPU erlaubt das Lesen von beliebigen Elementen. Es kann aber nur an eine vorbestimmte Stelle geschrieben werden. Dies schließt sehr schnelle Algorithmen wie Quicksort mit $\mathcal{O}(n \log n)$ aus oder macht sie sehr komplex, da hier der Algorithmus die Ausgabeposition jedes Elements berechnet. Sortierverfahren wie Bubblesort sind mit BrookGPU möglich, aber relativ langsam, da zusätzlich zur Laufzeit von $\mathcal{O}(n^2)$ die doppelte Anzahl Vergleiche wie bei der klassischen Implementation nötig sind.

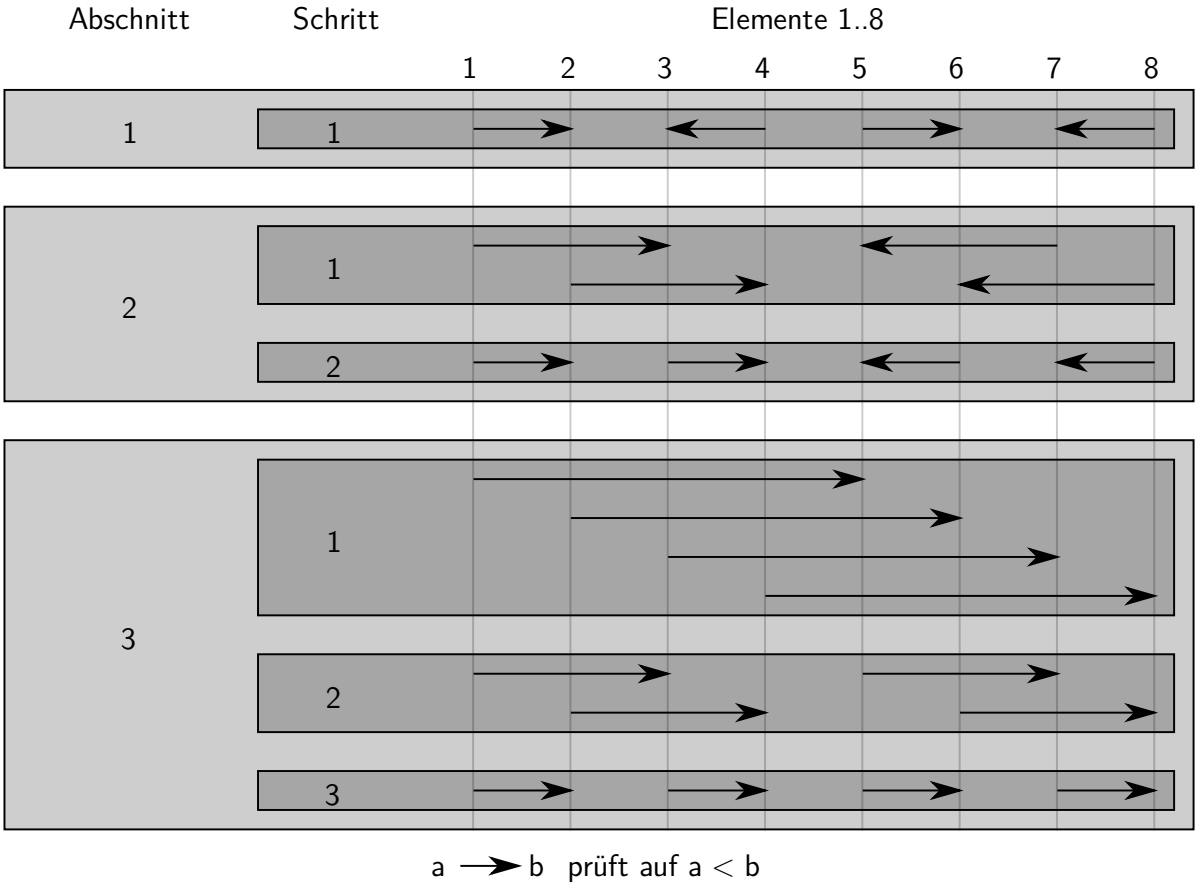

Abbildung 4.4: *Sortiernetzwerk für bitonisches Mergesort mit acht Elementen*

Schnellere Algorithmen lassen sich dennoch umsetzen, beispielsweise das bitonische Mergesort. Hier muss ebenfalls jeder Vergleich ($elem_a \overset{?}{<} elem_b$ und $elem_b \overset{?}{>} elem_a$) doppelt ausgeführt werden, da zum Vertauschen zweier Elemente nur je ein Element pro parallelem Aufruf geschrieben werden kann. Der Algorithmus ist in-place anwendbar und das Zugriffsmuster nur von der Anzahl Elemente bestimmt, nicht von deren Inhalt. Diese beiden Eigenschaften sind gute Anzeichen, dass ein Algorithmus in BrookGPU umsetzbar ist, da rekursive Aufrufe von GPU-Funktionen nicht möglich sind und ein statisches Zugriffsmuster den Ausgabebeschränkungen genügt.

In Abbildung 4.4 ist das Verfahren als Sortiernetzwerk visualisiert [27]. In jedem Abschnitt erfolgt eine bitonische Sortierung größer werdender Bereiche. Der letzte dieser Abschnitte erzeugt aus der bitonischen Folge eine monoton steigende Folge. Die Laufzeit des Algorithmus kann folgendermaßen ermittelt werden: Es werden bei n zu sortierenden Zahlen $\log_2 n$ Abschnitte benötigt. Die Abschnittsnummer a ist gleich mit der Anzahl Schritte in diesem Abschnitt. Gesucht ist die Summe aller Schritte s bei x Abschnitten. Jeder Schritt besteht wiederum aus n Vergleichen zweier Zahlen, deshalb werden insgesamt $n * s$ Berechnungen benötigt. In Formel 4.3 ist der weitere Rechenweg bis zur Laufzeitabschätzung zu sehen.

$$n * \sum_{a=1}^{x} a = n * \frac{x * (x+1)}{2} = n * \frac{\log_2 n * (\log_2 n + 1)}{2} = \mathcal{O}(n * \log_2^2 n) \qquad (4.3)$$

Die Laufzeit des bitonischen Mergesort ist folglich etwa $\log n$ mal höher als bei Quicksort.

5 Implementation

5.1 Octave

Octave ist eine kostenlose und freie Programmiersprache, die als Alternative zu Matlab entwickelt wurde. Octave ist mit Matlab weitestgehend kompatibel, wobei an einer besseren Kompatibilität zum Beispiel beim Erstellen von Fenstern gearbeitet wird. Das Grundelement beider Sprachen sind Matrizen, sie dienen unter anderem als Ein- und Ausgabe für Funktionen. Dies führt zu einem leichter lesbaren Quelltext, da alle Daten auf einmal an eine Funktion übergeben werden können.

Octave rechnet immer mit doppelter Genauigkeit, das heißt, dass etwa 15 Ziffern des Ergebnisses korrekt sind, danach treten Ungenauigkeiten auf. Grafikkarten rechnen üblicherweise in einfacher Genauigkeit, bei der ungefähr sieben Ziffern richtig berechnet werden. Doppelt-genaue Rechenoperationen benötigen zwei- bis fünfmal mehr Rechenleistung. Die in diesem Kapitel folgenden Abbildungen vergleichen somit Systeme mit verschiedener Rechengenauigkeit. In vielen Fällen reicht die einfache Genauigkeit dennoch aus, beispielsweise für das schnelle Berechnen einer Vorschauversion. Bei neueren Grafikkarten ist das Rechnen mit doppelter Genauigkeit möglich, doch leider unterstützt BrookGPU dies nicht.

Listing 5.1: *Nutzung von Matrizen und Funktionen in Octave*

```
octave:1> A = [ [1, 0.5, 0.25]; [0.75, 2, 1.5] ]
A =
   1.00000   0.50000   0.25000
   0.75000   2.00000   1.50000

octave:2> B = A * pi
B =
   3.14159   1.57080   0.78540
   2.35619   6.28319   4.71239

octave:3> C = sin(B)
C =
   1.2246e-16   1.0000e+00   7.0711e-01
   7.0711e-01  -2.4492e-16  -1.0000e+00
```

Listing 5.1 besteht aus drei Befehlen, welche im Interpreter von Octave ausgeführt wurden und deren Ergebnissen. Der erste Befehl erzeugt eine Matrix, welche in der Variable

`A` abgespeichert wird. Diese wird daraufhin mit π multipliziert und in `B` gespeichert. Es ist nicht nötig, jedes Element einzeln mit π zu multiplizieren, somit fällt die explizite Iteration über alle Elemente weg. Mit dem dritten Befehl `C = sin(B)` wird zum Schluss der Sinus jedes Elementes aus `B` berechnet. Mit diesem Vorgehen ist es dem Interpreter möglich, die einzelnen Berechnungen im Hintergrund in einer beliebigen Reihenfolge auszuführen. Weiterhin wäre eine Berechnung in einem parallelen Schritt denkbar, sofern die Hardware für eine Parallelverarbeitung zur Verfügung steht.

Funktionen sind in Octave entweder Skripte, die sich aus anderen Octave-Funktionen zusammensetzen, oder Plugins, die hauptsächlich in C++ geschrieben sind. Mit Hilfe dieser Befehlserweiterungen ist es möglich, den Funktionsumfang von Octave zu vergrößern und vorhandene Funktionen zu beschleunigen. Die Bibliotheken Basic Linear Algebra Subprograms (BLAS) und Automatically Tuned Linear Algebra Software (ATLAS) können beispielsweise für schnellere Berechnungen genutzt werden.

Die Octave Plugin-Schnittstelle

Plugins für Octave können in C, C++ oder Fortran geschrieben werden. Nach dem Kompilieren stehen sie als dynamische Bibliothek zur Verfügung, sind aber im Gegensatz zu normalen Octave-Skripts abhängig vom Betriebssystem und der Prozessorarchitektur. Da die oben genannten Sprachen zu Maschinencode kompiliert werden, laufen aufwendige Algorithmen alleine durch Nutzung eines Plugins schneller ab als dies mit dem Interpreter von Octave möglich ist. Dem Programmierer stehen außerdem alle Möglichkeiten der gewählten Programmiersprache und die in Octave nutzbaren Funktionen und Datentypen zur Verfügung.

