UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

COMPUTERVISUALISTIK
Institut für Computervisualistik

Untersuchung und Bewertung von Standardalgorithmen zur parallelen Programmierung auf der GPU

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Elias Zervudakis

Erstgutachter:	Prof. Dr. Stefan Müller Institut für Computervisualistik
Zweitgutachter:	M.Sc. Kevin Keul Institut für Computervisualistik

Koblenz, im Februar 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

Koblenz, 19.02.2018

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die vorliegende Arbeit gibt einen Überblick über die Rahmenbedingungen der Programmierung von Grafikkarten. Dazu werden die zur wichtigsten am Markt vorhandenen Application Programming Interfaces (APIs) vorgestellt und miteinander verglichen. Anschließend werden zwei Standardalgorithmen aus der Datenverarbeitung, Prefix Sum und Radixsort vorgestellt und im Hinblick auf die Implementierung mit paralleler Programmierung auf der GPU zu untersucht. Beide Algorithmen wurden unter Nutzung der OpenGL-API und OpenGL Compute Shadern implementiert. Abschließend wurden die Ausführungszeiten der beiden Algorithmen miteinander verglichen.

Abstract

The present thesis gives an overview of the general conditions for the programming of graphics cards. For this purpose, the most important Application Programming Interfaces (APIs) available on the market are presented and compared. Subsequently, two standard algorithms from the field data processing, prefix sum and radixsort are presented and examined with regard to the implementation with parallel programming on the GPU. Both algorithms were implemented using the OpenGL-API and OpenGL compute shaders. Finally, the execution times of the two algorithms were compared.

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Elias Zervudakis (Matr.-Nr. 213 100 937)

Thema: Untersuchung und Bewertung von Standardalgorithmen zur parallelen Programmierung auf der GPU

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt, wie im Bereich der GPU. Während die Grafikkarten anfangs vor allem für die Rasterisierung von grafischen Elementen optimiert war, so ist die Hardware heute sehr viel offener für GPGPU (general purpose GPU) Anwendungen konzipiert. Bei der Programmierung der GPU stellen sich aber schnell Herausforderungen in der parallelen Umsetzung von Standardalgorithmen wie z.B. das Suchen oder das Sortieren in großen Datenmengen. Die vorliegende Arbeit setzt hier an und untersucht verschiedene Standardalgorithmen auf ihre Implementierbarkeit und ihre Performance.

Ziel dieser Arbeit ist es, verschiedene Standardalgorithmen auszusuchen, um die diese im Hinblick auf die Implementation mit paralleler Programmierung auf der GPU zu untersuchen. Angedacht sind dabei Algorithmen wie das Suchen und das Sortieren in großen Datenmengen und Prefix sum. Die verschiedenen Algorithmen sollen geprüft, implementiert und schließlich in Bezug auf den Performancegewinn bei variabler Datengröße verglichen bzw. kategorisiert werden.

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in die Programmierung der GPU
2. Auswahl von Standardalgorithmen zur parallelen Programmierung
3. Implementation und Vergleich der Verfahren
4. Bewertung der Verfahren im Hinblick auf die Grenzen und Möglichkeiten
5. Dokumentation

Koblenz, den 17.07.2017

S. Müller

- Elias Zervudakis - - Prof. Dr. Stefan Müller-

Inhaltsverzeichnis

1 Einleitung

Dieser Abschnitt erläutert zuächst die Motivation zur Erstellung dieser Arbeit und die damit verbundene Zielsetzung. Anschließend wird auf die Vorgehensweise zur Erstellung dieser Arbeit, die gewonnen Ergebnisse sowie die Gliederung der Arbeit eingegangen. Zuletzt wird die Ausgangssituation am Markt für Grafikkarten besprochen.

1.1 Motivation

In der Informationsverarbeitung wächst die Menge an gespeicherten Daten durch die immer größer werdende Anzahl an Geräten und Sensoren, sowie deren Vernetzung. Auch die Speicherung dieser Daten wird durch kontinuierlich fallende Preise für Speicherplatz[And] immer kostengünstiger, sodass kein Anreiz besteht, diese Datenmengen zu löschen. Dementsprechend wird aber auch eine immer höhere Rechenleistung zur Verarbeitung benötigt, um beispielsweise nach entsprechenden Daten zu suchen oder diese zu ordnen. Die Folgen ist eine signifikante Kostensteigerung. Da sequentielle Standardalgorithmen zur Datenverarbeitung nur schlecht bei wachsenden Datenmengen skalieren, kann die Verarbeitung der Daten eine lange Zeit in Anspruch nehmen.

Ein Ansatz zur Bewältigung dieses Problems ist die Parallelisierung von Algorithmen, um die Datenmengen in schnellerer Zeit mit geringerer Rechenleistung zu bewältigen. Diese Algorithmen müssen anschließend in paralleler Programmierung auf passender Hardware implementiert werden.

Hier sind es vor allem Grafikkarten, die für die Ausführung von parallelen Programmen genutzt werden. Grafikkarten waren anfangs vor allem für die Rasterisierung von grafischen Elementen optimiert und boten nur wenige Möglichkeiten, andere Programme auszuführen. Heute sind sie für diese nicht-grafikbezogene Anwendungen sehr viel offener konzipiert.

1.2 Zielsetzung

Die vorliegende Arbeit beschäftigt sich mit der parallelen Programmierung auf Grafikkarten und untersucht zwei Standardalgorithmen, die anschließend parallel implementiert werden. Sie zeigt die Grundlagen der parallelen Programmierung auf Grafikkarten und die Vorgehensweise zur Implementieruneng der beiden Standardalgorithmen auf. Bei der Programmierung der Graphics Processing Unit (GPU) gibt es verschiedene Herausforderungen in der parallelen Umsetzung von Standardalgorithmen, die im Verlauf der Arbeit aufgezeigt werden sollen. Im Anschluss an die Implementierung sollen die beiden Standardalgorithmen auf ihre Performance hin untersucht werden. Dazu wird ihre Ausführungsgeschwindigkeit bei variabler Datengröße verglichen. Hier wird insbesondere zwischen kleinen und vergleichsweise großen Datensätzen mit bis zu einer Million Elementen unterschieden.

1.3 Vorgehensweise

Zur Umsetzung der Ziele wurde zunächst ein Überblick über die zur Programmierung der GPU nutzbaren Programmiersprachen und Application Programming Interfaces (APIs) geschaffen. Auf dieser Grundlage wurde sich für die Nutzung von Open Graphics Library (OpenGL) und OpenGL Compute Shadern entschieden. Da bisher keine Erfahrungen mit C++ und der OpenGL-Shader Programmiersprache OpenGL Shading Language (GLSL) vorhanden waren, wurden sich diese angeeignet.

Im Anschluss wurden verschiedene Standardalgorithmen betrachtet, die in der Datenverarbeitung eingesetzt werden. Die Wahl fiel hierbei auf Prefix Sum und Radixsort, wobei Radixsort intern Prefix Sum für einzelne Rechenschritte nutzt.

Beide Algorithmen wurden erlernt und anschließend als parallele Programme implementiert. Diese Implementierungen wurden anschließend auf ihre Laufzeit hin untersucht.

1.4 Ergebnisse

Die Implementierungen zeigten mehrere Herausforderungen auf, die bei der parallelen Programmierung auf der GPU zu bewältigen sind. Der weitere Umgang mit diesen Herausforderungen wird in Abschnitt 5 auf Seite 31 besprochen.

Die Ergebnisse der Laufzeitvergleiche haben den Unterschied zwischen paralleler und nicht-paralleler Implementierung deutlich gemacht. Sie werden im Detail in in Abschnitt 4 auf Seite 25 erläutert.

1.5 Gliederung

Zu Beginn der Arbeit wird in Abschnitt 2 auf Seite 5 auf die verschiedenen Begriffe im Kontext dieser Arbeit eingegangen. Dazu gehören die Grundlagen im Bereich der Grafikkarten sowie die verschiedenen APIs, die zur Programmierung der GPU genutzt werden können. Anschließend werden die Kriterien für die Auswahl der Standardalgorithmen und der API erläutert. In Abschnitt 3 auf Seite 11 wird an exemplarischen Teilen des Programmcodes gezeigt, wie die Algorithmen implementiert wurden. In Abschnitt 4 auf Seite 25 werden die gewonnen Erkentnisse aufgezeigt und auf die Laufzeiten der Algorithmen unter verschiedenen Bedingungen und Eingabedaten eingegangen. Abschließend wird in Abschnitt 5 auf Seite 31 ein Fazit gezogen und ein Ausblick auf die weitere Entwicklung im Bereich der parallelen Programmierung auf der GPU gegeben.

1.6 Ausgangssituation

Die bisherige Entwicklung der Leistungsfähigkeit von GPUs lässt darauf schließen, dass sich der Markt für leistungsfähige GPUs in den kommenden Jahren stetig weiter entwickeln wird.

Nach Moore's Law verdoppelt sich die Anzahl an Transistoren pro Flächeneinheit regelmäßig in einem Zeitraum von zwei Jahren [BM06]. Abbildung 1 zeigt diese Verdoppelung der Anzahl an Transistoren Anhand von NVIDIA GPUs und Intel Central Processing Units (CPUs) im Zeitraum der Jahre 1997 bis 2016.

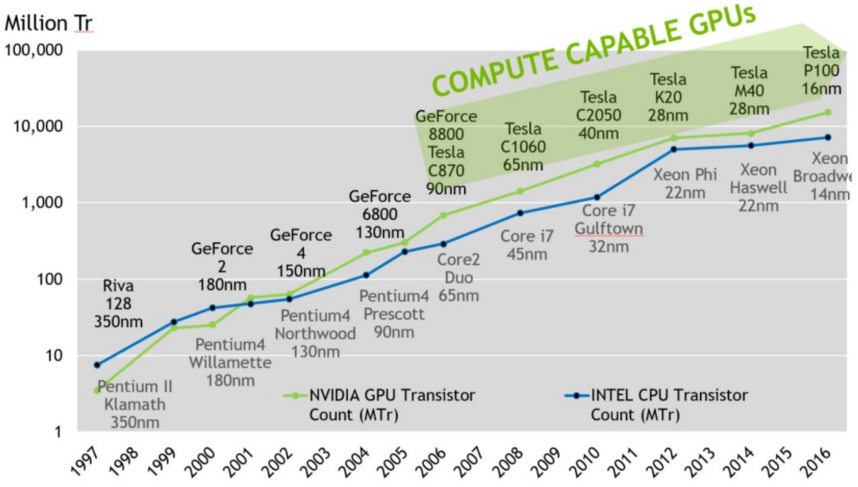


Abbildung 1: Entwicklung der Anzahl an Transistoren in GPUs und CPUs[Baj17]

Genauso wie die Anzahl der Transistoren pro Flächeneinheit entwickelt sich auch die Rechenleistung der Prozessoren stetig. Abbildung 2 auf der nächsten Seite zeigt die Entwicklung der theoretischen Höchstleistung verschiedener NVIDIA GPUs, AMD GPUs und Intel CPUs in GFLOP/sec bei einfacher Genauigkeit im Zeitraum der Jahre 2007 bis 2016. Die CPU-Werte beziehen sich dabei auf einen einzelnen Prozessorsockel.

