Performanceanalyse von SkelCL mittels B+Baum Traversierung und 3D Jacobi Stencil Berechnung

eine
Bachelorarbeit
an der
Westfälischen Wilhelms-Universität Münster
von
Patrick Schiffler

Matrikelnummer: 374414

Betreuer: Michel Steuwer, Prof. Sergei Gorlatch

8. März 2013

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**
2.1. Die Central Processing Unit . . . 3
2.2. Die Graphical Processing Unit . . . 3
2.3. Flynn'sche Klassifikation . . . 4
2.4. OpenCL . . . 4
2.4.1. Das OpenCL Plattformmodell . . . 5
2.4.2. Das OpenCL Ausführungsmodell . . . 5
2.4.3. Das OpenCL Speichermodell . . . 7
2.4.4. Aufbau eines OpenCL Programms . . . 8
2.5. Algorithmische Skelette . . . 9
2.6. Die SkelCL Bibliothek . . . 9
2.7. Die Rodinia Benchmark-Suite . . . 11
2.8. Die Parboil Benchmark-Suite . . . 11

3. 3D Jacobi Stencil Berechnung **13**
3.1. 3D Jacobi Stencil . . . 14
3.2. OpenCL Implementierung . . . 14
3.3. SkelCL Implementierung . . . 16
3.3.1. SkelCL Implementierung mit Vektoren . . . 17
3.3.2. SkelCL Implementierung mit Matrizen . . . 19
3.3.3. SkelCL Implementierung mit MapOverlap . . . 21
3.4. Evaluation . . . 23

4. B+Baum Traversierung **27**
4.1. Implementierung . . . 28
4.2. OpenCL Implementierung . . . 29
4.3. SkelCL Implementierung . . . 30
4.4. Evaluation . . . 33

5. Fazit **37**

Literatur **39**

A. Quelltexte **41**

1. Einleitung

Die Entwicklung von Anwendungen für multi-core CPU und GPU Systeme mittels OpenCL ist sehr komplex und fehleranfällig. Dies liegt daran, dass die Kommunikation sowie die Datenverteilung bei GPUs explizit vom Programmierer erfolgen muss. Daher wird von der Arbeitsgruppe Parallele und Verteilte Systeme an der Universität Münster eine auf OpenCL aufsetzende Bibliothek namens SkelCL entwickelt, die das Erstellen von parallelen Anwendung erleichtern soll. Dafür nutzt SkelCL algorithmische Skelette, die oft auftretende Berechnungs- und Kommunikationsmuster repräsentieren. Außerdem werden dem Benutzer die abstrakten Datentypen Vektor und Matrix angeboten, welche die Verteilung der Daten auf die GPU Speicher vereinfachen. Dadurch wird der Entwickler um diese komplexen Aufgaben entlastet.

Ziel dieser Bachelorarbeit ist ein Vergleich der Performance und des Entwicklungsaufwands von Anwendungen entwickelt mit SkelCL gegenüber den selben Anwendungen, entwickelt mit OpenCL. Da SkelCL eine Bibliothek ist, die auf OpenCL aufsetzt, ist der Performance- und Aufwandsunterschied interessant, um zu zeigen welche Vorteile der Einsatz von SkelCL, als Entwicklungsansatz für Anwendungen, bietet. Es werden zwei Benchmarks aus verschiedenen Anwendungsgebieten ausgewählt.

1. Es wird zum einen die Traversierung von B+Bäumen behandelt. Sie ist in der Benchmark Suite Rodinia als OpenCL Version enthalten. B+Bäume sind spezielle B Bäume, bei denen die Daten nur in den Blättern gehalten werden. Ihr Haupteinsatzgebiet sind Datenbanken, in denen sie als Suchbäume fungieren und Dateisysteme, in denen sie die Dateizuordnungstabellen darstellen. Bei der Traversierung werden die B+Bäume nach ein oder mehreren dieser Daten durchsucht.

2. Zum anderen wird die Jacobi Stencil Berechnung auf einem regulären 3D Gitter umgesetzt. Eine Implementierung für OpenCL ist in der Benchmark Suite Parboil zu finden. Hierbei werden alle Knoten eines 3D Gitters durch eine bestimmte Vorschrift, mit den Werten ihrer Nachbarknoten, ein- oder mehrfach aktualisiert. Zu finden ist diese Berechnung hauptsächlich in der Physik, da sie numerisch die Wärmeleitungsgleichung löst, welche die Wärmeentwicklung in dreidimensionalen Objekten beschreibt.