5.2 Matrizenmultiplikation

In Abschnitt 4.1 auf Seite 16 wurde der allgemeine Multiplikationsalgorithmus für Matrizen vorgestellt. Dieser Algorithmus soll nun mit Hilfe von BrookGPU parallelisiert werden. Eine Möglichkeit besteht darin, alle Elemente der Ergebnismatrix gleichzeitig zu berechnen, dafür aber die innere Berechnung des Ergebniselements sequenziell durchzuführen. Es ist auch möglich, die Elemente der Ergebnismatrix nacheinander abzuarbeiten, innerhalb der Berechnung aber die Parallelisierung anzuwenden. Die zweite Möglichkeit wird aktuell durch einen Bug in BrookGPU erschwert, welcher das gleichzeitige Kopieren mehrerer Elemente aus einer Matrix in einen Vektor und umgekehrt verhindert. Diese Vorgehensweise wäre für eine schnelle Arbeitsweise des Algorithmus nötig. Selbst ohne diesen Bug ist der erste Ansatz wahrscheinlich eine bessere Lösung, da bei der Multiplikation von zwei Matrizen der Größe $n \times n$ im ersten Fall n^2 Einheiten parallel rechnen würden, wobei in Fall zwei nur n parallelen Einheiten eine Aufgabe zugewiesen werden könnte. Aus diesen Gründen wird hier nur der erste Ansatz weiter verfolgt.

Listing 5.2: *einfachste Implementation einer Matrizenmultiplikation*

```
kernel void matmatmul( float mat1[][], float mat2[][],
                       float size, out float output<> ){
        float2 index = indexof(output).xy;
        float sum = 0;
        int i=0;
        for( i=0; i<size; i+=1 )
                sum += mat1[index.y][i] * mat2[i][index.x];
        output = sum;
}
```

Listing 5.2 zeigt den einfachsten Implementationsversuch der Matrizenmultiplikation mit BrookGPU. Dem Algorithmus werden die beiden zu multiplizierenden Matrizen in den Streams `mat1` und `mat2` übergeben. Die gemeinsame Größe beider Matrizen wird in `size` angegeben, da es in einem Shader nicht möglich ist, die Anzahl der Elemente eines Streams zu ermitteln. Der letzte Parameter ist der Ausgabestream, der nach der Berechnung die Ergebnismatrix enthält. Als Erstes wird die Position des aktuellen Ausgabeelementes in der Matrix ermittelt. Dies geschieht mit dem Befehl `indexof`, welcher einen `float4`-Wert zurückliefert. Da mit einem zweidimensionalen Stream gerechnet wird, sind nur die Werte für x (Spaltenzahl) und y (Zeilenzahl) wichtig. Die folgende Schleife wird so oft abgearbeitet, wie `mat1` Spalten und `mat2` Zeilen hat. Im Schleifenkörper wird das Skalarprodukt berechnet, indem ein Wert aus jeder Matrix gelesen wird, beide Werte multipliziert werden und das Ergebnis auf die bisherige Summe aufaddiert wird. Die Summe wird sequentiell berechnet, allerdings werden die beschriebenen Aktionen gleichzeitig für alle Elemente der Ergebnismatrix ausgeführt.

Für kleine Matrizen funktioniert das recht gut, sobald sie eine Breite oder Höhe von 256 Elementen übersteigen, kommt es jedoch zu Berechnungsfehlern. Die Ursache ist der an die Grafikkarte übergebene Schleifenbefehl. Dieser kann maximal 256 Iterationen ausführen, da für den Iterationszähler nur ein Byte Speicher reserviert sind [28]. BrookGPU bricht das Kompilieren in diesem Fall nicht mit einem Fehler ab, sondern beendet die Schleife nach 256 Durchläufen. Die Lösung des Problems ist eine zweite Schleife innerhalb der bestehenden Schleife. In Listing 5.3 ist die erweiterte Version zu sehen. Die Berechnung des aktuellen Index ist im Vergleich zur vorherigen Variante komplizierter, rechnet dafür aber fehlerfrei.

Die Matrizen werden in dieser Variante etwas anders indiziert, wofür die Variable `index` die drei nötigen Zahlenwerte enthält. Zu Beginn sind nur zwei der Werte bekannt, weswegen in der ersten Zeile provisorisch zweimal der x-Wert abgespeichert wird. Die zwei ineinander geschachtelten Schleifen haben eigene Laufvariablen, die addiert den aktuellen Index ergeben. Dieser Wert wird in die Indexierungsvariable übertragen. Danach folgt die Berechnung der Summe. Im letzten Schritt ist zu beachten, dass `mat1[index.zy]` und `mat1[index.y][index.z]` das gleiche Ergebnis bewirken, wobei die erste Variante weniger Befehlen in der Shader-Sprache entspricht.

Listing 5.3: *zweite Implementation einer Matrizenmultiplikation*

```
kernel void matmatmul( float mat1[][], float mat2[][],
                       float size, out float output<> ){
        float3 index = indexof(output).xyx;
        float sum = 0;
        int i=0;
        int j=0;
        for( i=0; i<size; i+=256 ) {
                for( j=0; j<256 && j<size-i; j+=1 ) {
                        index.z = i+j;
                        sum += mat1[index.zy] * mat2[index.xz];
                }
        }
        output = sum;
}
```

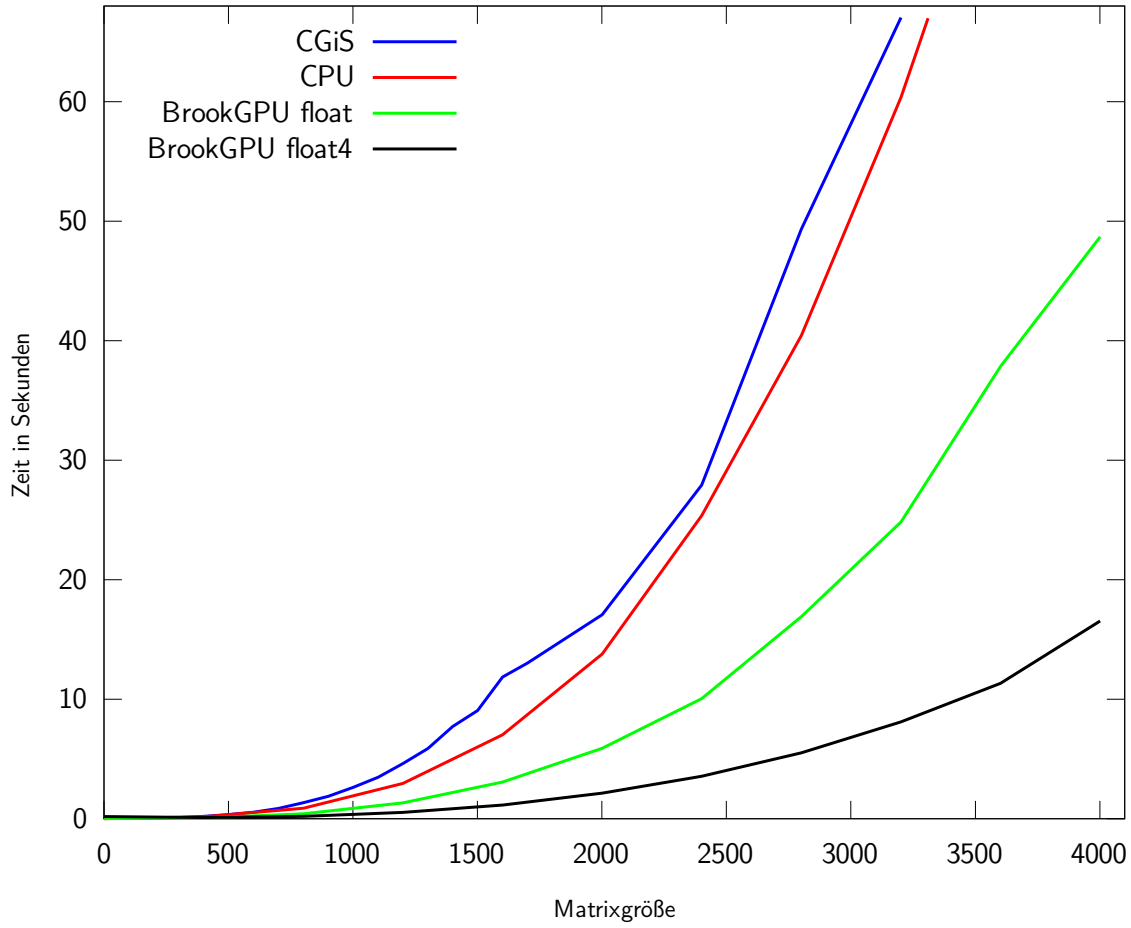

Abbildung 5.1: *Vergleich der Geschwindigkeit von BrookGPU, CGiS und CPU bei der Multiplikation quadratischer Matrizen*

Die Abbildung 5.1 zeigt die benötigte Zeit der Multiplikation von zwei quadratischen Matrizen der Größe 1×1 bis 4096×4096. Die BrookGPU-float-Version ist die bereits vorgestellte Implementation der einfachen Matrizenmultiplikation und für CGiS wurde die eingebaute Multiplikationsmethode benutzt. Am Ende des Abschnittes wird die Version "BrookGPU-float4" erklärt, die eine Verbesserung des ursprünglichen Algorithmus darstellt. Im Vergleich zur CPU-Implementation erreicht die einfache BrookGPU-Version einen Speedup von Zwei, während die verbesserte Variante sogar die sieben-fache Geschwindigkeit gegenüber der CPU hat. CGiS liegt etwa gleichauf mit der CPU, was eventuell durch eine unoptimierte Implementation der bereits in CGiS integrierten Matrizenmultiplikation zu erklären ist. Da es aber noch nicht offiziell verfügbar ist, wird auf weitere Tests und Spekulationen verzichtet.