Diese Entwicklungen führen zu einer regelmäßigen Reduzierung der Kosten der Rechenleistung pro Dollar [Rup]. Es wird daher immer Attraktiver, Programme mit hohen Anforderungen an die Berechnungen zu nutzen oder diese Berechnungen überhaupt erst durchzuführen.

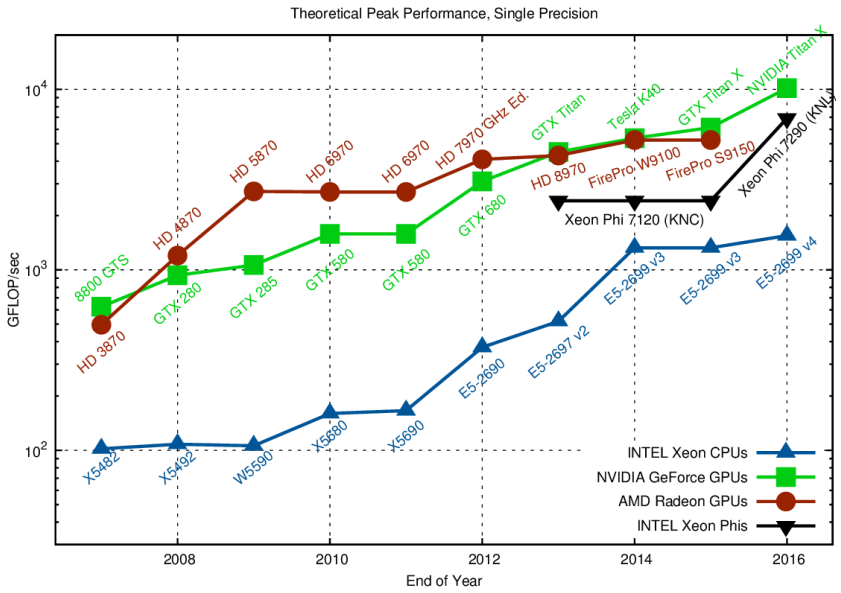


Abbildung 2: Entwicklung der theoretischen Höchstleistung verschiedener Prozessoren in GFLOP/sec[Rup]

2 Grundlagen und Begriffe

Dieser Abschnitt geht auf die grundlegenden Begriffe im bereich der Grafikkarten und der parallelen Programmierung ein. Dabei werden die verschiedenen verfügbaren APIs vorgestellt und verglichen.

2.1 Graphics Processing Unit (GPU)

Die GPU ist ein Prozessor, der für die Berechnung von Grafiken ausgelegt ist. Sie ist entweder zusammen mit der CPU in einem Chip oder unabhängig von dieser in einem Computer verbaut. In Letzterem ist im Regelfall eine CPU vorhanden. Eine GPU muss daher nicht alle Aufgaben erfüllen und kann sich auf eine Aufgabe spezialisieren. In einer Grafikkarte können mehrere GPUs vorhanden sein, ebenso können mehrere Grafikkarten gleichzeitig in einem System genutzt werden[PH11].

Die Nutzung von GPUs für Aufgaben, die über die Berechnung von Grafiken hinausgeht, wird als General Purpose Computation on Graphics Processing Unit (GPGPU) bezeichnet. Dazu zählen verschiedenste Anwendungsfälle wie physikalische Simulationen, neuronale Netze und Kryptographie.

Der Vorteil der Nutzung von GPGPU wird bei parallel ausgeführten Rechenaufgaben genutzt. GPUs bieten eine höhere Rechenleistung sowie eine höhere Speicherbandbreite als CPUs. [PH11]

CPUs nutzen Cache mit mehreren unterschiedlich schnellen Hierarchieebenen, um höhere Latenzen beim Speicherzugriff abzufangen. GPUs dagegen nutzen nur eine Speicherstruktur für alle Aufgaben. Stattdessen wird durch die Nutzung von Parallelisierung der Fokus auf hohe Bandbreite statt niedriger Latenz gelegt. Die Speicherstrukturen auf GPUs unterscheiden sich daher von denen auf CPUs. [PH11]

2.1.1 Architektur

Für diese Arbeit wird eine NVIDIA GeForce GTX 560, die auf der Fermi-Architektur basiert, genutzt. Abbildung 3 auf der nächsten Seite zeigt den Aufbau der Fermi-Architektur. Die 16 Streaming Multiprocessors (SMs) der Fermi-Architektur sind um einen gemeinsamen L2 Cache positioniert. Jeder SM besteht aus 32 Compute Unified Device Architecture (CUDA)-Kernen. Die einzelnen SM bestehen aus einem orangenen Teil (Scheduler und Dispatch), einem grünen Teil (Execution Einheiten) und einem hellblauen Teil (Register und L1-Cache). Abbildung 4 auf Seite 7 zeigt den Aufbau eines einzelnen Fermi SM. Er besteht aus 32 CUDA-Kernen, 16 Load and Store Units (LD/ST) und vier Special Function Units (SFU). Threads innerhalb einer lokalen Work Group, die durch einen SM ausgeführt werden, können miteinander kooperieren. [NVI]

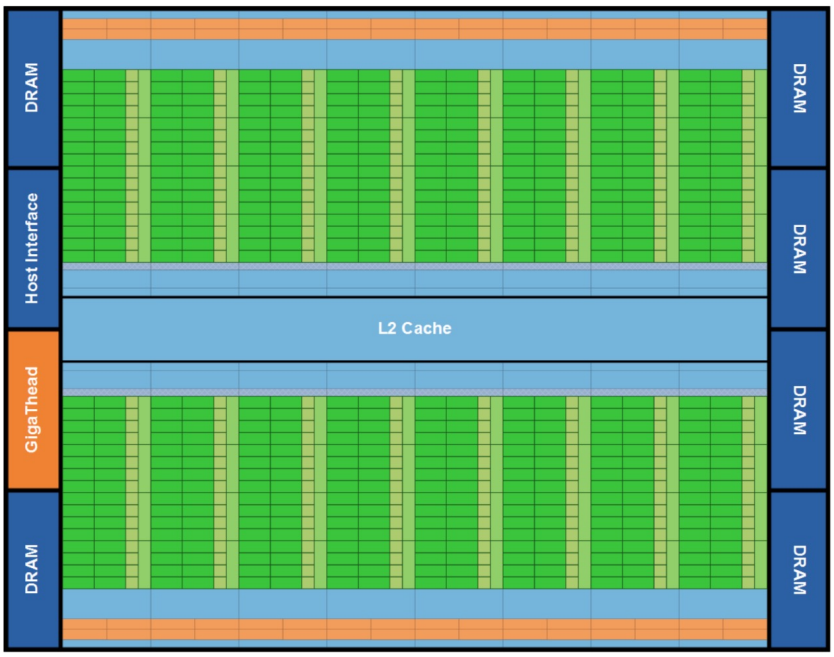


Abbildung 3: Aufbau der Fermi-Architektur. [NVI]

2.2 Einfache und Doppelte Genauigkeit

Beim Rechnen mit Gleitkommazahlen auf der GPU unterscheidet man zwischen einfacher und doppelter Genauigkeit der Zahlen. Nach dem IEEE 754 Standard belegen Zahlen in einfacher Genaugkeit 32 Bit Speicher, Zahlen mit doppelter Genauigkeit benötigen mit 64 Bit doppelt so viel.

Von den 32 Bit wird 1 Bit für das Vorzeichen S , 8 Bit für den Exponenten e und 23 Bit für die Mantisse m genutzt. Bei 64 Bit wird 1 Bit für das Vorzeichen, 11 Bit für den Exponenten und 52 Bit für die Mantisse genutzt. Bei der Nutzung von Dezimalzahlen fürt dies zu 6 bzw. 16 speicherbaren Nachkommastellen. Die Berechnung wird in Gleichung (1) gezeigt.

$$(-1)^S \cdot m \cdot 2^e \tag{1}$$

Abhängig vom Einsatzzweck nutzt man für Berechnungen einfache oder doppelte Genauigkeit. GPUs waren lange nur für Berechnungen mit einfacher Genauigkeit ausgelegt, auch heute noch sind solche Berechnungen siginifikant schneller. Zur Veranschaulichung dieses Geschwindigkeitsunterschieds gibt es verschiedene Untersuchungen. Itu et al. haben beispielsweise die Geschwindigkeit mehrere Algorithmen, darunter die Multiplikation zweier Matrizen und die Navier-Stokes-Gleichungen für inkompressible Fluide auf einer NVIDIA GTX260 Grafikkarte verglichen. Die Ergebnise werden in Abbildung 5 auf Seite 8 dargestellt. Bei beiden Algorithmen ist die Ausführungszeit mit einfacher Genauigkeit signifikant schneller. [Itu+11]