Diese beiden Aufgaben sollen in SkelCL implementiert werden, um sie mit den Implementierungen in OpenCL vergleichen zu können. Dazu muss ein geeignetes Maß gewählt werden, mit welchem der Entwicklungsaufwand und die Performance gemessen und dargestellt werden können.

Im folgenden ersten Abschnitt werden einige grundlegende Begriffe erklärt und in das Thema eingeführt. In Abschnitt 3 wird die Implementierung der 3D Jacobi Stencil Berechnung, in OpenCL und SkelCL, vorgestellt, wobei die SkelCL Implementierung in mehreren Versionen vorliegt. Außerdem wird auf das Anwendungsgebiet und den Nutzen dieser Berechnung eingegangen. Nach dem Einführen in die Implementierungen, wird

ein Vergleich der Performance und des Programmieraufwands, für die unterschiedlichen Versionen der 3D Jacobi Stencil Berechnung durchgeführt. Hier werden auch die unterschiedlichen Aspekte der Datentypen, die SkelCL zur Verfügung stellt, erläutert. Im folgenden Abschnitt 4 werden dann die Implementierungen der B+Baum Traversierung dargestellt, wobei diese auch in OpenCL und in SkelCL vorliegen. Allerdings werden zunächst das Einsatzgebiet und die grundlegenden Eigenschaften von B+Bäumen erläutert, um den Leser einen Einblick in die Thematik zu bieten. Am Ende des Abschnitts wird der Programmieraufwand und die Performance der beiden Versionen verglichen. Schlussendlich werden in Abschnitt 5 die gesammelten Informationen in einem Vergleich von OpenCL und SkelCL zusammengefasst und Vorschläge für mögliche Erweiterungen von SkelCL erläutert.

2. Grundlagen

In diesem Abschnitt werden zunächst die grundlegenden Begriffe erklärt, die zum Verständnis dieser Arbeit essentiell sind. Darauf folgend werden die zu untersuchenden Anwendungen sowie verwendeten Programmieransätze beschrieben.

2.1. Die Central Processing Unit

Die *Central Processing Unit* (oft auch kurz Prozessor oder CPU) ist die zentrale Recheneinheit eines Computers. Durch sie werden alle Programme und Datenoperationen ausgeführt. Bei einem Prozessor mit nur einem Kern (engl. *Core*), der nur eine Operation zu einem Zeitpunkt ausführen kann, spricht man von einem skalaren Prozessor. Bei einem Prozessor mit mehreren Kernen spricht man von einem multi-core Prozessor. Die Geschwindigkeit von Prozessoren steigt sehr schnell. Laut dem Moorschen Gesetz verdoppelt sich alle 18 Monate die Anzahl der Transistoren auf einem Chip, womit auch das Leistungspotential der Chips steigt. Da aber bei einer Taktrate von 4GHz, d.h. 4 Milliarden Taktzyklen in der Sekunde, ein Takt nur 0,25ns benötigt und das Licht in dieser Zeit nur 7,5cm zurücklegt, stößt man bei der Erweiterung der skalaren Performance auf physikalische Grenzen. Daher wird versucht den Zuwachs an Transistoren nicht mehr für die Steigerung der Taktrate zu verwenden, sondern zur Erhöhung der Anzahl der Kerne auf einem Prozessor, wodurch auch eine Steigerung der maximalen Performance erzielt werden kann. Die Programmierung von Anwendungen für solche multi-core Prozessoren nennt man parallele Programmierung. Sie hat in den letzten Jahren deutlich an Bedeutung gewonnen, denn um die volle Rechenleistung solcher multi-core Prozessoren zu erreichen, ist es nötig alle Kerne einer CPU gleichermaßen auszulasten. Mit einer heutigen CPU, z.B. dem i7-3770T aus der Intel i-Serie ist eine theoretische Spitzenleistung von 118 Milliarden Gleitpunktberechnungen pro Sekunde (118 GFLOPS) möglich [9].