Beide Bibliotheken initialisieren sich beim ersten Aufruf in einer neuen Octave-Session nur einmalig, bei allen weiteren Aufrufen ist die benötigte Zeit verringert. Abbildung 5.2 zeigt die Auswirkungen auf die Laufzeit bei einer Multiplikation von zwei 10×10-Matrizen. Der jeweils erste Balken stellt die Rechenzeit inklusive Initialisierung dar, während im zweiten Balken die Initialisierungzeit nicht enthalten ist. Die Initialisierung beinhaltet die Erzeugung eines OpenGL-Fensters und die Reservierung von Systemresourcen. Dieser Prozess benötigt bei BrookGPU über 98 % der Zeit, bei CGiS sind es immerhin noch knapp 80 %. Bei reiner CPU-Nutzung sind die Zeiten viel geringer, aber auch hier ergibt sich ein Unterschied von etwa 60–70 %.

In Abbildung 5.3 ist eine ähnliche Berechnung wie in Abbildung 5.2 zu sehen, wobei hier aber die Matrizen eine Größe von 1000×1000 haben. War die Initialiserungszeit bei einer kleinen Datenmenge noch deutlich größer als die Berechnungszeit, ist sie bei einem komplexeren Problem mit einem Zeitanteil von 6 % bei BrookGPU und 2 % bei CGiS nicht mehr relevant. Die absolute Initialisierungszeit hat sich bei beiden Szenarien nicht verändert, es vergehen weiterhin 0.5 beziehungsweise 0.4 Sekunden.

Nach dem ersten Aufruf ist die Standardabweichung der Initialisierungszeit bei BrookGPU und CGiS niedrig, aber weitaus höher als die Zeitabweichung beim Berechnen mit CPU. Sehr gut ist dies bei Abbildung 5.6 und Abbildung 6.1 sichtbar, hier ergeben die Messwerte der CPU eine glatte Kurve, wogegen in den Werten von BrookGPU starke Schwankungen vorhanden sind.

Grafikkarten arbeiten meist mit vier zusammengehörigen Werten, etwa x, y, z und w bei Objektdaten oder Rot, Grün, Blau und Transparenzwert bei Bildfarben. Aus diesem Grund unterstützen die Befehle der Grafikkarte die gleichzeitige Berechnung von bis zu vier Werten. Gruppiert man die Streamwerte in `float4`-Elementen, dann können vier Multiplikationen des Skalarproduktes gleichzeitig berechnet werden. Eine optimierte Implementation des vorgestellten Algorithmus basiert auf diesem Konzept. Dafür werden die Matrizen in Streams geladen, die in jedem Element vier Werte speichern können. Abbildung 5.4 zeigt die Anordnung der Elemente in den Matrizen. Der umrandete grüne Bereich enthält die ursprünglichen Daten, die restlichen Elemente sind auf null gesetzt. Es ist nötig, dass die Matrix in allen Richtungen ein Vielfaches von vier als Länge hat, da sie sich so einfacher transponieren lassen. Die Implementation besteht aus einem Shader,

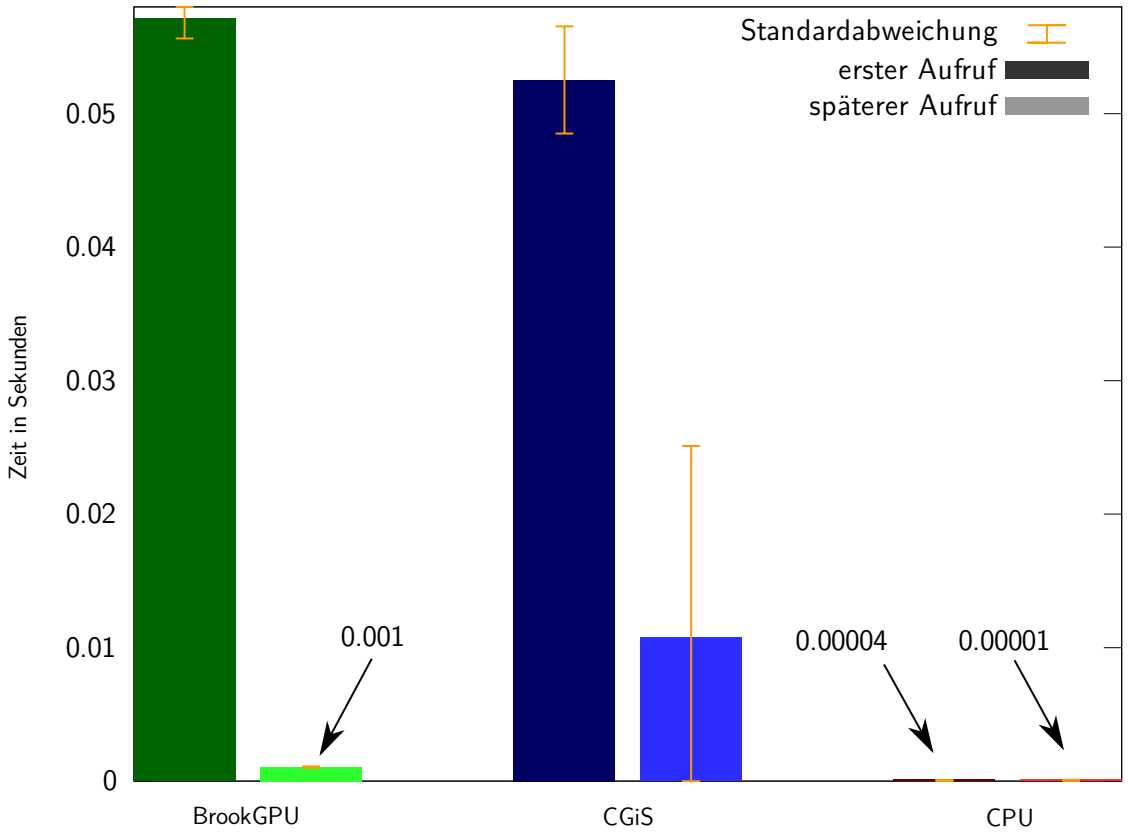

Abbildung 5.2: *Vergleich der Initialiserungszeit von BrookGPU, CGiS und CPU bei der Multiplikation zweier* 10×10 *Matrizen (Median aus 5 Messungen)*

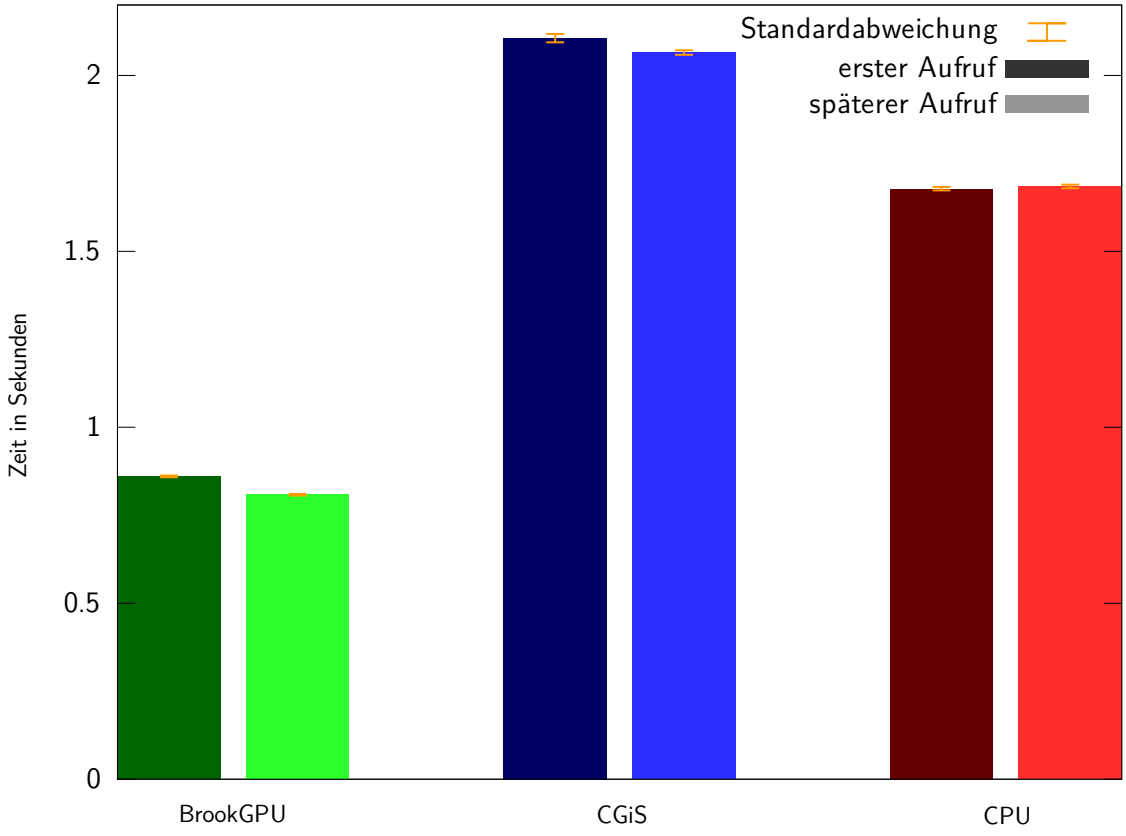

Abbildung 5.3: *Vergleich der Initialisierungszeit von BrookGPU, CGiS und CPU bei der Multiplikation zweier* 1000×1000 *Matrizen (Median aus 5 Messungen)*

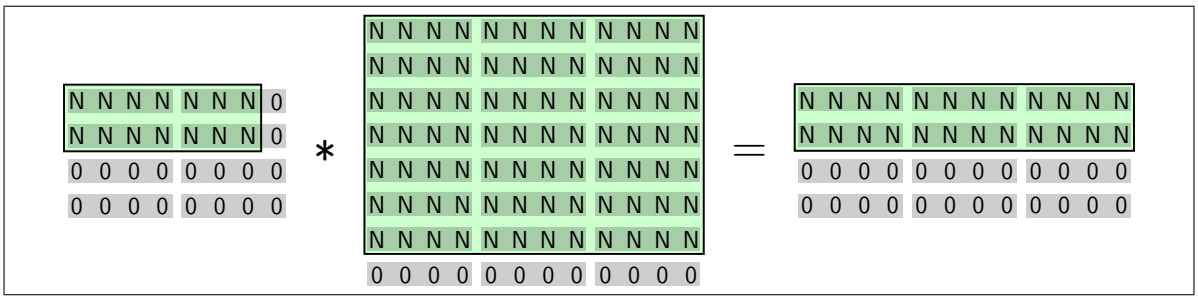

Abbildung 5.4: *Matrizenmultiplikation mit Speicherung der Werte in Vierergruppen*

der `float`-Streams in `float4` umwandelt, dem Berechnungsshader und einem Shader, der die Daten wieder nach `float` zurückwandelt. Der Quelltext benötigt viel Platz und wird aus diesem Grund in Anhang B mitgeliefert. Trotz des Umwandlungsaufwandes ist diese Version zwei bis drei Mal so schnell wie die erste Implementation. Dies ist in Abbildung 5.1 beim Vergleich der grünen und schwarzen Linie ersichtlich.