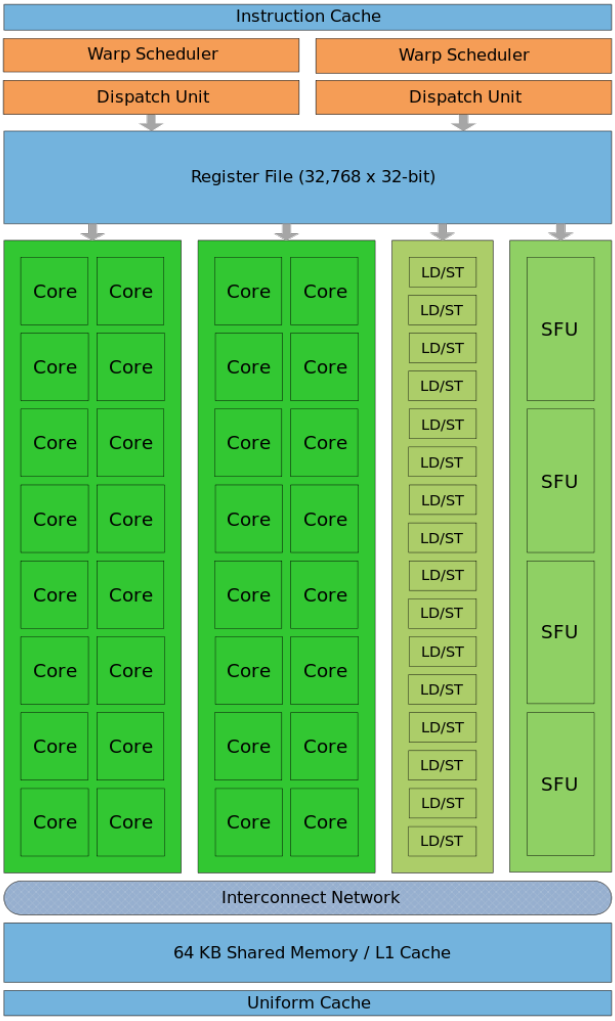


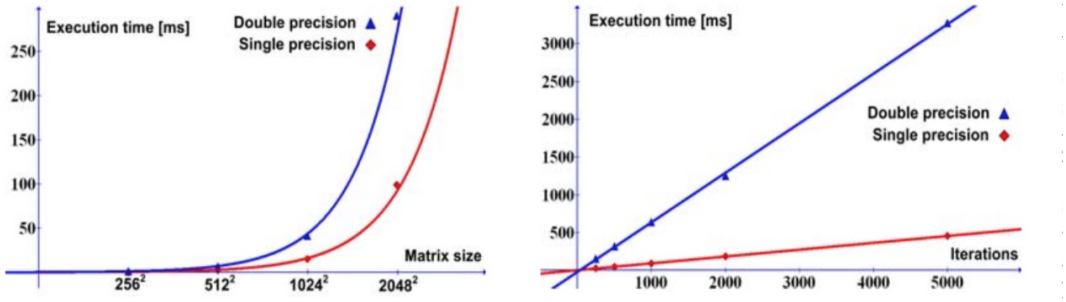
Abbildung 4: Aufbau eines einzelnen SM[NVI]

2.3 Application Programming Interface (API)

Ein API ist eine Sammlung von Funktionen und Datenstruktur-Definitionen, die von Dritten als Schnittstelle zur Anbindung und Nutzung eines Systems verwendet werden kann. Sie dient dem Zugriff auf bestimmte Teile einer Software oder bestimmter Hardware. [PH11]

Teil einer API ist eine genaue Dokumentation aller Funktionen und Datenstrukturen. Diese Dokumentation wird üblicherweise durch ein Konsortium oder Unternehmen verwaltet und herausgegeben. [PH11]

Die meistgenutzten APIs zur Programmierung der GPU sind für höhere Programmiersprachen geschrieben, das heißt der Programmcode wird nicht unmittelbar von der GPU ausgeführt, sondern muss erst durch einen Compiler in Maschinenspra-



(a) Matrixmultiplikations-Algorithmus (b) Navier-Stokes-Algorithmus

Abbildung 5: Vergleich der Ausführungszeiten mit einfacher und doppelter Genauigkeit zweier Algorithmen[Itu+11]

che übersetzt werden. Der Zugriff auf die GPU durch eine API in einer höheren Programmiersprache bietet den Vorteil eines abstrakteren Programmcodes. Da sich die APIs und die Programmiesprachen fortwährend weiterentwickeln, können Entwickler ständig neue Architekturen und Funktionalitäten ausprobieren und nutzen. GPUs bieten aus diesem Grund Abwärtskompatibilität für bereits kompilierten Programmcode. [PH11]

In dieser Arbeit liegt der Fokus auf APIs, die den Zugriff auf die GPUs bieten.

2.3.1 Open Graphics Library (OpenGL)

Die OpenGL ist eine offene, hardwareunabhängige API zur Programmierung von GPUs.

Die erste Version der OpenGL-API wurde im Jahr 1992 von Entwicklern bei Silicon Graphics, Inc. (SGI) als Alternative zur eigenen proprietären Grafik-API Integrated Raster Imaging System Graphics Library (IRIS GL) entwickelt. IRIS GL erlaubte den Zugriff auf die GPU der durch das Unternehmen angebotenen Rechner. Grund für die Neuentwicklung war unter anderem die fehlende formale Spezifikation. SGI lizensierte die erste OpenGL-Version an Wettbewerber und setzte ein Industriekonsortium zur Weiterentwicklung des OpenGL-Standards ein, das OpenGL Architecture Review Board. Im Jahr 2006 gingen die Aufgaben des Konsortiums an die Khronos Group[Grod] über, die seitdem den OpenGL-Standard verwaltet [Groa].

Seit der ersten Veröffentlichung hat der OpenGL-Standard weitere umfangreiche Erweiterungen erfahren. So wurde in Version 2.0 die Programmiersprache GLSL eingeführt, mit der die Shader-Schritte in der Grafikpipeline programmiert werden können. Version 4.3 des Standards führte Compute-Shader ein. Diese Compute-Shader werden genauso wie andere Shader in der GLSL geschrieben und können unabhängig von der Grafikpipeline Daten bearbeiten. Sie sind für die Nutzung im GPGPU-Kontext vorgesehen [Groa].

Derzeit ist OpenGL die einzige API, die die geläufigsten am Markt vorhandenen

Betriebssystemen unterstützt. Durch OpenGL ES werden seit 2003 auch eingebettete Systeme wie Smartphones abgedeckt.

2.3.2 Open Computing Language (OpenCL)

Die Open Computing Language (OpenCL) ist ein offenes und hardwareunabhängiges Framework zur parallelen Programmierung von CPUs, GPUs und anderer Hardware wie beispielsweise Field Programmable Gate Arrays (FPGAs). Das Framework setzt sich aus Programmiersprachen zur Programmierung der Prozessoren und einer API zusammen.

In OpenCL geschriebene Programme können daher auf einer vielzahl verschiedener Plattformen, die sich auch aus verschiedenen Typen von Prozessoren zusammensetzen können, ausgeführt werden.

Die erste Version der OpenCLs wurde 2008 durch die Khronos Group[Groc] veröffentlicht. Seitdem wurden regelmäßig neue Versionen freigegeben, die OpenCL um Funktionen erweitern. Zu den umfangreichsten Erweiterungen gehört neben der Integration von Standard Portable Intermediate Representation (SPIR)-V die Einführung der Programmiersprache OpenCL C++ Kernel Language. Im Mai 2017 hat die Khronos Group angekündigt, OpenCL mit der Vulkan-API zusammenzuführen. [Grob]

2.3.3 Compute Unified Device Architecture (CUDA)

CUDA bezeichnet hier eine prorietäre API, die von NVIDIA entwickelt wird.

Die erste version von CUDA wurde im Jahr 2007 veröffentlicht. Die API wird häufig um neue Funktionen ergänzt und unterstützt alle Besonderheiten der NVIDIA Grafikkarten. Dieser Vorteil ist gleichzeitig ein Nachteil bei der Nutzung von CUDA, da es nicht möglich ist, GPUs anderer Hersteller zu nutzen.

2.3.4 DirectX

DirectX ist ein proprietäre Sammlung verschiedener APIs aus dem Multimedia-Umfeld. Ein bedeutender Teil von DirectX ist Direct3D, eine API zur Programmierung von GPUs. DirectX wird von Microsoft verwaltet und ist nur für die Nutzung auf Microsoft-Plattformen wie Windows ausgelegt.

Die erste Version von DirectX wurde im Jahr 1995 durch Microsoft veröffentlicht. Seitdem gab es regelmäßig neue Versionen, die den Funktionsumfang erweiterten.

Durch die Beschränkung auf Microsoft-Plattformen können Programme, die DirectX nutzen, nicht ohne hohen Aufwand auf anderen Systemen ausgeführt werden. Da Windows-Plattformen im privaten Umfeld eine hohe Verbreitungsrate haben [Sta], sind in der Vergangenheit viele Computerspiele unter Nutzung von DirectX programmiert worden.

2.3.5 Vulkan

Vulkan ist eine offene, plattformübergreifende und hardwareunabhängige API zur Programmierung von GPUs.

Die erste Version von Vulkan wurde im Jahr 2016 durch die Khronos Group[Grof] veröffentlicht. Derzeit unterstützen nur neuere GPU-Generationen Vulkan.

Verglichen mit OpenGL abstrahiert Vulkan vergleichsweise wenig von der Hardware und ermöglicht dadurch eine höhere Rechenleistung. Gemein mit OpenCL hat Vulkan die Unterstützung der Zwischensprache SPIR-V.

3 Implementierung

Dieser Abschnitt beschreibt den Vorgang der Implementierung. Dazu gehört zunächst die Auswahl der API und die Vorstellung der Grundlagen zur Nutzung dieser API. Anschließend werden die zwei ausgewählten Standardalgorithmen vorgestellt und erläutert. Zuletzt wird auf die programmiertechnische Umsetzung dieser Algorithmen eingegangen.

3.1 Auswahl einer geeigneten API

Zur Umsetzung der ausgewählten Algorithmen wurden die in Abschnitt 2.3 auf Seite 7 genannten APIs gegeneinander abgewogen.

Die Nutzung von Direct3D hätte zu einer Abhängigkeit von Microsoft-Plattformen geführt, ähnlich wäre die Nutzung von CUDA nur mit Grafikkarten der Firma NVIDIA möglich gewesen. OpenGL, OpenCL und Vulkan dagagen sind Hersteller- und Plattformunabhängig, sodass die Programme auf unterschiedlichen Rechnern entwickelt, getestet und ausgeführt werden könnten. Vulkan wäre durch seine höhere Rechenleistung gut geeignet gewesen, leider ist die Menge an untersützten GPUs bisher sehr klein. Gegen OpenCL sprach der aufwendigere Prozess zur Einrichtung, darunter die Installation von speziellen Treibern und Bibliotheken [Bai].

Für OpenGL und die Nutzung von Compute Shadern sprach die gute Dokumentation und das in der Arbeitsgruppe Computergrafik vorhandene Wissen. Die Entscheidung fiel damit auf diese Kombination.

3.2 OpenGL Compute Shader

OpenGL Compute Shader ähneln den in der Computergrafik genutzten Shadern in vielen Punkten. Shader ist eine allgemeine Bezeichnung für Programme, die auf GPUs ausgeführt werden. Sie werden in speziellen Programmiersprachen programmiert, im Fall von OpenGL und Compute Shadern ist dies die GLSL. Durch die Nutzung von GLSL können Entwickler, die bereits mit OpenGL und den verschiedenen Grafik Shadern vertraut sind, ohne hohe Einstiegshürdern GPGPU Programme schreiben.