2.2. Die Graphical Processing Unit

Die zentrale Recheneinheit einer Grafikkarte ist die *Graphical Processing Unit* (GPU). Ursprünglich diente sie nur dazu die CPU bei der Bild- und Videoverarbeitung zu unterstützen. Mit der Zeit wurden ihre Funktionseinheiten (damals als *Shader* bezeichnet) immer allgemeiner und waren nicht mehr nur auf Bildverarbeitung spezialisiert. Aus dieser Entwicklung heraus wurde erkannt, dass sich die Architektur einer GPU gut zur Berechnung hoch paralleler Aufgaben eignet. Dies liegt daran, dass eine GPU, im Gegensatz zur CPU, aus sehr vielen Funktionseinheiten aufgebaut ist, die allerdings nicht so komplex wie die einer CPU sind. Dadurch kann mit einer GPU eine viel höhere theoretische Maximalperformance erzielt werden. Bei einer aktuellen Grafikkarte, z.B. bei der GeForce GTX 680 von NVIDIA, sind theoretisch bis zu 3,09 Billion Gleitpunktberechnungen pro Sekunde (3,09 TFLOPS) möglich [12]. Diese Maximalperformance zu erreichen ist aber sehr schwer, da die klassischen Programmieransätze (z.B. Thread Programmierung) diese hohe Parallelität nicht hinreichend unterstützen und nicht jede Berechnung sich beliebig parallelisieren lässt.

2.3. Flynn'sche Klassifikation

Die Klassifikation von Flynn [8] ist eine häufig verwendete Klassifikation, um die Art eines Parallelrechners darzustellen. Sie beschreibt wie ein Parallelrechner die Merkmale *instruction streams* (Instruktionsströme) und *data streams* (Datenströme) verarbeitet und gibt für diese jeweils zwei mögliche Arten der Ausprägung an, welche zum einen *single* (einzeln) und zum anderen *multiple* (mehrfach) sind. Hieraus ergeben sich vier verschiedene Klassen, die in Tabelle 1 dargestellt sind.

Instruction / Data	Single	Multiple
Single	SISD	MISD
Multiple	SIMD	MIMD

Tabelle 1: Die Klassen der Flynn'schen Klassifikation

- **SISD** (*single-instruction single-data*) bedeutet, dass keine Parallelität vorhanden ist. Es wird immer nur eine Instruktion (*single-instruction*) zu einem Zeitpunkt ausgeführt, welche auf einem Datenelement (*single-data*) arbeitet. CPUs mit einem Kern, welche vor einigen Jahren noch die gängige Rechnerarchitektur war, gehören in diese Klasse.
- **SIMD** (*single-instruction multiple-data*) beschreibt eine Klasse von Parallelrechnern, bei der eine Instruktion (*single-instruction*) gleichzeitig auf mehreren Datenelementen (*multiple-data*) ausgeführt werden kann. Die Kerne einer modernen GPU lassen sich in diese Klasse einordnen. Anders als oft beschrieben, gehört die gesamte GPU nicht der SIMD Klasse an.
- **MISD** (*multiple-instruction single-data*) ist heutzutage keine relevante Klasse, da es gegenwärtig keine Rechner gibt, die sich in diese Klasse einordnen lassen. Bei diesem Prinzip werden mehrere Instruktionen (*multiple-instruction*) gleichzeitig auf einem Datenelement (*single-data*) ausgeführt.
- **MIMD** ist derzeit die am häufigsten anzutreffende Klasse von Rechnern, da ihr multi-core CPUs und GPUs zuzuordnen sind. Hier werden mehrere Instruktionen (*multiple-instruction*) gleichzeitig auf mehreren Datenelementen (*multiple-data*) ausgeführt. Dies ist in multi-core CPUs der Fall, da sie aus mehreren SISD- oder SIMD-Kernen aufgebaut sind. Auch moderne GPUs lassen sich in diese Klasse einordnen, da sie aus mehreren SIMD-Kernen bestehen.

2.4. OpenCL

Die *Open Computing Language* (OpenCL) [10] ist ein freier Standard zur Entwicklung von parallelen Anwendungen. OpenCL wurde erstmals 2008 von der Khronos Group veröffentlicht, wo er von mehreren großen Unternehmen (Apple, AMD, IBM, Intel, NVIDIA

u.a.) kooperativ entwickelt wurde und immer noch erweitert wird. Mit der, auf der Programmiersprache C aufbauenden, Programmiersprache *OpenCL C* ermöglicht OpenCL die einheitliche Entwicklung für CPUs, GPUs und andere Recheneinheiten. Dies erreicht OpenCL durch die Einführung eines eigenen Plattform-, Ausführungs-, Speicher- und Programmiermodells, welche in [10] beschrieben werden.