5.3 Independent Component Analysis

Die hier verwendete Implementation der ICA nennt sich fastICA und wurde von Aapo Hyvärinen entwickelt [29]. Sie ist laut Erfinder die wohl am weitesten verbreitete Variante der ICA, da sie sehr schnell und robust ist. Das Verfahren basiert auf der Annahme, dass alle Ursprungssignale voneinander statistisch unabhängig sind. Ziel der Berechnung ist dann, die Nicht-Gaußhaftigkeit der Signale zu maximieren. Dies bedeutet umgekehrt, dass die Zufälligkeit der Signale minimiert wird. Das führt zu der statistischen Unabhängigkeit, welche die Ursprungssignale auszeichnet. Im Algorithmus wird eine Mischungsmatrix errechnet, mit deren Hilfe die Eingangssignale in möglichst unabhängige Signale umgewandelt werden können. In jeder Iteration des Algorithmus wird eine Berechnung ausgeführt, welche die Mischungsmatrix verändert und die Unabhängigkeit der Signale vergrößert. Für diese Berechnung stehen im hier verwendeten Algorithmus drei Möglichkeiten zur Verfügung: Kurtosis, Kosinus Hyperbolicus und Normalverteilung. Abhängig von den Eingangsdaten ist einer der Optimierungsansätze besser für die Trennung der Signale geeignet. In Abbildung 4.3 wird die Kurtosis verwendet, da mit den anderen Optimierungsansätzen die Sinus- und Rechteckschwingung nicht so gut von der Sägezahnschwingung getrennt werden können, so dass die errechneten Signale den Ursprungssignalen weniger ähnlich sind.

Der ebenfalls in Abschnitt 4.2 auf Seite 17 benutzte fastICA-Algorithmus erhält als Eingaben alle Datenpunkte als Matrix. Weiterhin wird übergeben, dass beispielsweise die Kurtosis genutzt werden soll. Außerdem wird eine maximale Anzahl zu berechnender Durchgänge angegeben und eine minimale Verbesserung ε, die bei jeder Iteration erreicht werden muss.

In der hier verwendeten Implementation kommen nur vektorbasierte Operationen vor. Da Matrizen in BrookGPU maximal 4096×4096 Elemente enthalten können, fasst ein Vektor auch nicht mehr als 4096 Werte. Das volle Parallelisierungspotenzial der Grafikkarte wird leider erst bei größeren Datenmengen erreicht. Dies führt zu einer recht niedrigen Berechnungsgeschwindigkeit. Abbildung 5.5 zeigt dieses Verhalten. Der Zeitunterschied zwischen dem kleinsten und größten dargestellten Messwert beträgt nur 40 %. Dies ist auf eine schlechte Auslastung der Grafikkarte zurückzuführen, da zu wenige Elemente gleichzeitig zur Verfügung stehen. Deswegen benötigen Berechnungen mit einer größeren Datenmenge kaum zusätzliche Zeit.

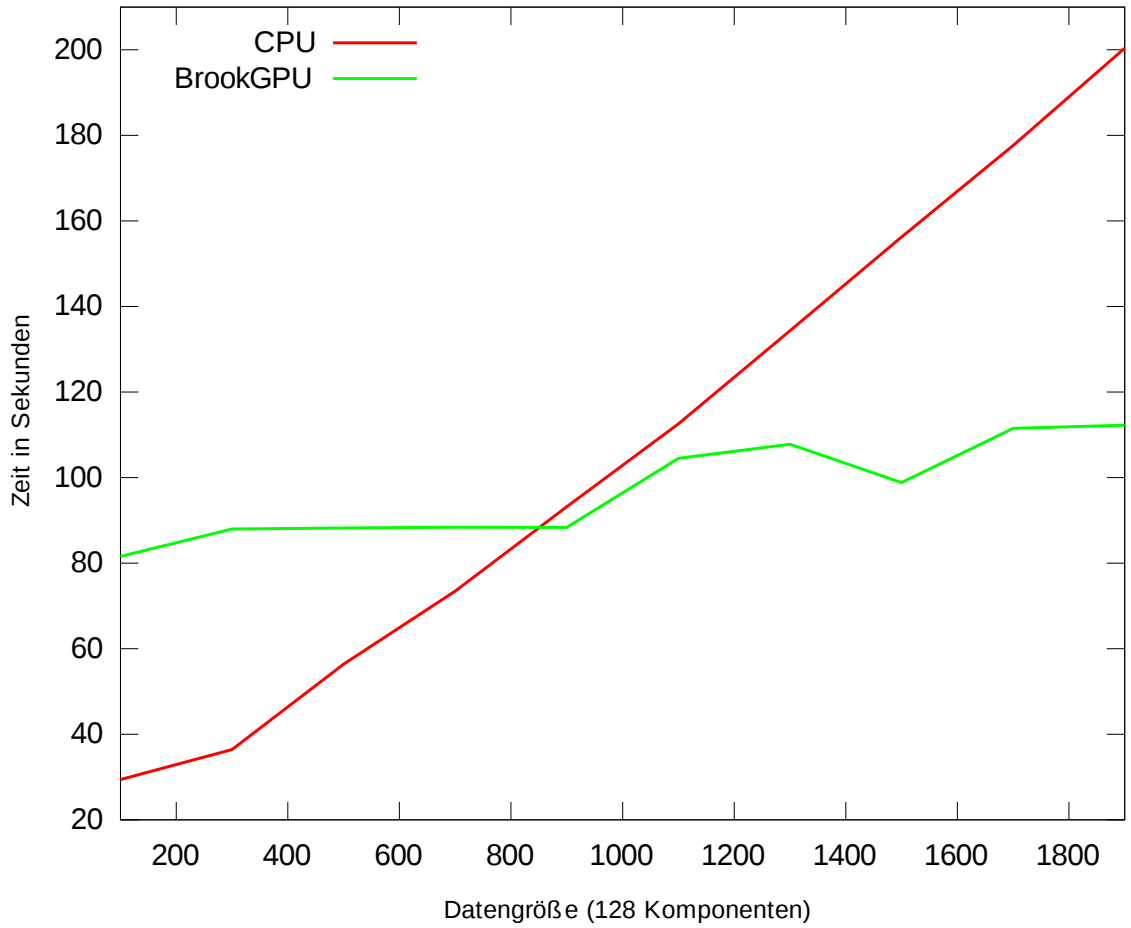

Abbildung 5.5: *Identischer fastICA-Algorithmus ausgeführt von Octave und BrookGPU* ($\varepsilon = 0$)

Die verwendete Implementation von fastICA ist sicher noch optimierbar. Beispielsweise könnten die Vektoroperationen so gruppiert werden, dass die Berechnung auf der kompletten Matrix stattfindet, anstatt jede Zeile oder Spalte einzeln zu berechnen. Die Schwierigkeit besteht darin, dass jedes zu extrahierende Signal unterschiedlich viele Iterationen für ein gutes Ergebnis benötigen kann. Fasst man die Berechnungen nun so zusammen, dass alle Signale gleichzeitig errechnet werden, dann führt dies zu ungenaueren Ergebnissen. Eine weitere Möglichkeit ist die Nutzung eines Algorithmus mit einer höheren Parallelität. Aktuell lohnt sich diese ICA-Implementation auf Grafikkarte nur, wenn mit vielen Daten gerechnet werden muss. Je nach Anwendungszweck wäre es demnach durchaus vorteilhaft, die vorgestellte Version einzusetzen.

5.4 Nicht-negative Matrizenfaktorisierung

Wie bereits erwähnt, eignet sich die NMF gut zum Test des Parallelitätspotenzials eines Systems. Der Algorithmus wird in Listing 5.4 als Octave-Implementation dargestellt. In Anhang B befinden sich zwei Implementationen in BrookGPU, die sich sehr nah an der Octave-Version orientieren.