Compute Shader können auf die selben Daten wie andere Shader zugreifen, dazu gehören beispielsweise Texturen und Atomic Counter. Da Compute Shader im Gegensatz zu anderen Shadern keine vordefinierten Ein- und Ausgaben haben, müssen andere Wege genutzt werden, um Daten zu übertragen [Sel+].

3.2.1 Erstellung und Ausführung

Compute Shader werden ähnlich wie andere Shader erstellt. Mit dem Befehl *GLuint shad glCreateShader(GL_COMPUTE_SHADER)* wird ein Shaderobjekt des Shadertyps *GL_COMPUTE_SHADER* erstellt, das anschließend mit dem Befehl *glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)*

geladen werden kann. Der Befehl *glCompileShader(GLuint shader)* kompiliert den Shader. Mit *glCreateProgram()* wird ein Compute Program erzeugt, an den der kompilierte Shader mit *glAttachShader(GLuint program, GLuint shader)* angehängt wird. Mit *glLinkProgram(GLuint program)* wird das Compute Program gelinkt, der Shader kann im Anschluss mit *glDeleteShader(GLuint shader)* gelöscht werden.
Zur Ausführung eines Compute Shaders wird das Compute Program mit *glUseProgram(GLuint program)* als das aktuelle Programm gebunden. Alle anschließend aufgerufenen Befehle beziehen sich damit auf das aktuelle Programm. Die Ausführung wird mit dem Befehl *void glDispatchCompute(GLuint num_groups_x, GLuint num_groups_y, GLuint num_groups_z)* gestartet. Wird die OpenGL Erweiterung *ARB_compute_variable_group_size* genutzt, lautet der Befehl *void glDispatchComputeGroupSizeARB(GLuint num_groups_x, GLuint num_groups_y, GLuint num_groups_z , GLuint group_size_x, GLuint group_size_y, GLuint group_size_z)*.
Die drei Parameter *GLuint num_groups_x*, *GLuint num_groups_y* und *GLuint num_groups_z* geben die Größe der globalen Work Group, die drei Parameter *GLuint group_size_x*, *GLuint group_size_y* und *GLuint group_size_z* die der lokalen Work Group an und werden in Abschnitt 3.2.2 näher erläutert.
Compute Shader können nicht mit anderen Shadertypen gemischt werden. Ein Programm kann entweder aus Grafikshadern wie beispielsweise Vertex Shadern oder aus Compute Shadern bestehen. In Quelltext 1 auf Seite 13 wird die Erstellung eines Compute Programs und das Ausführen des Compute Shaders in einem Minimalbeispiel gezeigt.

3.2.2 Globale und lokale Work Groups

Compute Shader werden in Work Groups ausgeführt. Es wird zwischen globalen und lokalen Work Groups unterschieden. Die lokale Work Group kann als die Anzahl der Threads, die gleichzeitig ausgeführt werden, angesehen werden.
Die Dimensionen der globalen Work Groups werden bei der Ausführung des Shaders mit den Parametern *GLuint num_groups_x*, *GLuint num_groups_y* und *GLuint num_groups_z* festgelegt. Abbildung 6 auf Seite 14 zeigt die Aufteilung der drei Dimensionen.
Die Dimension der lokalen Work Groups werden im Shader mit *layout(local_size_x = X, local_size_y = Y, local_size_z = Z) in* deklariert, dabei stehen X, Y und Z für die Werte der einzelnen Dimensionen. Der voreingestellte Wert liegt für alle drei bei 1. Eine lokale Work Group mit der Deklaration *layout(local_size_x = 32, local_size_y = 32) in* resultiert also in $32 * 32 * 1 = 1024$ Aufrufen.
Mit dem Befehl *void glGetProgramiv(GLuint program, GL_COMPUTE_WORKGROUP_SIZE, GLint *params)* kann die Dimension der lokalen Work Groups eines bestehenden Compute Programs abgefragt werden. Wird die OpenGL Erweiterung *ARB_compute_variable_group_size* genutzt, kann die Dimension der lokalen Work Group statt im Shader im Aufruf mit den Parametern

```
GLuint example_shader = glCreateShader(GL_COMPUTE_SHADER);

static const GLchar * source[] = {
    "#version 460 core                                          \n"
    "                                                           \n"
    "layout (local_size_x = 1024, local_size_y = 1) in;         \n"
    "                                                           \n"
    "void main() {                                              \n"
    "    // Führe etwas aus                                     \n"
    "}                                                          \n"
};

glShaderSource(example_shader, 1, &source, 0);
GLuint program = glCreateProgram();
glCompileShader(shader)
glAttachShader(program, shader);
glLinkProgram(program);
glDeleteShader(shader);

glUseProgram ( program );
glDispatchCompute( 1, 1, 1 ); // x, y, z Größe der Work Group
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Quelltext 1: Erstellung eines Compute Programs sowie dispatchen und synchronisieren des Compute Shaders

GLuint group_size_x, GLuint group_size_y und *GLuint group_size_z* festgelegt werden. Im Shader wird dann das Layout mit *layout (local_size_variable) in* deklariert.
Die minimale Anzahl an globalen Work Groups in jeder der drei Dimensionen liegt bei 65535. Die maximale Anzahl kann mit dem Befehl *void glGetIntergeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, GLuint index, GLint* params* abgefragt werden. Die minimale Anzahl an lokalen Work Groups liegt bei 1024 in der x und y Dimension sowie bei 64 in der z Dimension. Die maximale Anzahl kann mit dem Befehl *void glGetIntergeri_v(GL_MAX_COMPUTE_WORK_GROUP_ SIZE, GLuint index, GLint* params* abgefragt werden. Das Produkt der drei Dimensionen darf bei mindestens 1024 liegen. Die maximale Größe kann mit dem Befehl *void glGetIntergeri_v(GL_MAX_COMPUTE_WORK_GROUP_ INVOCATIONS, GLuint index, GLint* params* abgefragt werden.
Einem Shader stehen bei seiner Ausführung mehrere Eingabevariablen zur

Verfügung. Die Variable *uvec3 gl_WorkGroupSize* beinhaltet die Dimensionen der lokalen Work Group, die Variable *uvec3 gl_LocalInvocationID* fungiert als Index der jeweiligen Ausführung in den drei Dimensionen.
Analog dazu können mit der Variable *gl_NumWorkGroups* die Dimensionen der globalen Work Group und mit der Variable *uvec3 gl_WorkGroupID* der Index in den drei Dimensionen abgefragt werden.

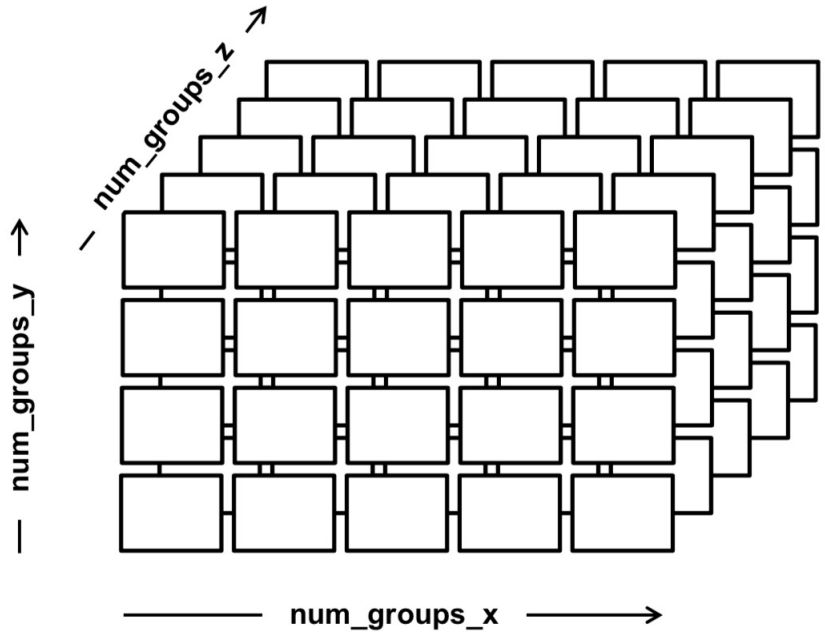


Abbildung 6: Visualisierung der Dimensionen der Globalen Work Groups[Bai]

3.2.3 Buffer Object

Buffer Objects sind OpenGL Objekte, die zur Speicherung von Daten im Arbeitsspeicher der GPU genutzt werden. Ein Compute Shader kann direkt auf Buffer Objects lesen und schreiben.
Ein Uniform Buffer Object (UBO) ist mindestens 16KB groß. Die maximale Größe eines UBO kann mit dem Befehl
*glGetIntegerv(GL_MAX_SHADER_STORAGE_BLOCK_SIZE, GLint * data)* abgefragt werden, sie liegt in der Regel bei 64KB.
Ein Shader Storage Buffer Object (SSBO) ist mindestens 16MB groß. Die Maximale Größe eines SSBO kann mit dem Befehl
GetInteger64v(MAX_SHADER_STORAGE_BLOCK_SIZE) abgefragt werden. Da bei der Erstellung eines SSBO keine Größe angegeben werden muss, kann es zur Speicherung eines Arrays beliebiger Länge genutzt werden. Die Größe des

übergebenen Arrays kann dann im Shader mit der *length*-Funktion abgefragt werden. Werden in einem SSBO mehrere Arrays übergeben, so kann nur das letzte eine variable Länge besitzen.
Da die in dieser Arbeit implementierten Algorithmen für die Nutzung mit großen Datenmengen ausgelegt sind, wäre eine Nutzung von UBOs impraktikabel gewesen. Der Ablauf, ein SSBO zu erstellen, diesen mit Daten zu füllen und die Daten auszulesen, ist bei UBOs und SSBOs ähnlich.
Ein SSBO wird mit dem Befehl *void glBindBuffer(GL_SHADER_STORAGE_BUFFER, GLuint buffer)* gebunden. Anschließend kann mit dem Befehl *void glBufferData(GL_SHADER_STORAGE_BUFFER, GLsizeiptr size, const GLvoid * data, GLenum usage)* der Datenspeicher des SSBOs initialisiert werden. Der Parameter *usage* gibt den späteren Verwendungszweck des Buffers an, er wird intern zur Verbesserung der Performance verwendet. Zuletzt kann mit dem Befehl *void glBindBufferBase(GL_SHADER_STORAGE_BUFFER, GLuint index, GLuint buffer)* der erstellte Buffer an einen bestimmten Index gebunden werden. Ein Anwendungsbeispiel ist in Quelltext 2 auf Seite 15 zu sehen.