2.4.1. Das OpenCL Plattformmodell

Ein OpenCL System besteht nach dem OpenCL Plattformmodell aus einem *Host* und mehreren *Devices*. Jedes Device besteht aus ein oder mehreren *Compute Units* (CU), welche wiederum ein oder mehrere *Processing Elements* (PE) beinhalten. Dieses Schema ist in Abbildung 1 dargestellt. Wenn man es auf ein System mit einer multi-core CPU übertragen würde, übernähm die CPU sowohl die Rolle des Hosts als auch des Devices. Die Compute Units des Devices entsprächen den einzelnen CPU Kernen, wobei diese wiederum aus den einzelnen Processing Elements bestehen würden. Interessanter gestaltet sich die Abbildung eines Systems, bestehend aus einer CPU und einer oder mehreren GPUs, auf das OpenCL Plattformmodell. Hier würde wieder die CPU die Rolle des Hosts einnehmen, wobei die GPUs den Devices entsprächen (genau genommen könnte hier die CPU auch einem Device entsprechen). Die Compute Units entsprächen in diesem Fall den SIMD-Kernen der GPU und deren Funktionseinheiten würden schließlich als die Processing Elements abgebildet werden. Durch diesen Vergleich wird deutlich, dass sich das OpenCL Plattform Modell für einfache Strukturen, wie Systeme mit nur einer CPU, eignet aber sich auch auf komplexere Strukturen übertragen lässt.

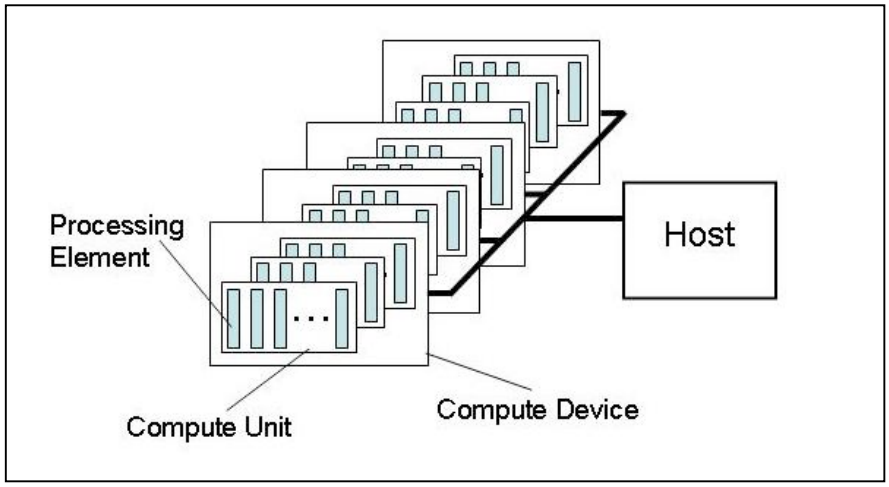


Abbildung 1: Das OpenCL Plattformmodell [10]

2.4.2. Das OpenCL Ausführungsmodell

Das OpenCL Ausführungsmodell beschreibt wie die mit OpenCL entwickelten Anwendungen auf dem System ausgeführt werden. Generell besteht jede OpenCL Anwendung

aus zwei Teilen; dem *Hostprogramm*, welches auf dem Host ausgeführt wird und den *Kerneln*, die auf den Devices ausgeführt werden. Hier zeigt sich ein großer Unterschied im Vergleich zur klassischen Programmierung für parallele Anwendungen (wie z.B. *OpenMP* [3]). Das Hostprogramm läuft nur sequentiell auf dem Host und startet die Kernel, welche parallel auf den Devices ausgeführt werden. In OpenCL werden diese Kernelinstanzen, also die Instanzen die den Kernelcode ausführen, als *Work-Items* bezeichnet. Hier tut sich eine Besonderheit auf, denn diese Work-Items arbeiten nicht nur parallel, sondern können auch bis zu dreidimensional angeordnet werden. Ein einfaches Beispiel hierfür ist die Bearbeitung eines Bildes, bei dem jedes einzelne Pixel verändert werden soll. Hier ordnet man die Work-Items zweidimensional, mit den selben Ausmaßen wie sie dem Bild entsprechen, an. Dann entsprechen die Positionen der Work-Items genau den einzelnen