Listing 5.4: *NMF in Octave, Implementation von Andreas Romeyke/MPI*

```
1  [zeilen, spalten] = size(eingabe);
2  u = rand(zeilen, kompression); %  Matrix mit Zufallszahlen fuellen
3  v = rand(kompression, spalten); % Matrix mit Zufallszahlen fuellen
4  eingabe = eingabe / max(max(eingabe)); % Daten auf 0..1 skalieren
5  for n = 1:maxIterationen
6      a = u' * eingabe;
7      b = u' * u * v;
8      v = v .* a ./ b;
9      a = eingabe * v';
10     b = u * v * v';
11     u = u .* a ./ b;
12     % Fehler berechnen:
13     eingabeBerechnet = u * v;
14     fehler = sum (sum( abs(eingabe-eingabeBerechnet) ));
15     if (fehlerAlt - fehler < epsilon)
16         break;
17     end
18     fehlerAlt = fehler;
19 end
```

Zu Beginn des Algorithmus werden die Ergebnismatrizen `u` und `v` mit Zufallszahlen initialisiert. Die Spalten von `u` und die korrespondierenden Zeilen aus `v` enthalten zum Schluss die Komponenten, aus denen die Eingabematrix besteht. Mit Hilfe des Parameters `kompression` wird bestimmt, wie viele dieser Komponenten errechnet werden sollen. Die Eingabematrix darf nur Daten im Bereich von Null bis Eins enthalten, deshalb werden die Daten in Zeile vier in diesen Bereich skaliert. Danach folgt die Hauptschleife, in der das Optimierungsverfahren abläuft. In den Zeilen sechs bis elf wird die meiste Zeit verbraucht, hier werden Matrizen miteinander multipliziert. Das Hochkomma (`'`) drückt in Octave das Transponieren der Matrix aus. `.*` und `./` drücken die elementweise Multiplikation beziehungsweise Division aus. Zum Schluss jeder Iteration wird der Unterschied zwischen der berechneten Matrix und der Eingangsdaten berechnet. Verringert sich dieser Fehler bei einer Iteration nur noch um einen kleinen Betrag, dann bricht der Algorithmus ab, spätestens aber, wenn die vom Benutzer angegebene maximale Anzahl Iterationen erreicht ist.

Auf der Grafikkarte wurden die Zeilen sechs bis 14 umgesetzt, da die nachfolgenden Zeilen keine parallelen Elemente enthalten und die Berechnungen der Zeilen eins bis vier auf der CPU sehr schnell ablaufen.

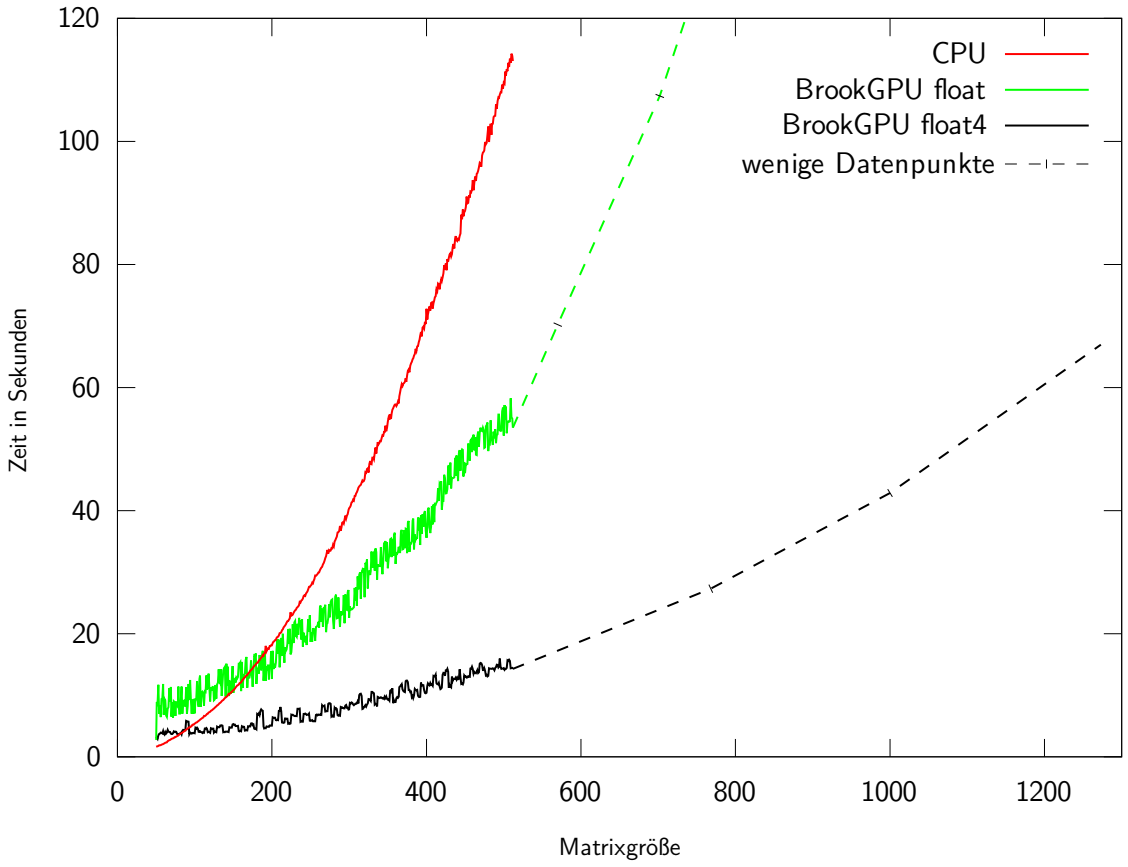

Abbildung 5.6: *Vergleich verschiedener NMF-Implementationen mit NVIDIA GeForce 7900GTX (Iterationen: 1000, Kompression: 50, quadratische Matrix)*

Beim Erstellen der Abbildung 5.6 trat ein Bug auf, der das Octave-System nach einigen Aufrufen der NMF-Berechnung zum Absturz brachte. Es stellte sich heraus, dass der verwendete Speicher von BrookGPU nicht ordnungsgemäß freigegeben wurde, weshalb es nach mehrmaligem Ausführen der Funktion zu einem Speicherüberlauf kam. Aus diesem Grund ist in der Abbildung nur mit wenigen Messwerten dargestellt, wie der weitere Verlauf der Kurven ist. Dieser Fehler ist auch bei den anderen geschriebenen Funktionen mit BrookGPU provozierbar, allerdings deutlich schwerer als hier, da sonst wesentlich weniger Streams genutzt werden.

Die rote Kurve zeigt die Laufzeit der oben vorgestellten Implementation in Octave. Die benötigte Zeit schnellt bei größeren Matrizen stark in die Höhe, was bei der häufigen Verwendung von Matrizenmultiplikationen mit einer Laufzeit von $\mathcal{O}(n^3)$ nicht verwunderlich ist. Die direkte Umsetzung dieser Implementation in BrookGPU wird mit der grünen Kurve dargestellt. Es ist deutlich ersichtlich, dass die Grafikkarte das Ergebnis ab einer Matrizengröße von 200×200 Elementen schneller berechnet als die CPU. Die schwarze Linie stellt ein sehr gutes Laufzeitverhalten dar. Dies ist mit Hilfe der `float4`-Implementation der Matrizenmultiplikation möglich, die schon in Abschnitt 5.2

auf Seite 23 vorgestellt wurde. Der Algorithmus ist gleich zu den anderen beiden Varianten. Die Matrizen werden auf der Grafikkarte zu Begin nach `float4` gewandelt. Die darauffolgenden Berechnungen wurden angepasst, damit sie vier Werte gleichzeitig verarbeiten können. Zum Schluss wird die Ergebnismatrix wiederum in eine `float`-Matrix umgewandelt, so dass die Implementation keine anderen Parameter erfordert als die einfache Variante. Der Quelltext der Programme ist auch hier recht groß, weshalb er über Anhang B erreichbar ist. Dort befinden sich beide Versionen der NMF und auch die Octave-Implementation.

5.5 Sortieren

Listing 5.5: *Hauptschleife des bitonischen Mergesort in C++*

```
1 for( int stage = 2; stage <= arrSizeP2; stage *= 2 ) {
2         for( int step = stage; step > 1; step /= 2 ) {
3                 sortStep( stage, step / 2, step,
4                           streamIn, streamOut );
5                 streamIn.swap(streamOut);
6         }
7 }
```

Listing 5.5 zeigt den Hauptteil der BrookGPU-Implementation als C++-Quelltext mit dem Aufruf einer parallel abgearbeiteten BrookGPU-Funktion. Der Algorithmus wurde anhand der Struktur in Abbildung 4.4 implementiert. Die äußere Schleife ab Zeile eins iteriert über alle Abschnitte, wobei $stage = 2^{Abschnitt}$ gilt. In der inneren Schleife wird dieser Wert in jedem Durchgang halbiert, so dass die Variable `step` immer dem doppelten Abstand zwischen zwei zu vergleichenden Werten entspricht. Dieses Vorgehen erleichtert die weitere Berechnung, da für die folgenden Rechenschritte die Zweierpotenzen von Abschnitt und Schritt benötigt werden. An die BrookGPU-Funktion `sortStep` in Zeile drei wird `step / 2` und `step` übergeben, da so die Division nur einmal durchgeführt werden muss und nicht in jedem Shader. Mit einem Aufruf von Zeile drei wird ein ganzer Schritt des Algorithmus berechnet, wobei alle Elemente parallel verglichen werden. Der letzte Befehl tauscht schließlich die Ein- und Ausgabestreams. Dies ist nötig, da eine BrookGPU-Funktion nicht denselben Stream für die Eingabe und die Ausgabe nutzen darf. In so einem Fall, der vom Compiler übrigens nicht abgefangen wird, wäre das Ergebnis undefiniert.

Die BrookGPU-Implementation kann nur genau 2^n Elemente sortieren. Wenn die gegebene Anzahl geringer ist, dann müssen die zusätzlichen Werte auf ∞ gesetzt werden. In Abbildung 5.7 sind Stufen zu sehen, da nicht nur die übergebenen Elemente sortiert werden, sondern die Anzahl der Werte auf die nächste Zweierpotenz aufgerundet wird.

Wie man sehen kann, benötigt BrookGPU bei diesem Algorithmus mehr Zeit als die CPU. Das bei der Grafikkarte verwandte bitonische Mergesort ist algorithmisch schlechter als Quicksort, das von Octave genutzt wird. Dieser Umstand wurde in Abschnitt 4.4 auf Seite 20 bereits angesprochen. In den GPU Gems 2 [27] wird dieser Algorithmus etwas anders implementiert, wobei außer dem Pixelshader auch der Vertexshader genutzt wird. Bei der Programmierung mit BrookGPU kann man dieses Verhalten nicht erzwingen. Obwohl eine andere Implementation eingesetzt wird, übertrifft der Algorithmus das für die CPU genutzte Sortierverfahren nicht. Es eignet sich deshalb nur, wenn bereits im Grafikspeicher befindliche Streams sortiert werden sollen und die Transferzeiten zur und von der CPU zu hoch sind.