```
1  // SSBO struct
2  struct ssbo_1_t
3  {
4          float data[];
5  } ssbo_1;
6
7  for (int i = 0; i < elements; i++)
8          ssbo_1.data[i] = 0.0; // Initialize with zeroes
9
10 glBindBuffer(GL_SHADER_STORAGE_BUFFER, data_buffer[1]);
11 glBufferData(GL_SHADER_STORAGE_BUFFER, elements*sizeof(float), &ssbo_1, GL_DYNAMIC_COPY);
12 glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, data_buffer[1]);
```

Quelltext 2: Erstellung eines Shader Storage Buffer Objects (SSBO)

3.2.4 Shared Memory

Einen im Vergleich zu Buffer Objects signifikant schnelleren Datenzugriff bietet der Shared Memory. Auf Variablen, die im Shared Memory abgelegt werden, kann nur in allen Ausführungen der lokalen Work Group zugegriffen werden.
Die Spezifikation garantiert mindestens 32 Kilobyte an Shared Memory, die maximale Größe kann mit dem Befehl *void glGetIntegerv(GL_MAX_COMPUTE_SHARED_MEMORY_SIZE, GLint * data)* abgefragt werden.

Durch die geringe Größe des Shared Memorys ist seine Nutzung nur begrenzt möglich. In den Implementierungen wurde daher auf die Nutzung von Shared Memory verzichtet, alle Daten sind in SSBOs abgelegt.

3.2.5 Synchronisierung

Zur Synchronisierung der einzelnen Instanzen der Shader können Programmflussbarrieren verwendet werden. Wird in einer Shaderausführung die Funktion *void barrier()* aufgerufen, wird die Ausführung pausiert, bis alle Ausführungen in der lokalen Work Group diesen Punkt erreicht haben.

Weiterhin steht Shadern die Funktion *void memoryBarrierShared()* zur Verfügung. Wird die Funktion aufgerufen, wird die Ausführung pausiert, bis alle Speicherzugriffe abgeschlossen sind.

3.3 Auswahl von Standardalgorithmen zur parallelen Programmierung

Es wurden zwei Standardalgorithmen ausgewählt, die exemplarisch für viele weitere Anwendungsfälle stehen. Zum einen wurde Prefix Sum implementiert, ein Standardalgorithmus, der vielfältige Anwendungsmöglichkeiten bietet und vielfach Grundlage anderer Algorithmen ist. Als zweiten Standardalgorithmus wurde Radixsort implementiert, der Prefix Sum in mehreren seiner Berechungsschritte nutzt, um den Zielindex zu errechnen.

3.3.1 Prefix Sum

Prefix Sum ist ein Algorithmus, der aus einer Folge von Zahlen eine neue Folge von Zahlen berechnet, in der jedes Element die Summe aller bisherigen Elemente ist. Diese Definition als Summe geschrieben ist in Gleichung (2) zu sehen [LF80; SWH15].

$$y_i = \sum_{k=1}^{i} x_i \tag{2}$$

Es gibt zwei Arten Prefix Sum zu berechnen, die gerade gezeigte inklusive und eine exklusive. Bei der inklusiven Variante wird das aktuelle Element in die Berechnung mit einbezogen, bei der exklusiven nicht. Eine Implementierung der inklusiven Berechnung von Prefix Sum wird in Quelltext 3 auf Seite 17 gezeigt, eine Implementierung der exklusiven Berechnung in Quelltext 4 auf Seite 17. Ein Minimalbeispiel wird in Tabelle 1 auf der nächsten Seite gezeigt.

Genutzt wird der Algorithmus in den verschiedensten Anwendungen, beispielsweise beim lexikalischen Vergleichen einer Folge von Buchstaben, bei der Umsetzung von regulären Ausdrücken in Computerprogrammen wie $grep$ und bei der Berechnung der Tiefe aller Knoten eines Baums. Auch mehrere Sortieralgorithmen

Eingabe	1	2	3	4	5
inklusive Prefix Sum	1	3	6	10	15
exklusive Prefix Sum	0	1	3	6	10

Tabelle 1: Inklusive und Exklusive Berechnung von Prefix Sum

```
1 float f = 0.0;
2 int in_array[5] = {1, 2, 3, 4, 5};
3 int out_array[5];
4 for (int i = 0; i < (sizeof(array)/sizeof(array[0])); i++) {
5     f += in_array[i];
6     out_array[i] = f;
7 }
```

Quelltext 3: Implementierung des inklusiven Prefix Sum Algorithmus

```
1 float f = 0.0;
2 int in_array[5] = {1, 2, 3, 4, 5};
3 int out_array[5];
4 for (int i = 0; i < (sizeof(array)/sizeof(array[0])); i++) {
5     out_array[i] = f;
6     f += in_array[i];
7 }
```

Quelltext 4: Implementierung des exklusiven Prefix Sum Algorithmus

nutzen Prefix Sum, beispielsweise Quicksort und der auch in der Arbeit behandelte Radixsort [Ble90].

3.3.2 Radixsort

Radixsort ist ein Sortierverfahren, das anders als viele gängige Sortierverfahren nicht durch Vergleichen einzelner Elemente arbeitet, sondern durch die Verarbeitung der einzelnen Ziffern. Die Anzahl an Ziffern, die ein Zahlensystem zur Nummerndarstellung verwendet, wird Basis oder auch Radix genannt. In dieser Arbeit wurde das Binärsystem genutzt, damit ist der Radix 2 [Knu98].

Der Algorithmus lässt sich in fünf Schritte unterteilen, die für jedes Bit der Binärdarstellung der Zahlen durchlaufen werden. Er geht dabei von rechts nach links vor. In einem Minimalbeispiel soll die Funktion des Algorithmus für die Zahlenfolge 7, 7, 1, 1, 2 gezeigt werden.

Index der Zahlenfolge	0	1	2	3	4
Dezimal	7	7	1	1	2
Binärdarstellung	0111	0111	0001	0001	0010
Bit 0	1	1	1	1	0

Im ersten Schritt wird für das Bit 0 das Histogramm des Auftretens jeder Ziffer gebildet. Von diesem Histogramm wird anschließend die Prefix Sum berechnet.

	Null-Bits	Eins-Bits
Histogramm	1	5
Prefix Sum	0	1

In einem weiteren Schritt werden die Prefix Sums der Eins- und Null-Bits getrennt berechnet, um die Position der Elemente untereinander zu bestimmen.

Bit 0	1	1	1	1	0
Prefix Sum der Null-Bits	/	/	/	/	0
Prefix Sum der Eins-Bits	0	1	2	3	/

Nun werden die in den beiden vorherigen Schritten berechneten Prefix Sums addiert, um den Zielindex zu berechnen, an welchem die Zahl verschoben werden muss.

Index	0	1	2	3	4
Dezimal	7	7	1	1	2
Binär	0111	0111	0001	0001	0010
Bit 0	1	1	1	1	0
Prefix Sum des Histogramms für dieses Bit (Schritt 2)	1	1	1	1	0
Prefix Sum desjeweiligen Bits (Schritt 3)	0	1	2	3	0
Zielindex=Summe	1	2	3	4	0

Dies führt nach dem ersten Durchlauf zu folgender Sortierung:

Index	0	1	2	3	4
Dezimal	2	7	7	1	1
Binär	0010	0111	0111	0001	0001

Die selben Schritte werden für jedes weitere Bit wiederholt. Der Algorithmus benötigt daher so viele Durchläufe wie die Zahl Ziffern besitzt. In diesem Fall ist die finale Sortierung nach vier Durchläufen erreicht.

Index	0	1	2	3	4
Dezimal	1	1	2	7	7
Binär	0001	0001	0010	0111	0111

3.4 Implementierung von Prefix Sum

Soll Prefix Sum parallel implementiert werden, muss der Algorithmus in Teilschritte unterteilt werden, die dann einzen parallelisiert ausgeführt werden. Hierfür gibt es zwei Ansätze: Der eine erfordert zwei getrennte Durchläufe von links nach rechts und anschließend von rechts nach links. Der andere führt nur einen Durchlauf von links nach rechts aus.

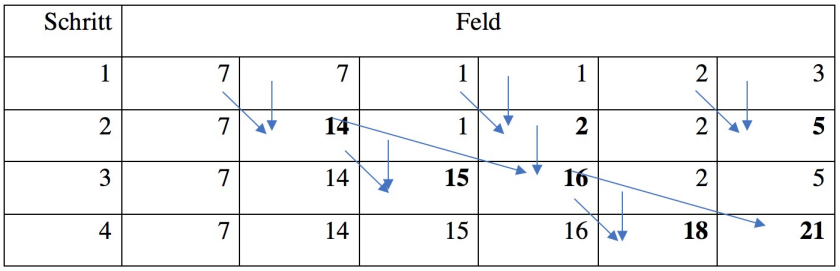


Abbildung 7: Parallele Ausführung von Prefix Sum

Der zweite Ansatz ist in Abbildung 7 dargestellt und wurde in dieser Arbeit genutzt. Ist ein vierelementiges Eingabearray mit den Werten x_n und ein Ausgabearray mit den Werten y_n gegeben, war die bisherige nicht-parallele Berechnungsweise die folgende:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 + x_2 \\ y_3 &= x_0 + x_1 + x_2 + x_3 \end{aligned}$$

Die parallele Variante unterteilt die Berechnung nun in mehrere Schritte. Im ersten Schritt werden die Ergebnisse wie folgt berechnet:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 \\ y_3 &= x_2 + x_3 \end{aligned}$$

Im zweiten Schritt werden die noch fehlenden Anteile hinzuaddiert:

$$\begin{aligned} y_2 &= y_2 + y_1 \\ y_3 &= y_3 + y_1 \end{aligned}$$

Zur Implementierung dieses Ansatzes wurde ein Compute Shader programmiert, der in drei SSBOs ein Eingabearray, ein Ausgabearray und ein Array für die temporäre Speicherung von Zwischenergebnissen nutzen kann. Die Größe der lokalen Work Group liegt in der y und z Dimension jeweils bei 1. In der X-Dimension ist die Dimension abhängig von der Größe des Eingabearrays festgelegt.

Jeder Shader ist für die Berechnung eines Vielfachen von zwei Elementen zuständig. Dieser Faktor wird in der Variable *factor* angegeben und muss bereits zum Zeitpunkt der Kompilierung feststehen. Hierbei gilt $elemente = factor * gl_WorkGroupSize.x$. Der für diese Eingabedaten zuständige Abschnitt des Compute Shaders ist in Quelltext 5 auf Seite 20 sichtbar. Die drei SSBOs sind an den Binding Points 0 bis 2 gebunden und nutzen alle den Memory Layout Qualifier $std430$.