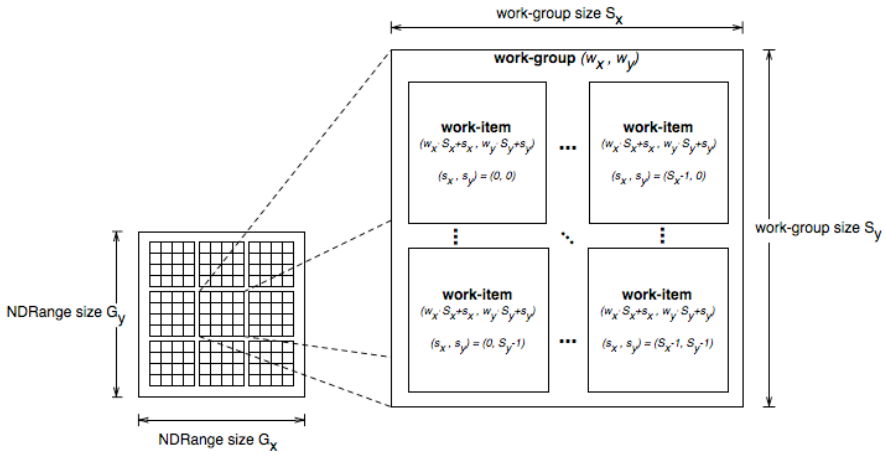


Abbildung 2: Schema einer Work-Item, Work-Group Anordnung [10]

Pixel des Bildes und so kann das Bild parallel, Pixel für Pixel bearbeitet werden. Dies vereinfacht das Programmieren von Algorithmenstrukturen die sich in solche Muster einordnen lassen sehr. Eine weitere Möglichkeit die OpenCL bietet ist das Gruppieren von Work-Items. Diese Gruppen werden als *Work-Groups* bezeichnet und können, wie auch die Work-Items, bis zu dreidimensional angeordnet werden. Eine mögliche Anordnung der Work-Items und Work-Groups ist in Abbildung 2 schematisch dargestellt. Hier ist zu sehen, dass $G_y \times G_x$ Work-Items gestartet werden, die zweidimensional angeordnet sind (links in der Abbildung). Diese werden in Work-Groups zusammengefasst, die ebenfalls zweidimensional angeordnet sind, wobei sich $S_y \times S_x$ Work-Items in einer Work-Group befinden. Rechts in der Abbildung ist zu sehen, dass jedes Work-Item zwei Positionen besitzt, die zum einen die *global ID* darstellen, welche die Position unter allen Work-Items angibt und zum anderen die *local ID*, welche die Position innerhalb der Work-Group angibt. Außerdem besitzt jede Work-Group eine eindeutige Position, welche als *Work-Group ID* bezeichnet wird. Innerhalb der Work-Items kann man auf diese *IDs* zugreifen und sie daher, je nach ID, unterschiedliche Berechnungen oder Berechnungen auf unterschiedlichen Elementen ausführen lassen.

2.4.3. Das OpenCL Speichermodell

Das OpenCL Speichermodell dient der Abstraktion der verschiedenen Speicher. Da z.B. CPUs und GPUs unterschiedliche Speicher besitzen, wird hierdurch eine einheitliche Sicht auf die Speicher ermöglicht. Die vier Speicherarten die OpenCL beschreibt sind der globale, der konstante, der lokale und der private Speicher. In Abbildung 3 ist zu sehen, wie diese Speicher miteinander in Verbindung stehen und woher diese Speicher erreichbar sind. Da die Devices keinen direkten Zugriff auf den Hauptspeicher besitzen, müssen alle zu bearbeitenden Daten in einem dieser Speicher vorliegen. Dies wird dadurch erreicht, dass die Daten von dem Hostprogramm in den globalen oder konstanten Speicher kopiert werden oder indem die Daten als Argumente den Kerneln übergeben werden.