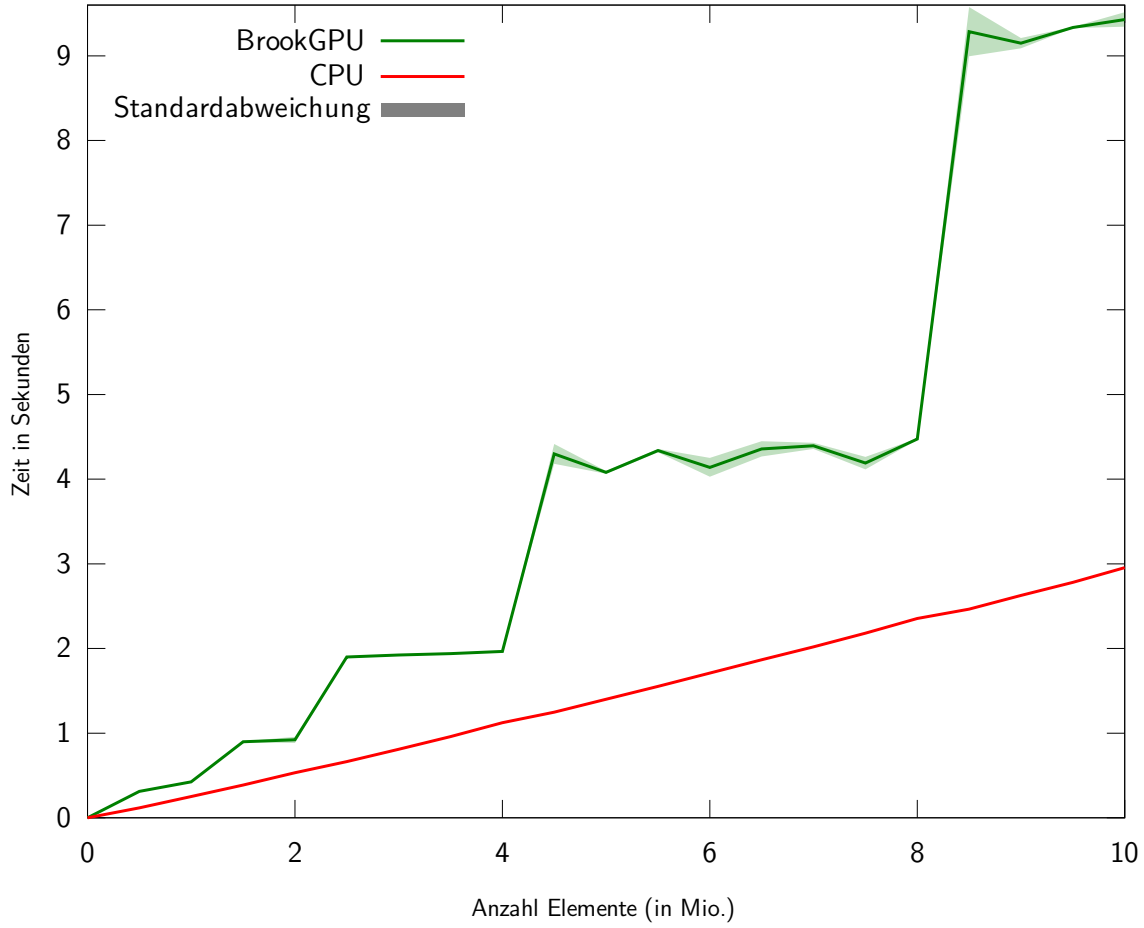

Abbildung 5.7: *Vergleich der Sortiergeschwindigkeit von BrookGPU und CPU mit zufälligen Daten (Median aus 9 Messungen)*

6 Diskussion

In diesem Kapitel werden Methoden beschrieben, mit denen Algorithmen mit BrookGPU effektiver ausgeführt werden können. Des Weiteren stehen Wege für die Vermeidung von Fehlern im Vordergrund.

6.1 Optimierung

Die Grafikkartenhersteller geben nicht alle internen Details ihrer Grafikkarten frei, deswegen sind Tests nötig, um die Anpassungen mit dem größten Optimierungspotenzial zu finden. Da sich der Aufbau zwischen den Grafikkartengenerationen stark unterscheiden kann, variieren Optimierungsbedarf und -strategie teilweise sehr stark.

Nachfolgend werden mehrere Optimierungsstrategien kurz vorgestellt. Dies soll Entwicklern helfen, ihre Programme zu beschleunigen.

Je nach Grafikkartenarchitektur sind unterschiedliche Optimierungsansätze hilfreich. Karten bis zur 7900er Reihe von NVIDIA profitieren beispielsweise durch Nutzung von `float4`-Datentypen für Berechnungen sehr stark. In Abbildung 5.6 ist ein zusätzlicher Speedup von zwei bis vier sichtbar, der mit der obigen Optimierung erreicht wurde. Eine Architekturänderung bei aktuelleren Grafikkartengenerationen verringert die Effizienz dieser Änderung. Es ließ sich nur noch ein Speedup von eins bis drei zeigen [30]. In Abbildung 6.1 ist bei einem Vergleich der grünen und schwarzen Linien deutlich sichtbar, dass nach der `float4`-Anpassung weniger Rechenzeit benötigt wird. Gegenüber Abbildung 5.6 fällt die Verbesserung hier aber deutlich geringer aus. Dies ist auf die oben angesprochene Architekturänderung zurückzuführen.

Tests zeigen außerdem, dass NVIDIA-Karten bis zur 7900er Reihe Texture Fetches nicht maskieren [30]. Deshalb können keine Berechnungen ausgeführt werden, während auf Daten aus dem Texturspeicher gewartet wird. Aktuellere Grafikkartengenerationen setzen während der Wartezeit unabhängige Berechnungen fort, wodurch die Lesezeit des Texturspeichers weniger stark die Gesamtberechnungszeit beeinflusst.

Bei manchen Programmen lässt sich mit nicht offensichtlichen Tricks etwas Geschwindigkeit dazugewinnen. Es ist beispielsweise in BrookGPU schneller, zwei einzelne `float`-Streams zur Grafikkarte zu kopieren und dann mit Hilfe eines Shaders zu einem `float2`-Stream zusammenzufügen, als direkt den `float2`-Stream zur Grafikkarte zu kopieren.

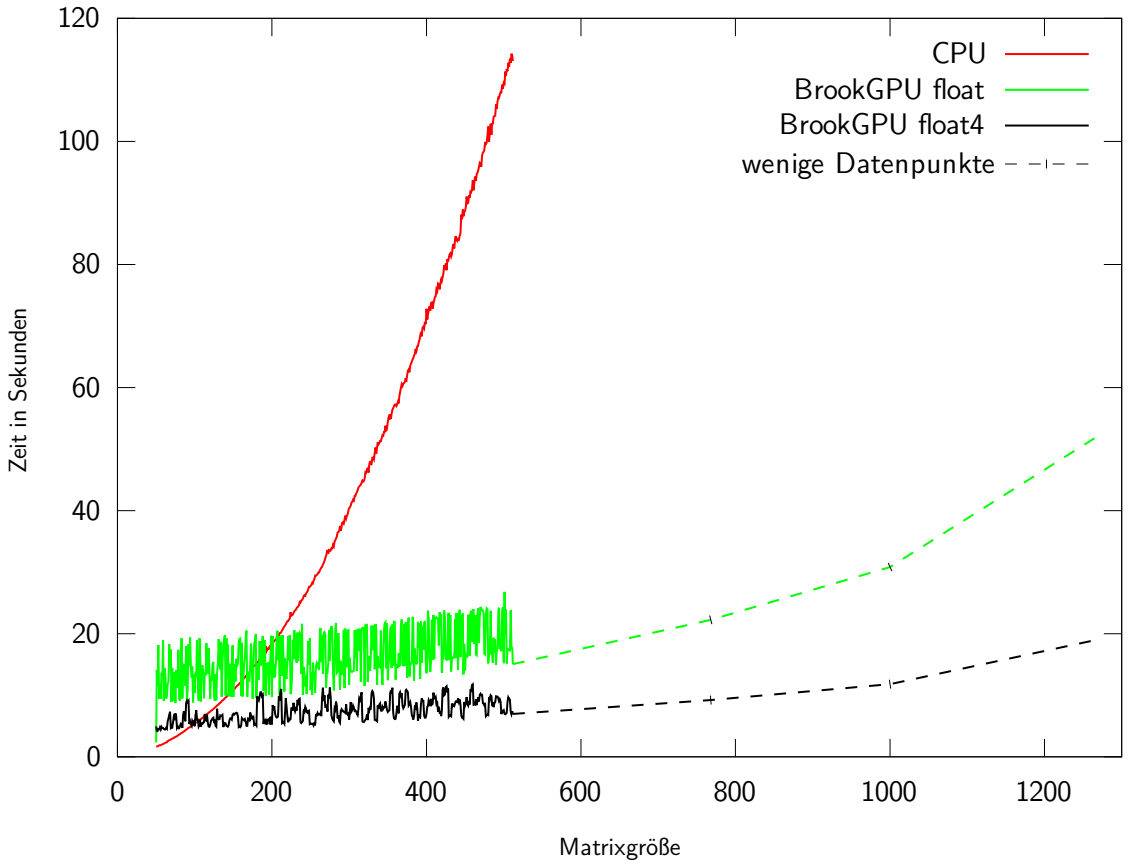

Abbildung 6.1: *Vergleich verschiedener NMF-Implementationen mit NVIDIA GeForce GTX280 (Iterationen: 1000, Kompression: 50)*

Dies lässt sich durch den Overhead bei Zugriffen auf den `float2`-Datentyp in C++ erklären. Diese Optimierung ist bei jeder `float`X-Variante (2,3,4) anwendbar.