```
3  layout (local_size_x = 1024) in;
4  const int factor = 2;
5
6  layout (std430, binding = 0 ) buffer shader_dataSSBO
7  {
8          float in_array[gl_WorkGroupSize.x * factor];
9  };
10
11 layout (std430, binding = 1 ) buffer shader_dataSSBO_2
12 {
13         float out_array[gl_WorkGroupSize.x * factor];
14 };
15
16 layout (std430, binding = 2 ) buffer shader_dataSSBO_3
17 {
18         float tmp[gl_WorkGroupSize.x * factor];
19 };
```

Quelltext 5: Eingabedaten des Compute Shaders zur parallelen Berechnung von Prefix Sum

Um den Shader in der Laufzeitmessung mit verschiedenen Eingabegrößen zu nutzen, wurden dynamisch eine vielzahl an Compute Shadern erzeugt. Zum einen Compute Shader mit einem festen Faktor 2 und einer in 16er-Schritten größer werdenden X- Dimension für die Berechnung der Prefix Sum von 32 bis 2048 Elementen. Mit einer X-Dimension von 1024 war die maximale Anzahl an lokalen Work Groups erreicht, die genutzt werden konnte. Zum anderen wurden Compute Shader erzeugt, die ab der X-Dimension von 1024 Punkten einen veränderlichen Faktor besitzen. Der Faktor wurde in Zweierschritten bis zu einem Wert von 500 erhöht. Ein Ansatz, die X-Dimension dynamisch zu übergeben, wäre die Nutzung der OpenGL-Erweiterung *ARB_compute_variable_group_size*. Bei dieser werden Dimensionen der lokalen Work Group erst beim Aufruf des Shaders übergeben. Da die Größe des Shaders in dieser Implementierung jedoch zum Zeitpunkt der Kompilierung vorliegen muss, konnte diese Erweiterung nicht genutzt werden.

Die Berechnung der Prefix Sum findet in einem temporären Array statt. Vor und

nach der Berechnung müssen daher die Daten aus dem Eingabearray in das temporäre Array bzw. aus dem temporären Array in das Ausgabearray kopiert werden. Im Anschluss wird durch die Befehle *barrier()* und *memoryBarrierShared()* sichergestellt, dass alle Instanzen des Compute Shaders den Vorgang abgeschlossen haben. Dieser Vorgang ist in Quelltext 6 auf Seite 21 dargestellt.

```
28      for (int offset = 0; offset < factor; offset++) {
29              tmp[id * factor + offset] = in_array[id *
                 ↪ factor + offset];
30      }
31
32      barrier();
33      memoryBarrierShared();
```

Quelltext 6: Kopieren des Eingabearrays in das temporäre Array

Ausgehend vom Ansatz der Implementierung wird die Anzahl der Durchläufe berechnet, die jede Instanz ausführen muss. Sie liegt bei $log2(gl_WorkGroupSize.x * factor) + 1$. Für die Anzahl der Elemente, für die eine Instanz zuständig ist, berechnet sie nun die Lese- und Schreibposition abhängig von ihrer ID und der Nummer ihrer Ausführug. Der berechneten Schreibposition wird nun der Wert der berechneten Leseposition hinzuaddiert. Nach jedem Durchlauf wird durch die Nutzung von *barrier()* und *memoryBarrierShared()* sichergestellt, dass alle Instanzen des Compute Shaders den Vorgang abgeschlossen haben. Quelltext 7 auf Seite 22 zeigt die Umsetzung dieses Verfahrens im Compute Shader.

Um die Ausführungszeiten der Implementierungen vergleichen zu können, wurde zusätzlich eine nicht-parallele Variante von Prefix Sum, die in Quelltext 3 auf Seite 17 gezeigt wurde, zur Ausführung auf der GPU implementiert. Hierzu wird eine globale Work Group mit den Dimensionen $1, 1, 1$ und eine lokale Work Group mit den Dimensionen $1, 1, 1$ verwendet. Zu beachten ist bei dieser Vorgehensweise, dass ein Großteil der verfügbaren Rechenleistung der GPU ungenutzt bleibt, da diese auf die Ausführung von parallelen Programmen ausgelegt ist.

3.5 Implementierung von Radixsort

Der Ablauf der Ausführung der Implementierung von Radixsort ist in Quelltext 8 auf Seite 23 zu sehen. Dabei durchläuft die Ausführung jede Ziffer der Binärdarstellung von rechts nach links. Eine Parallelisierung dieses Vorgangs ist nicht möglich, da jeweils das Ergebnis des Durchlaufs für die vorhergehende Ziffer benötigt wird. Stattdessen wir die Parallelität durch die Parallelisierung einzelner Teile des Algorithmus erreicht. Zwischen den einzelnen Teilen der Ausführung wird durch die Nutzung von *barrier()* und *memoryBarrierShared()* sichergestellt, dass alle Instanzen des Compute Shaders den Vorgang abgeschlossen haben.

```
35      uint steps = uint(log2(gl_WorkGroupSize.x * factor)) + 1;
36      uint step;
37      uint offset = 0;
38      uint factor_loop = factor/2;
39      for (step = 0; step < steps; step++)
40      {
41              for (; factor_loop > 0; factor_loop--, offset +=
                 ↪ gl_WorkGroupSize.x) {
42                      mask = (1 << step) - 1;
43                      rd_id = (((id + offset)  >> step) <<
                         ↪ (step + 1)) + mask;
44                      wr_id = rd_id + 1 + ((id + offset)  &
                         ↪ mask);
45
46                      tmp[wr_id] += tmp[rd_id];
47              }
48              offset = 0;
49              factor_loop = factor/2;
50
51              barrier();
52              memoryBarrierShared();
53      }
```

Quelltext 7: Berechnung der einzelnen Compute Shader zur parallelen Berechnung von Prefix Sum

Der Compute Shader nutzt drei SSBOs im Speicher, darunter eins mit dem Eingabearray, eins mit dem Ausgabearray und eins mit Arrays für die temporäre Speicherung von Zwischenergebnissen.

Im ersten Schritt der wird das Histogramm an jeder Ziffer des Eingabearrays berechnet. Dieser Vorgang muss nur einmalig ausgeführt werden, da sich das Histogramm auch durch die Änderung der Reihenfolge der Elemente nicht ändert. Die parallele Implementierung der Berechnung des Histogramms ist in Quelltext 9 auf Seite 24 zu sehen. Sie nutzt nur 32 der Instanzen, für jede Ziffer eine.

Nun durchläuft jede Instanz für jede Ziffer der Zahlen die selben Schritte. Die Funktion $init()$ initialisiert die Arrays der Null- und Eins-Bits, damit auf diesen die Prefix Sum berechnet werden kann. Die verwendete Funktion $prefixSum()$ ist die selbe, die auch in Abschnitt 3.4 auf Seite 19 vorgestellt wurde. Ist die Prefix Sum berechnet, kann die Funktion $move()$ die Zahlen an ihre neue Position verschieben. Abschließend werden die Ergebnisse der einzelnen Instanzen mit der Funktion $sync()$ synchronisiert.

Der Vorgang des Verschiebens der Elemente in der Funktion $move()$ ist in Quell-

```
1   histogramLSB(); // Berechne das Histogramm für alle Ziffern
2
3   barrier();
4   memoryBarrierShared();
5
6         for(uint digit = 0; digit < 32; digit++) {
7           init();
8
9           barrier();
10          memoryBarrierShared();
11
12          prefixSum();
13
14          barrier();
15          memoryBarrierShared();
16
17          move(digit);
18
19          barrier();
20          memoryBarrierShared();
21
22          sync();
23  }
```

Quelltext 8: Aufrufe des Compute Shaders zur Ausführung von Radixsort

text 10 auf Seite 24 dargestellt. Jede Instanz ist für ein Vielfaches von zwei Elementen zuständig. Dieser Faktor wird genauso wie in der Implementierung von Prefix Sum in der Variable *factor* angegeben und muss bereits zum Zeitpunkt der Kompilierung feststehen. Hierbei gilt ebenso $elemente = factor * gl\_WorkGroupSize.x$. Der Zielindex wird, wenn das aktuell betrachtete Bit gleich 0 ist, aus dem Ergebnis der Prefix Sum Berechnung für dieses Bit berechnet. Ist das aktuell betrachtete Bit 1, so wird zu dem Ergebnis der Prefix Sum Berechnung das Ergebnis des Histogramms der Null-Bits an dieser Position addiert.

```
void histogramLSB() {
  uint id = gl_LocalInvocationID.x;
  uint digit = gl_LocalInvocationID.x;

  if (digit < 32) {
        for(int i=0; i < in_array.length(); i++) {
          uint potenz = 1 << digit;
                uint bit = (in_array[i] & potenz) >> digit;
                if (bit == 0) {
                  histo_zero[digit]++;
                }
          }
        }
}
```

Quelltext 9: Berechnung des Histogramms

```
void move(uint id, uint aktuelleZiffer) {
        uint zielindex;
        for (int offset = 0; offset < responsible; offset++)
         ↪ {
          if (current_bit[id * responsible + offset] == 0) {
                zielindex = prefix_sum_exclusive_0[id *
                 ↪ responsible + offset];
          } else {
                zielindex = histo_zero[aktuelleZiffer] +
                 ↪ prefix_sum_exclusive_1[id * responsible +
                 ↪ offset];
          }
          out_array[zielindex] = in_array[id * responsible +
           ↪ offset];
        }
}
```

Quelltext 10: Funktion zur Verschiebung der Elemente an ihren Zielindex

4 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Implementierung präsentiert und verglichen. Zunächst wird erläutert, wie die Ausführungszeit der einzelnen Algorithmen berechnet wurde. Im Anschluss werden die gewonnen Daten in Abbildungen präsentiert und diskutiert. Abschließend werden die aus der Implementierung und der Laufzeitmessung gewonnen Erkenntnisse angesprochen.

4.1 Laufzeiten

Die in Abschnitt 3 auf Seite 11 gezeigten Implementierungen wurden auf einer NVIDIA GeForce GTX 560, die auf der in Abschnitt 2.1.1 auf Seite 5 erläuterten Fermi-Architektur basiert, ausgeführt. Sie besitzt eine GPU des Typs GF114 und hat 1024 MB GDDR5 Grafikspeicher. Das Speicherinterface ist mit 256 Bit angebunden, die theoretische Rechenleistung bei Gleitkommazahlen mit einfacher Genauigkeit liegt bei 1088 Gflops, bei doppelter Genauigkeit bei 90 Gflops. Es werden Befehle bis zu der aktuellen OpenGL Version 4.6 unterstützt [Micb; Mica].