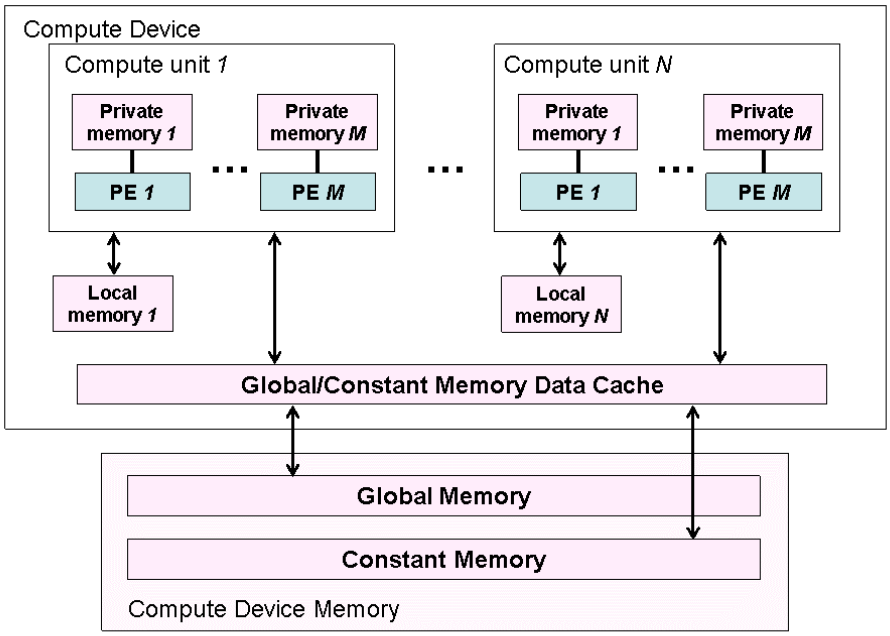


Abbildung 3: Das OpenCL Speichermodell [10]

- Der **globale Speicher** entspricht bei einer GPU dem Grafikspeicher der Grafikkarte. Er ist aus dem Host und aus dem Kernel lesbar und beschreibbar und ist in der Regel sehr viel größer, aber dafür langsamer, als der lokale und der private Speicher. Außerdem stellt er einen gemeinsamen Speicher für alle Work-Items dar.

- Daten die nur vom Host geschrieben werden sollen, also im Kernel nicht bearbeitet bzw. verändert werden sollen, können im **konstanten Speicher** gehalten werden. Da er ein Teil des globalen Speichers ist, stellt auch er einen gemeinsamen Speicher für alle Work-Items dar, der allerdings aus dem Kernel nicht beschreibbar ist. Zusammen mit dem globalen sind sie die einzigen Speicher, die aus dem Hostprogramm les- und beschreibbar sind.

- Die **lokalen Speicher** stellen jeweils einen viel kleineren aber auch schnelleren Speicher dar. Man kann nur aus dem Kernel auf sie zugreifen und sie dienen als gemeinsamer Speicher für alle Work-Items einer Work-Group.
- Die letzten und kleinsten Speicher sind die **privaten Speicher**, sie sind deutlich schneller als der globale und der konstante Speicher. Außerdem sind sie für jedes Work-Item einzigartig und es werden normalerweise Variablen, die im Kernel angelegt werden, in ihnen gehalten.

2.4.4. Aufbau eines OpenCL Programms

Listing 1 zeigt den schematischen Aufbau eines OpenCL Programms, der im Folgenden näher erläutert wird.

1. Zunächst werden mit `clGetPlatformIDs` (in Zeile 2) die verfügbaren Plattformen des Systems ermittelt.
2. `clGetDeviceIDs` (Zeile 4) erzeugt Informationen über die Devices, die durch die ausgewählte Plattform bereitgestellt werden.
3. Danach wird in Zeile 6 mit `clCreateContext` ein *Context* erzeugt. Dieser dient als Verwaltungsobjekt für weitere OpenCL Funktionen.
4. Dann erfolgt, in Zeile 8, das Erstellen einer *Command Queue* mittels `clCreateCommandQueue`. In ihr werden die einzelnen Befehle gespeichert, die später vom Program abgearbeitet werden sollen.
5. Durch den Aufruf von `clCreateProgramWithSource` wird ein *Program* aus dem Kernelquellcode, erzeugt (Zeilen 10 bis 12).
6. Daraufhin wird das Program durch den Aufruf von `clBuildProgram`, in Zeile 14, kompiliert.
7. Mit `clCreateKernel` wird aus dem kompilierten Program ein Kernelobjekt erstellt (Zeile 16).
8. Dann können durch die Aufrufe `clCreateBuffer` und `clSetKernelArg`, zu sehen in den Zeilen 18 bis 20, Argumente auf das Device hochgeladen und an den Kernel übergeben werden.
9. Dann kann der Kernel, wie in Zeile 22, gestartet werden. Dies geschieht mit dem Ausführen von `clEnqueueNDRangeKernel`. Hier werden die Anordnung und Größe der Work-Items und Work-Groups festgelegt.
10. Anschließend können die berechneten Daten mit `clEnqueueReadBuffer` vom Device wieder zurück zum Host kopiert werden (Zeile 25).
11. `clFinish`, in Zeile 26, stellt schlussendlich sicher, dass die Command Queue vollständig abgearbeitet wurde.

```
1  //Ermittle Plattformen
2  clGetPlatformIDs(1, &platform, NULL);
3  //Ermittle Devices und waehle eine GPU
4  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
5  //Erzeuge Context
6  clCreateContext(NULL, 1, &device, NULL, NULL, &err);
7  //Erzeuge command Queue
8  clCreateCommandQueue(context, device, 0, &err);
9  //Erzeuge Programm aus Quelltext
10 const char* kernelSource = "__kernel void (...) {...}";
11 size_t sourceLength = strlen(kernelSource);
12 clCreateProgramWithSource(context,1,&kernelSource,&sourceLength,&err);
13 //Compiliere das Programm
14 clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
15 //Erzeuge Kernel
16 clCreateKernel(program, "add", &err);
17 //Kopiere Daten zu Device
18 cl_mem arg = clCreateBuffer(context, ...);
19 //Setze Argumente
20 clSetKernelArg(kernel, 0, sizeof(arg), &arg);
21 //Fuehre Kernel aus
22 clEnqueueNDRangeKernel(commandQueue, kernel, dimensions, global_offset,
23                        global_size, local_size, 0, NULL, NULL);
24 //Hole Daten vom Device
25 clEnqueueReadBuffer(commandQueue, ...);
26 clFinish(commandQueue);
```