Kleine Änderungen der Ausführungsreihenfolge können einen Algorithmus stark beschleunigen. Beispielsweise kann die Matrizenmultiplikation $A * B * C$ entweder als $A * (B * C)$ oder als $(A * B) * C$ implementiert werden. Je nachdem, wie groß die Zwischenmatrizen $A * B$ und $B * C$ sind, ist eine der Varianten schneller. Hier empfiehlt es sich, die Ausführungsreihenfolge je nach der Größe der Eingabematrizen vom Programm wählen zu lassen.

Alle Aufrufe von BrookGPU-Funktionen lassen sich in den Programmfluss von C++-Programmen integrieren. Dies ermöglicht die Gruppierung von C++-Code und BrookGPU-Befehlen zu Einheiten, wodurch die Lesbarkeit und Wiederverwendbarkeit der Programmteile stark verbessert wird.

Wie sich in Tests herausstellte, ist das OpenGL-Runtime unter Windows und Linux etwa gleich schnell. Der Vergleich von OpenGL zu DirectX zeigte einen minimalen Geschwindigkeitsvorteil für DirectX. Letzteres war lange das hauptsächlich unterstützte und ge-

nutzte Runtime. Erst die Änderungen in den letzten Monaten der aktiven Entwicklung von BrookGPU brachten ein Update des OpenGL-Runtimes, welches möglicherweise noch nicht so gut optimiert ist.

6.2 Tipps zur Fehlervermeidung

Die hier erläuterten Fälle sind Probleme, die während der Entwicklung der in Kapitel 5 vorgestellten Programme auftraten. Sie können alle als Bugs in BrookGPU bezeichnet werden, da der Compiler den Quelltext nicht genügend auf Fehler geprüft hat und Programme mit undefinierbarem Verhalten erzeugte.

Der `loop`-Befehl des häufig verwendeten Pixelshader 3.0 unterstützt maximal 256 Iterationen pro Schleife [28]. Wird diese Obergrenze überschritten, führt dies in BrookGPU nicht zu einer Fehlermeldung, sondern der Shader rechnet bis 256 und beendet danach die Schleife! Mit etwas erhöhtem Berechnungsaufwand kann die Grenze auf maximal $256^4 = 4294967296$ Schleifendurchläufe erhöht werden, indem bis zu vier Schleifen ineinander geschachtelt werden [31]. Dieses Problem tritt erst ab einer bestimmten Datengröße auf. Es ist empfehlenswert, alle Algorithmen mit unterschiedlich vielen Daten zu testen, da manche Fehler eventuell nur bei sehr großen oder kleinen Datenmengen auftreten.

In BrookGPU wird der Datentyp `int` nur teilweise unterstützt. Im OpenGL-Runtime werden Ganzzahlen in Funktionsparametern mit Hilfe von Fließkommazahlen emuliert. Beim Aufruf mittels CPU-Runtime sind diese Variablen allerdings mit zufälligen Werten gefüllt. Einer der Entwickler rät deshalb, den Datentyp `int` nur für Laufvariablen von Schleifen zu verwenden und in allen anderen Fällen auf `float` auszuweichen [32].

Der BrookGPU-Compiler verarbeitet zwei verschiedene Eingabedateien: Der normale Quelltext steht in `*.br`-Dateien (ähnlich zu `*.cpp`) und die Funktionssignaturen können in `*.brh`-Dateien (analog zu `*.hpp`) geschrieben werden. Der Compiler arbeitet bei letzterer Variante falsch, was zu Laufzeitfehlern führen kann. Entweder sollte man die vom Parser erzeugten `*.hpp`-Dateien mit einem Skript korrigieren oder auf die `*.brh`-Dateien verzichten und stattdessen die Funktionssignaturen direkt in `*.hpp`-Dateien schreiben. Das Skript befindet sich in jedem Projekt am Ende des `Makefile`.

Mit Hilfe der Datei brook/brookgpu_bugfixes.diff lassen sich einige weitere Fehler beseitigen. Es existieren in BrookGPU dennoch viele Probleme, die bisher nicht behoben wurden. Sie schränken die Anzahl der umsetzbaren Algorithmen ein. Es ist beispielsweise aufgrund eines Fehlers nicht möglich, eine Zeile eines zweidimensionalen Streams in einen eindimensionalen Stream zu kopieren, obwohl diese Möglichkeit eigentlich vorgesehen ist.

6.3 Ergebnis

Beim Start eines BrookGPU-Programmes wird Speicher zum Austausch von Daten mit der Grafikkarte reserviert und beim Beenden des Programmes automatisch vom Betriebssystem freigegeben. Wenn ein BrookGPU-Programm in Octave eingebunden wird, dann erfolgt die Freigabe dieser Bereiche erst beim Beenden von Octave. Bei einem weiteren Aufruf desselben Programmes wird ein neuer Bereich reserviert. Dies führt dazu, dass immer weniger Arbeitsspeicher zur Verfügung steht, bis Octave aus Speichermangel abstürzt. Wenn mehrere verschiedene BrookGPU-Programme (oder BrookGPU in Kombination mit einem anderen OpenGL nutzenden Programm) in Octave geladen sind, kann es passieren, dass ein Programm mit den reservierten Teilen des anderen in Konflikt gerät und dies ebenfalls zu einem Absturz von Octave führt.

Diese Probleme sollten mit Hilfe eines in BrookGPU eingebauten Befehls zur Freigabe des Speichers lösbar sein, der aber noch fehlerhaft ist. Eine zukünftige Anpassung könnte für die vollständige Kompatibilität zwischen BrookGPU und Octave sorgen.

Die Programmierer von BrookGPU benutzen die GCC in der Version 3. Gravierende Änderungen in der aktuellen Version 4 von GCC führten dazu, dass sich BrookGPU nicht kompilieren ließ. Anhang B enthält eine maschinenlesbare Datei mit allen Unterschieden zwischen der offiziellen Subversion (SVN)-Revision 1889 [33] und der kompilierbaren und bugbereinigten Version. Hiermit ist es auch möglich, die oben genannte GCC 4 zu verwenden.

In Kombination mit den in Abschnitt 6.2 besprochenen Problemen enthält BrookGPU momentan noch zu viele Bugs, die eine effiziente Nutzung mit Octave verhindern. Der Speedup der vorgestellten Algorithmen zeigt, dass die Verwendung der Grafikkarte für Berechnungen durchaus sinnvoll ist. Für einen produktiven Einsatz von BrookGPU zusammen mit Octave ist es aber noch zu früh. Sobald der am Anfang des Abschnitts genannte Bug behoben ist, könnte es sich allerdings lohnen, erste Algorithmen umzusetzen.

7 Schlusswort

7.1 Zusammenfassung

Ziel dieser Diplomarbeit war es, die Kompatibilität von Octave mit verschiedenen Bibliotheken zum Rechnen auf der Grafikkarte zu überprüfen. BrookGPU wurde aufgrund der guten Unterstützung von verschiedenen Grafikkarten und wegen der zu Octave passenden Lizenz ausgewählt. Im Laufe der Arbeit stellte sich heraus, dass BrookGPU einige Bugs enthält, die schwer oder noch gar nicht behebbar sind. Dennoch zeigte sich schnell der Geschwindigkeitsvorteil bei der Berechnung mit der Grafikkarte gegenüber konventionellem Rechnen. Die Bibliothek CGiS wurde nur zu Beginn der Arbeit erwähnt, da sie aktuell noch in der Entstehung ist und von den Entwicklern deshalb nur eine Vorversion zur Verfügung gestellt werden konnte. Ein genauerer Blick war aus Zeitgründen nicht möglich, da CGiS auch zum Ende der Diplomarbeit noch nicht offiziell zur Verfügung stand.

7.2 Ausblick

Seit August 2008 erscheinen zunehmend Informationen über eine weitere GPGPU-Bibliothek. Das Open Computing Language (OpenCL) genannte Projekt wird von der Khronos-Gruppe entwickelt, die auch für OpenGL und OpenAL verantwortlich ist. Diese Projekte sind offene Industriestandards, an denen große Firmen des IT-Bereichs mitarbeiten. Die beiden Mitgliedsfirmen AMD und NVIDIA wollen ihre eigenen GPGPU-Bibliotheken AMD Stream (Brook+) und CUDA anpassen und OpenCL als Runtime unterstützen [34][35].

OpenCL ist eine Programmiersprache, die auf C basiert. Sie bietet Möglichkeiten, mit denen der Programmierer parallele Abläufe beschreibt. Die fertige Anwendung kann nicht nur auf dem Hauptprozessor genutzt werden, sondern wird beispielsweise auf der Grafikkarte ausgeführt oder auf allen Kernen eines Mehrprozessorsystems. Der Programmierer muss vorher nicht wissen, auf welchem System sein Programm ausgeführt wird, darum kümmert sich die Bibliothek. Die Idee ähnelt der von BrookGPU, wird hier aber noch erweitert, da die Hersteller selbst den Teil von OpenCL schreiben, der auf ihre Hardware zugreift. Dies ermöglicht eine bessere Optimierung des Programmes für spezielle Architekturen, ohne dass der Softwareentwickler sich darum kümmern muss.

Literaturverzeichnis

[1] Microsoft Corporation: *DirectX Developer Center (Programming One or More Streams (Direct3D 9))*. http://msdn.microsoft.com/en-us/directx/bb147299(VS.85).aspx, besucht: 05. 01. 09.

[2] Wolfgang Andermahr: *Test: ATi Radeon HD 4850 (CF) und HD 4870 (27/34)*. http://www.computerbase.de/artikel/hardware/grafikkarten/2008/test_ati_radeon_hd_4850_cf_hd_4870/27/#abschnitt_gpucomputing, besucht: 22. 02. 09.

[3] *Free Software Foundation - Licenses*. http://www.fsf.org/licensing/licenses/, besucht: 04. 02. 09.

[4] Gene M. Amdahl: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, Seiten 483–485, New York, NY, USA, 1967. ACM.

[5] Thomas Rauber und Gudula Rünger: *Multicore:: Parallele Programmierung*. Springer, Berlin, 2008, ISBN 978-3-540-73113-9.