```
1  GLuint64 start, stop;
2  unsigned int id[2];
3
4  glGenQueries(2, id);
5  glQueryCounter(id[0], GL_TIMESTAMP);
6
7  ComputShader->useProgram();
8  glDispatchCompute(1, 1, 1);
9  glMemoryBarrier(GL_ALL_BARRIER_BITS);
10
11 glQueryCounter(id[1], GL_TIMESTAMP);
12
13 GLint resultAvailable = 0;
14 while (!resultAvailable) {
15         glGetQueryObjectiv(id[1], GL_QUERY_RESULT_AVAILABLE,&resultAvailable);
16 }
17
18 glGetQueryObjectui64v(id[0], GL_QUERY_RESULT, &start);
19 glGetQueryObjectui64v(id[1], GL_QUERY_RESULT, &stop);
20
21 printf("Laufzeit: %f ms", (stop - start) / 1000000.0);
```

Quelltext 11: Berechnung der Ausführungszeit auf der GPU

Zur Messung der Laufzeiten der Compute Programs wurden OpenGL Timer Queries genutzt [Groe]. Mit dem Befehl *void glGenQueries(GLsizei n, GLuint * ids)* werden

zwei Queries erstellt, eine zum Aufzeichnen der Zeit vor, die andere zum Aufzeichnen der Zeit nach der Ausführung der zu messenden Abläufe. Anschließend kann mit *glQueryCounter(GLuint id, GL_TIMESTAMP)* die exakte Zeit gespeichert werden. Nun werden die zu messenden Befehle ausgeführt, beispielsweise ein Compute Program. Mit einem weiteren *glQueryCounter(GLuint id, GL_TIMESTAMP)* wird nun die exakte Zeit nach Ende der Ausführung aller vorherigen Befehle gespeichert. Da das Ergebnis erst nach Abschluss aller Berechnungen verfügbar ist, wird der Befehl *void glGetQueryObjectiv(GLuint id, GL_QUERY_RESULT_AVAILABLE, GLint * params)* in einer Schleife ausgeführt, bis das Ergebnis verfügbar ist. Die so gespeicherten Zeitmessungen können in einem letzten Schritt mit *void glGetQueryObjectiv(GLuint id, GL_QUERY_RESULT, GLint * params)* ausgelesen und voneinander subtrahiert werden.
Zur Umwandlung von Nanosekunden in Millisekunden wird das Ergebnis durch 1000000 geteilt. Ein Anwendungsbeispiel ist in Quelltext 11 auf Seite 25 zu sehen.

4.1.1 Prefix Sum

Abbildung 8 auf der nächsten Seite zeigt die Ausführungszeit in Millisekunden (ms) der parallelen Implementierung von Prefix Sum auf der GPU mit zufälligen Eingabewerten. In Schritten von 1024 Elementen wurde die Prefix Sum von 1024 bis zu 403456 Elementen berechnet. Als Eingabe wurden zufällige Fließkommazahlen verwendet. Der Algorithmus benötigt dabei zur Berechnung eine mit der Anzahl der Elemente wachsende Zeit. Abbildung 9 auf Seite 28 und Abbildung 10 auf Seite 28 vergleichen die Ausführungszeit mit der einer nicht-parallelen Implementierung des Algorithmus. Bei der ersten Abbildung wurde die Prefix Sum ebenso von 403456 Elementen berechnet, bei der zweiten Abbildung von bis zu einer Million Elementen.

Der Geschwindigkeitsvorteil der parallelen Implementierung ist in den beiden vergleichenden Abbildungen deutlich erkennbar. Da die Zeitmessung sehr präzise ist, ist es unwarscheinlich, dass die in allen drei Abbildungen zu sehende Variation durch eine Messungenauigkeit verursacht wurde. Eine wahrscheinlichere Erklärung könnte in anderen Aufgaben liegen, die die GPU zur selben Zeit ausführen musste, beispielsweise durch das Betriebssystem verursachte. Durch eine mehrfache Messung und Aggregierung der Ergebnisse könnte diesem Problem in der Zukunft entgegengewirkt werden.

4.1.2 Radixsort

Abbildung 11 auf Seite 29 zeigt die Ausführungszeit in Millisekunden (ms) der parallelen Implementierung von Radixort auf der GPU mit zufälligen Eingabewerten. Es wurde in Schritten von 8 Elementen 8 bis 1024 Elemente sortiert. Der Algorithmus benötigt eine mit der Anzahl der Elemente wachsende Zeit. Abbildung 12 auf Seite 29 zeigt die Ausführungszeit für die Sortierung von 1024 bis 55296 Elementen in Schritten von 1024 Elementen.

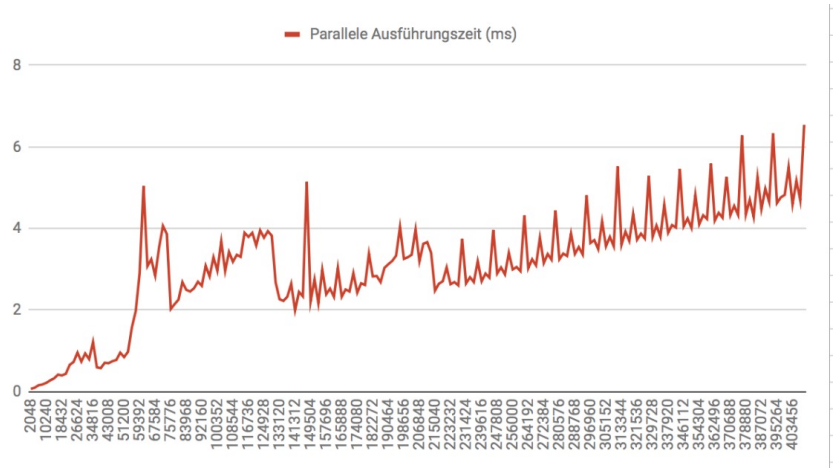


Abbildung 8: Ausführungszeit der parallelen Implementierung von Prefix Sum

4.2 Erkentnisse

Die Implementierung der beiden Algorithmen hat zu mehreren Erkenntnissen geführt. Zum einen kann eine merkliche Verbesserung der Ausführungsgeschwindigkeit festgestellt werden. Sie ist bei beiden Implementierungen messbar und wurde in Abbildungen dargestellt. Vor allem bei Prefix Sum ist der Geschwindigkeitsvorteil signifikant.

Bei der Implementierung von Radixsort hat sich gezeigt, dass sich nicht immer der gesamte Algorithmus parallelisieren lässt. Stattdessen müssen einzelne Teile parallelisiert werden. Dieser Nachteil kann ebenso als Vorteil angesehen werden, denn durch Optimierung der einzelnen Teilschritte lässt sich möglicherweise ein größerer Geschwindigkeitsvorteil erzielen als durch die Optimierung eines ganzen Algorithmus.

Auch lässt sich nicht jeder Standardalgorithmus, der in der Datenverarbeitung eingesetzt wird, parallelisieren, da manche Vorgänge nur sequentiell abgearbeitet werden können und zwingend die Ergebnisse von Zwischenschritten benötigen. Hier wäre ein möglicher Ansatz, den gesamten Prozess, den dieser Algorithmus abbildet, zu überdenken und anzupassen.

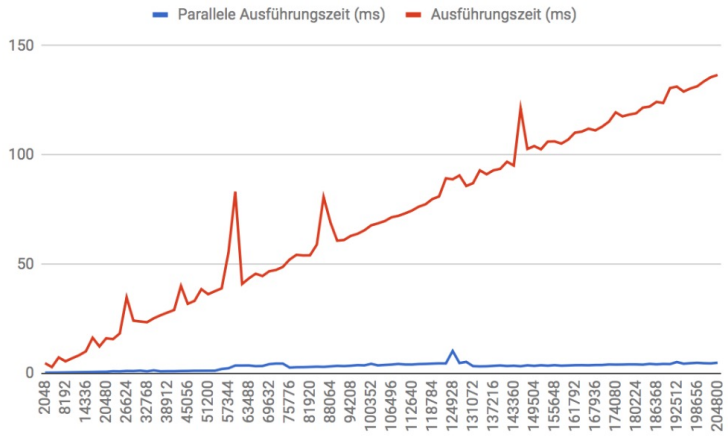


Abbildung 9: Ausführungszeit der nicht-parallelen und der parallelen Prefix Sum Implementierung mit 2048 bis 204800 Elementen

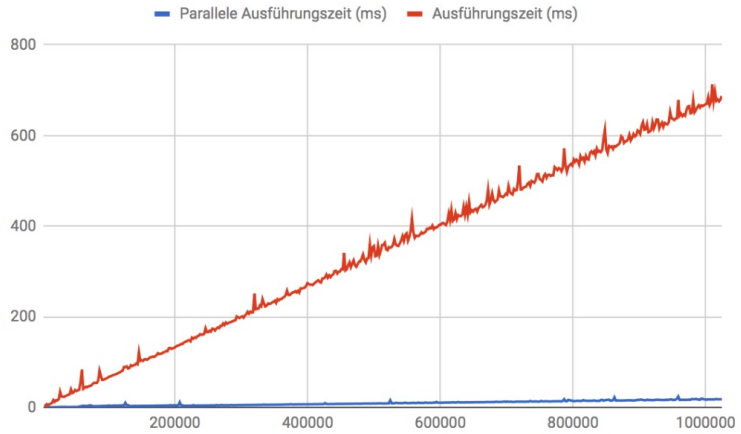


Abbildung 10: Ausführungszeit der nicht-parallelen und der parallelen Prefix Sum Implementierung mit 2048 bis 1024000 Elementen

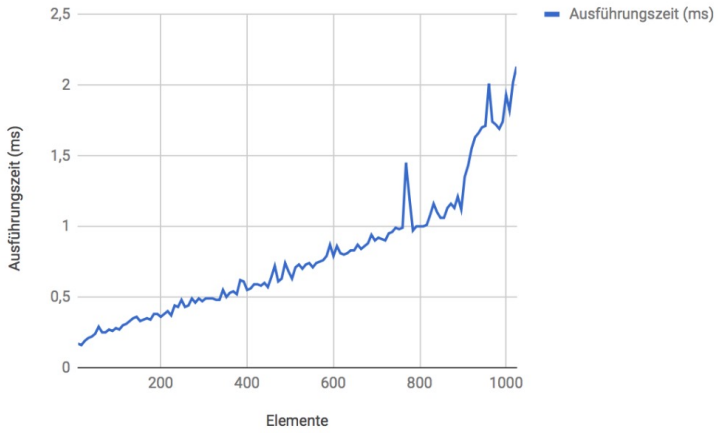


Abbildung 11: Ausführungszeit der Implementierung von Radixsort

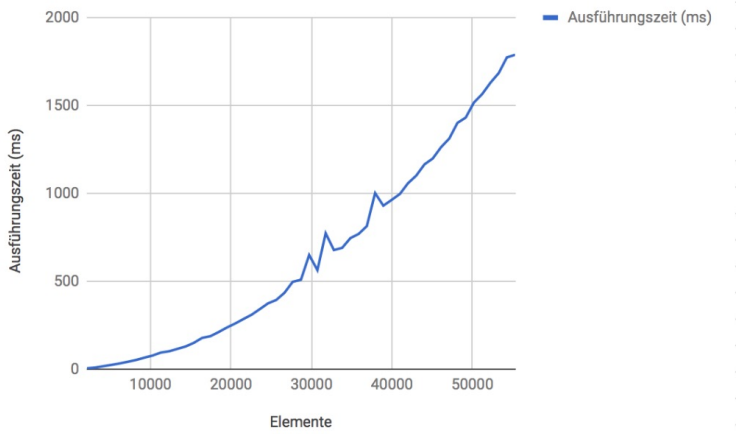


Abbildung 12: Ausführungszeit der Implementierung von Radixsort

5 Zusammenfassung

In diesem Abschnitt wird aus der Umsetzung dieser Arbeit gezogenen Erkenntnisse ein Fazit gezogen. Dabei wird noch einmal auf die Implementierungen und die Ergebnisse eingegangen. Anschließend wird auf den weiteren Forschungsbedarf im Themenfeld der parallelen Umsetzung von Algorithmen eingegangen und ein Ausblick auf die zukünftige Entwicklung gegeben.

5.1 Fazit

Diese Arbeit hat einen Überblick über die Möglichkeiten zur Programmierung von Graphics Processing Units (GPUs) im Rahmen von General Purpose Computation on Graphics Processing Unit (GPGPU)-Anwendungen gegeben. Dabei wurden die gängigsten Application Programming Interfaces (APIs) zur Programmierung, Open Graphics Library (OpenGL), Open Computing Language (OpenCL), Compute Unified Device Architecture (CUDA), DirectX und Vulkan, vorgestellt. Nach der Entscheidung für die Verwendung von OpenGL wurden die von dieser API bereitgestellten Compute Shader vorgestellt und beschrieben. Dabei wurde insbesondere auf die zu beachtenden Besonderheiten eingegangen.

Weiterhin wurden zwei Standardalgorithmen, die in der Datenverarbeitung Verwendung finden, ausgewählt und beschrieben. Hierbei wurden die Unterschiede zwischen einer nicht-parallelen und einer parallelen Implementierung dieser Algorithmen aufgezeigt.