Listing 1: Schematischer Aufbau eines OpenCL Programms

2.5. Algorithmische Skelette

Algorithmische Skelette [6] sind Funktionen höherer Ordnung, die wiederum Funktionen als Argumente entgegennehmen und diese mit einem bestimmten parallelen Berechnungsmuster, ausführen. Algorithmische Skelette kapseln oft benutzte Berechnungsstrukturen und abstrahieren von den Details dieser parallelen Berechnungen. Dadurch erleichtern sie die Parallelisierung von Algorithmen, denn der Entwickler muss nur ein passendes algorithmisches Skelett für seine Berechnung wählen, es in sein Programm einbauen und geeignet anpassen. Da es sich bei C++ um eine imperative und keine funktionale Programmiersprache handelt, unterstützt C++ standardmäßig keine Funktionen höherer Ordnung, was der Grund dafür ist, dass SkelCL diese Funktionen in C++ implementiert.

2.6. Die SkelCL Bibliothek

Um die Programmierung paralleler Anwendungen zu vereinfachen, wird von der Arbeitsgruppe Parallele und Verteilte Systeme an der Universität Münster SkelCL entwickelt [13]. SkelCL ist eine Bibliothek die auf OpenCL aufsetzt und sich algorithmischer Skelette bedient, um oft auftretende Berechnungs- und Kommunikationsmuster zu kapseln.

Die Skelette die SkelCL bereit stellt, zu sehen in Abbildung 4, sind *Map*, *Zip*, *Reduce* und *Scan*.

- **Map** wendet eine Funktion (f) auf einen der Datentypen von SkelCL, Vektor oder Matrix, auf jedes Element an.
- **Zip** verknüpft die Inhalte zweier Datentypen von SkelCL elementweise mit einer angegebenen Funktion ($\oplus$). Es muss sich hierbei um die selben Datentypen handeln, deren Instanzen jeweils gleich groß sind.
- **Reduce** berechnet aus allen Elementen eines Vektors von SkelCL, mittels einer assoziativen Funktion ($\oplus$), einen Wert.
- **Scan** berechnet aus allen Elementen eines SkelCL Vektors einen neuen Vektor. Hier besitzt das letzte Element des Vektors den selben Wert wie es auch das reduce Skelett berechnet hätte, allerdings werden die jeweiligen Zwischenergebnisse zusätzlich in dem Zielvektor gespeichert.

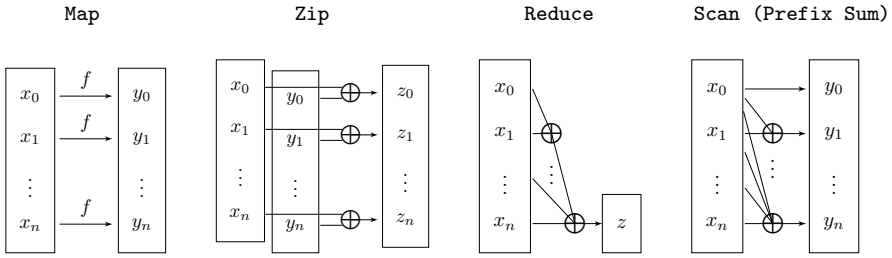


Abbildung 4: Die in SkelCL unterstützten Skelette [14]

SkelCL stellt die abstrakten Datentypen Vektor und Matrix zur Verfügung, mit denen die Verteilung von Daten auf die Devices vereinfacht wird. Die unterstützten Verteilungen, zu sehen in Abbildung 5, sind *Single*, *Block* und *Copy*.