[6] Thomas Rauber und Gudula Rünger: *Parallele Programmierung*. Springer, Berlin, 2007, ISBN 978-3-540-46549-2.

[7] AMD: *Terascale Graphics Engine*. http://static.pcinpact.com/pdf/docs/03ScottHartog-RV770Architecture-Final.pdf, besucht: 29. 01. 09.

[8] *The Quick PCI-Express 2.0 Guide*. http://www.10stripe.com/featured/quick/pci-express-2-0.php, besucht: 29. 01. 09.

[9] *GeForce GTX 280*. http://www.nvidia.com/object/product_geforce_gtx_280_us.html, besucht: 29. 01. 09.

[10] Sergey Lepilov: *Reinforcing the Line: ATI Radeon HD 4870 1024MB vs. Nvidia GeForce GTX 260 (216 SP) 896MB (page 8)*. http://www.xbitlabs.com/articles/video/display/radeon-hd4870-1024_8.html, besucht: 29. 01. 09.

[11] Wolfgang Andermahr: *Test: ATi Radeon HD 4870 X2 (27/30)*. http://www.computerbase.de/artikel/hardware/grafikkarten/2008/test_ati_radeon_hd_4870_x2/27/#abschnitt_leistungsaufnahme, besucht: 29. 01. 09.

[12] Martin Fischer und M. Bertuch: *Bildersprinter*. c't, (20):128–133, 2008, ISSN 0724-8679.

[13] *Intel® Core™ i7 Processor Extreme Edition*. http://www.intel.com/products/processor/corei7ee/specifications.htm, besucht: 02. 02. 09.

[14] *Blender Homepage*. http://www.blender.org/, besucht: 03. 02. 09.

[15] *techPowerUp! :: GPU Database : NVIDIA 7900 GTX*. http://www.techpowerup.com/gpudb/154/NVIDIA_7900_GTX.html, besucht: 20. 01. 09.

[16] *GNU General Public License - inoffizielle deutsche Übersetzung*. http://www.gnu.de/documents/gpl.de.html, besucht: 04. 02. 09.

[17] libSh. http://www.libsh.org/, besucht: 31. 12. 08.

[18] RapidMind. http://www.rapidmind.net/, besucht: 31. 12. 08.

[19] RapidMind: *RapidMind Corporate Fact Sheet*. http://www.rapidmind.net/images/media/RapidMindFactSheet.pdf, besucht: 31. 12. 08.

[20] *ATI Stream Computing FAQ*. http://ati.amd.com/technology/streamcomputing/faq.html#11, besucht: 04. 02. 09.

[21] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home.html, besucht: 31. 12. 08.

[22] CGiS. http://rw4.cs.uni-sb.de/projects/CGiS/, besucht: 31. 12. 08.

[23] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/, besucht: 31. 12. 08.

[24] NVIDIA: *GPU Gems 2 - Chapter 34. GPU Flow-Control Idioms*. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html, besucht: 06. 02. 09.

[25] Philipp Lucas: *CGiS: High-Level Data-Parallel GPU Programming*. Dissertation, Universität des Saarlandes, Saarbrücken, August 2007. http://rw4.cs.uni-sb.de/~phlucas/pubs/Diss.html, besucht: 06. 02. 09.

[26] Michael Vinther: *Independent Component Analysis of Evoked Potentials in EEG*. Technischer Bericht, Dänemarks Technische Universität - Fachbereich Elektrotechnik, 2002. http://logicnet.dk/reports/EEG/ICA.htm, besucht: 10. 02. 09.

[27] NVIDIA: *GPU Gems 2 - Chapter 46. Improved GPU Sorting*. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html, besucht: 05. 12. 08.

[28] Microsoft Corporation: *DirectX Developer Center (loop)*. http://msdn.microsoft.com/en-us/directx/bb174715(VS.85).aspx, besucht: 17. 11. 08.

[29] Aapo Hyvärinen: *Fast and Robust Fixed-Point Algorithms for Independent Component Analysis*. IEEE Transactions on Neural Networks, 10(3):626–634, 1999. http://www.cs.helsinki.fi/u/ahyvarin/papers/TNN99new.pdf, besucht: 15. 01. 09.

[30] Mike Houston: *Understanding GPUs Through Benchmarking*. http://www.gpgpu.org/s2007/slides/08-performance-overview.pdf, besucht: 17. 11. 08.

[31] Microsoft Corporation: *DirectX Developer Center (Flow Control Limitations)*. http://msdn.microsoft.com/en-us/directx/bb219848(VS.85).aspx#Instruction_Depth_Count_for_ps_3_0, besucht: 03. 12. 08.

[32] *Hinweis auf* ***int****-Support in BrookGPU SVN Changelog.* `http://brook.svn.sourceforge.net/viewvc/brook/branches/Niall'sUpdate/ChangeLog.txt?annotate=1865#l46`, besucht: 21. 02. 09.

[33] *BrookGPU SVN repository.* `http://brook.svn.sourceforge.net/viewvc/brook/branches/Niall%27s%20Update/`, besucht: 15. 01. 09.

[34] AMD: *AMD Drives Adoption of Industry Standards in GPGPU Software Development*, 06.08.08. `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~127451,00.html`, besucht: 16. 12. 08.

[35] NVIDIA: *NVIDIA Adds OpenCL To Its Industry Leading GPU Computing Toolkit*, 09.12.08. `http://www.nvidia.com/object/io_1228825271885.html`, besucht: 16. 12. 08.

Anhang A

Vorbereitung für Kompilation

Für die Kompilation von BrookGPU und Octave-Plugins benötigt man zusätzliche Dateien. Hier werden dafür Paketnamen angegeben, wie sie beispielsweise im Paketverwaltungssystem von Debian zu finden sind. Je nach System kann es vorkommen, dass nicht alle Pakete installiert werden müssen. Folgende Aufzählung ist nur eine Richtlinie:

nvidia-cg-toolkit, octave3.0-headers, make, libglu1-mesa-dev, bison, flex, libxt-dev

Kompilation

Nach den folgenden Schritten liegen die fertig kompilierten Plugins für Octave im Ordner `brookgpu/bin/`. Sie haben die Dateiendung `*.oct`. Die Befehlsfolge ist für das direkte Kopieren in eine Bash-Shell geeignet.

```
### herunterladen der Quellen - komplett auf CD verfuegbar
svn co https://brook.svn.sourceforge.net/svnroot/brook/branches/\
Niall%27s%20Update/ brookgpu
svn co https://svnserv.cbs.mpg.de/svn/psy/lindner/diplomarbeit/brook/
cd brookgpu
patch -p 0 < ../brook/brookgpu_bugfixes.diff
cd ..

### kompilieren von BrookGPU
cd brookgpu/config
make brook                  #- auf Brook-Makefile umstellen
cd ..
make brcc runtime           #- BrookGPU Compiler und Runtimes erstellen
cd config
make octave                 #- auf Octave-Makefile umstellen
cd ../..

### kompilieren eines Projektes
cd brook/progs/<projekt>    #- in ein Projektverzeichnis wechseln
make                        #- wenn kein Octave benutzt wird, wie oben
                            #  gezeigt auf Brook-Makefile umstellen
cd ../../../brookgpu/bin    #- Verzeichnis mit den ausfuehrbaren Dateien
```

Vorbereitung für Benutzung

Auf dem Rechner des Benutzers ist die Installation oben genannter Pakete und Dateien nicht erforderlich. Stattdessen muss die 3D-Beschleunigung aktiviert werden. Besitzer von ATI-Grafikkarten sollten zwischen den Paketen `xserver-xorg-video-radeonhd` und `fglrx-modules-2.6-686` wählen. Für NVIDIA-Grafikkarten existiert das Paket `nvidia-kernel-2.6-686`. Alternativ können die aktuellen Treiber auch direkt von der Herstellerseite heruntergeladen werden. Nach der Installation ist eventuell noch die Anpassung von Dateien wie etwa `/etc/X11/xorg.conf` nötig, damit der Grafiktreiber vom Computer auch genutzt wird.

Der Nutzer kann recht einfach testen, ob die 3D-Beschleunigung aktiviert ist. Wenn das Kommando `glxgears` eine sich bewegende 3D-Darstellung anzeigt, ist wahrscheinlich alles funktionstüchtig. Der Befehl wird bei dem Paket `mesa-utils` mitgeliefert.

Wie in den Abschnitten Vorbereitung für Kompilation und Kompilation erklärt wurde, muss man die Plugins für Octave zuerst kompilieren. Danach können sie in einen Ordner abgelegt werden, der von Octave aus erreichbar ist. Der Anwender startet Octave nun mit Hilfe des Befehls `BRT_RUNTIME=ogl octave` und kann das Plugin wie jeden anderen Befehl nutzen.

Anhang B

In der Online- und CD-Version liegen zusätzliche Dateien bei, deren Druck aufgrund der Größe nicht sinnvoll ist:

brook/brookgpu_bugfixes.diff enthält alle Unterschiede zwischen der offiziellen SVN-Revision 1889 [33] und der kompilierbaren und bugbereinigten Version.

brook/progs/fastica/ ist eine unabhängige Komponentenanalyse mittels fastICA-Algorithmus in BrookGPU.

brook/progs/matmatmul_f4/ enthält eine schnelle Matrizenmultiplikation in BrookGPU mit `float4`-Werten.

brook/progs/nmf_f4/ beinhaltet eine nicht-negative Matrizenfaktorisierung in BrookGPU und nutzt eine schnelle Matrizenmultiplikation mit `float4`-Werten.

brook/progs/sort/ ist eine Implementation zum Sortieren von Elementen mit BrookGPU (leider langsamer als per CPU). Es könnte nützlich sein, wenn das Sortieren als Zwischenschritt benötigt wird und die Transferkosten zur CPU zu hoch sind.

brook/test/ enthält verschiedene Testprogramme, die aussortiert wurden, da sie entweder sehr langsam sind oder eine schnellere Variante existiert.

brookgpu/ beinhaltet BrookGPU in der SVN-Revision 1889, eventuell mit Patches aus brook/brookgpu_bugfixes.diff.