Die Implementierung wurde mit den beschriebenen OpenGL Compute Shadern vorgenommen. Beide Implementierungen sind mit verschiedenen Größen an Eingabedaten lauffähig und zeigen dabei im Vergleich mit nicht-parallelen Implementierungen signifikante Verbesserung in der Ausführungszeit, die durch die massive Parallelisierung der einzelnen Teilschritte der Algorithmen erreicht wurde. Dies verdeutlicht die Möglichkeiten des GPGPU Programmieransatzes für die Parallelisierung von anderen Standardalgorithmen, um deren Laufzeit zu erheblich zu verkürzen.

5.2 Weiterer Forschungsbedarf

Das Forschungsfeld eröffnet viele weitere Möglichkeiten für eine Auseinandersetzung mit verwandten Themen. In zukünftigen Arbeiten könnten beispielsweise andere häufig genutzte Standardalgorithmen aus dem Bereich der Datenverarbeitung auf ihre Implementierbarkeit und Parallelisierbarkeit untersucht werden. Ein weiteres interessantes Feld, welches in dieser Arbeit nicht betrachtet wurde, ist das parallele Suchen in sortierten Datenmengen. Dies könnte auch unter der Nutzung des gezeigten Radixsort Algorithmus implementiert werden.

Auch die bestehenden Implementierungen könnten überarbeitet werden. So sind die von den Compute Shadern zum Datenzugriff genutzten Shader Storage Buffer Objects (SSBOs), die auf dem Arbeitsspeicher der Grafikkarte abgelegt werden,

signifikant langsamer als der interne Shared Memory. Die Algorithmen könnten so modifiziert werden, dass die Ergebnisse bestimmter Teilrechenschritte auf dem Shared Memory abgelegt werden. Der entstehende Geschwindigkeitsgewinn könnte hier gemessen werden.

Eine weitere Möglichkeit wäre, eine andere API, als die in dieser Arbeit verwendete OpenGL-API, zur Implementierung der Algorithmen zu nutzen. Hier könnten zukünftige Arbeiten ansetzen und die selben Algorithmen über verschiedene APIs hinweg vergleichen.

In beiden Implementierungen wurde nur die X-Dimension der lokalen Work Group genutzt, außerdem wurden alle Implementierungen mit einer globalen Work Group Dimension von $1, 1, 1$ ausgeführt. Stattdessen war eine Instanz immer für eine feste Anzahl an Elementen zuständig. Hier könnte der Performanceunterschied zu einer Implementierung unter der Nutzung vieler globaler Work Groups gemessen werden.

5.3 Ausblick

Die Arbeit hat gezeigt, dass die Verwendung paralleler Implementierungen von Algorithmen große Vorteile in der Ausführungsgeschwindigkeit bietet. In der Zukunft wird der Fokus vermehrt darauf liegen, bestehende Algorithmen zu parallelisieren oder neue Algorithmen für bestehende Probleme zu entwickeln, die es ermöglichen diese in paralleler Art und Weise lösen.
Die Hersteller von GPUs haben in den vergangenen Jahren das Potential für Anwendungsfälle, die unter GPGPU fallen, nicht übersehen und werden so zukünftig ein Interesse daran haben, die Nutzung ihrer GPUs möglichst entwicklerfreundlich zu gestalten. Hier werden sich die angebotenen APIs weiter entwickeln, ein erster Schritt in diese Richtung stellt die noch sehr junge Vulkan-API dar.

Abbildungsverzeichnis

Quelltext

Akronyme

API Application Programming Interface. v, 2, 5, 7–11, 31, 32

CPU Central Processing Unit. 3, 5, 9

CUDA Compute Unified Device Architecture. 5, 9, 11, 31

FPGA Field Programmable Gate Array. 9

GLSL OpenGL Shading Language. 2, 8, 11

GPGPU General Purpose Computation on Graphics Processing Unit. 5, 9, 11, 31, 32

GPU Graphics Processing Unit. 1–3, 5–11, 14, 21, 25, 26, 31, 32

IRIS GL Integrated Raster Imaging System Graphics Library. 8

OpenCL Open Computing Language. 9–11, 31

OpenGL Open Graphics Library. 2, 8–12, 14, 20, 31, 32

SGI Silicon Graphics, Inc.. 8

SM Streaming Multiprocessor. 5, 7

SPIR Standard Portable Intermediate Representation. 9, 10

SSBO Shader Storage Buffer Object. 14–16, 19, 20, 22, 31

UBO Uniform Buffer Object. 14, 15

Literatur

[LF80] Richard E. Ladner und Michael J. Fischer. „Parallel Prefix Computation“. In: *J. ACM* 27.4 (Oktober 1980), S. 831–838. ISSN: 0004-5411. DOI: 10.1145/322217.322232.

[Ble90] Guy E Blelloch. „Prefix sums and their applications“. In: (1990).

[Knu98] Donald E. Knuth. *The art of computer programming*. 2nd. Bd. 3. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1998. ISBN: 0-201-89685-0.

[BM06] D.C. Brock und G.E. Moore. *Understanding Moore's Law: Four Decades of Innovation*. Chemical Heritage Foundation, 2006. ISBN: 9780941901413.

[Itu+11] LM Itu u. a. „Comparison of single and double floating point precision performance for Tesla architecture GPUs“. In: *Bulletin of the Transilvania University of Braşov* 4.53 No. 2 (2011).

[PH11] David A. Patterson und John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 9780123747501.

[SWH15] Graham Sellers, Richard S. Wright und Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference*. 7th. Addison-Wesley Professional, 2015. ISBN: 9780672337475.

[Baj17] Toru Baji. „GPU: the biggest key processor for AI and parallel processing“. In: *Proc.SPIE* 10454 (2017), S. 10454–10454. DOI: 10.1117/12.2279088. URL: http://dx.doi.org/10.1117/12.2279088.

[And] Backblaze Andy Klein. *Hard Drive Cost Per Gigabyte*. URL: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/ (abgerufen am 10. Februar 2018).

[Bai] Mike Bailey. *OpenGL Compute Shaders*. URL: http://web.engr.oregonstate.edu/~mjb/cs557/Handouts/compute.shader.1pp.pdf (abgerufen am 30. Januar 2018).

[Groa] Khronos Group. *History of OpenGL*. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL#Overview (abgerufen am 30. Januar 2018).

[Grob] Khronos Group. *Khronos Releases OpenCL 2.2 With SPIR-V 1.2*. URL: https://www.khronos.org/vulkan/ (abgerufen am 30. Januar 2018).

[Groc] Khronos Group. *OpenCL Overview*. URL: https://www.khronos.org/opencl/ (abgerufen am 30. Januar 2018).

[Grod] Khronos Group. *OpenGL Overview*. URL: https://www.khronos.org/opengl/ (abgerufen am 30. Januar 2018).

[Groe] Khronos Group. *Timer queries - OpenGL Wiki*. URL: https://www.khronos.org/opengl/wiki/Query_Object#Timer_queries (abgerufen am 10. Februar 2018).

[Grof] Khronos Group. *Vulkan Overview*. URL: https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2 (abgerufen am 30. Januar 2018).

[Mica] Microsoft. *NVIDIA GeForce GTX 560*. URL: http://www.nvidia.de/object/product-geforce-gtx-560-de.html (abgerufen am 10. Februar 2018).

[Micb] Microsoft. *NVIDIA GeForce GTX 560 Performance*. URL: https://gfxbench.com/device.jsp?D=NVIDIA+GeForce+GTX+560&os=Windows&api=gl&testgroup=info (abgerufen am 10. Februar 2018).

[NVI] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. URL: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (abgerufen am 10. Februar 2018).

[Rup] Karl Rupp. *CPU, GPU and MIC Hardware Characteristics over Time*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (abgerufen am 30. Januar 2018).

[Sel+] Graham Sellers u. a. *ARB_compute_shader specification*. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_compute_shader.txt (abgerufen am 30. Januar 2018).

[Sta] StatCounter. *Desktop Operating System Market Share Worldwide*. URL: http://gs.statcounter.com/os-market-share/desktop/worldwide (abgerufen am 10. Februar 2018).