- Bei der Verteilung **Single** werden die Daten nicht verteilt, sondern liegen nur in einem Speicher der Devices vor. Damit kann man erreichen, dass ein Skelett nur auf einem Device ausgeführt wird.
- Sind die Daten **block** verteilt, werden sie in gleich große Stücke zerlegt und auf die verschiedenen Devices verteilt. Somit können Berechnungen, die keine Abhängigkeiten zwischen den einzelnen Daten voraussetzen, multi-Device fähig gemacht werden (d.h. es werden mehrere Devices für die Berechnung verwendet) ohne dass alle Eingabedaten zu jedem Device kopiert werden müssen.
- Durch die **Copy** Verteilung werden die kompletten Eingabedaten auf alle Devices kopiert. Dadurch ist es möglich Berechnungen mit Datenabhängigkeiten, innerhalb der Eingabedaten, multi-Device fähig zu machen, da hier alle Eingabedaten aus den Work-Items verfügbar sind.

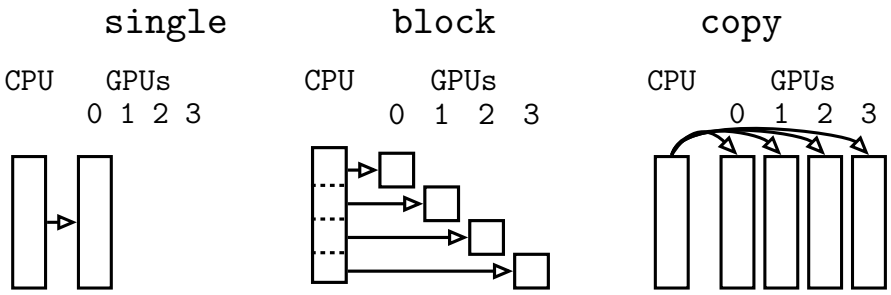


Abbildung 5: Die Verteilungen die SkelCL anbietet [14]

Zur Zeit wird an einer weiteren Verteilung, namens *Overlap* gearbeitet, welche ein neues Skelett benutzt (*MapOverlap*). Diese Verteilung wird in Abschnitt 3, Unterunterabschnitt 3.3.3 vorgestellt.

In Listing 2 ist ein kurzes SkelCL Codebeispiel aufgeführt, welches die Addition von zwei Vektoren mittels SkelCL darstellt. Hier wird als erstes, nach der Initialisierung von SkelCL, das Zip Skelett erstellt (Zeile 8), welches für diese Berechnungsstruktur am besten zutrifft. Danach werden die Vektoren angelegt und mit Daten gefüllt (Zeilen 9 bis 13). Hierbei ist zu beachten, dass die Vektoren in SkelCL sich genauso verhalten wie die der standard C++ Bibliothek (`std::vector`), d.h. sie können unter Anderem mit Iteratoren durchlaufen werden. Danach wird ein neuer Vektor angelegt und mit dem Ergebnis der Addition initialisiert, was in Zeile 15 erfolgt. SkelCL Code ist im Allgemeinen sehr viel kürzer als OpenCL Code, da Operationen wie z.B. das Initialisieren von OpenCL vor dem Nutzer verborgen bleiben.

2.7. Die Rodinia Benchmark-Suite

Rodinia [4, 5] ist eine Benchmark Suite für heterogene Berechnungen. Sie wurde 2009 von Shuai Che, Michael Boyer und Jiayuan Meng von der Universität Virginia vorgestellt. Rodinia wurde mit dem Ziel entwickelt, die Performance von multi-core CPU und GPU Systemen zu evaluieren. Dafür beinhaltet sie mehrere Anwendungen, die in verschiedene Kategorien einteilbar sind. Es existieren z.B. berechnungsintensive, datenintensive und auch synchronisationslastige Anwendungen, mit denen unterschiedliche Performanceaspekte beurteilt werden können. Für diese Bachelorarbeit wurde die OpenCL Implementierung zur Traversierung von B+Bäumen, als Benchmark aus Rodinia, ausgewählt.

2.8. Die Parboil Benchmark-Suite

Parboil [15] ist eine Benchmark Suite, die von der *IMPACT Research Group* an der Universität Illinois entwickelt wird. Sie wurde erstmals im Jahr 2012 vorgestellt und so wie Rodinia dient sie auch der Performacemessung auf multi-core CPU und GPU Systemen. Sie beinhaltet ebenfalls verschiedene Anwendungen in verschiedenen Programmierspra-

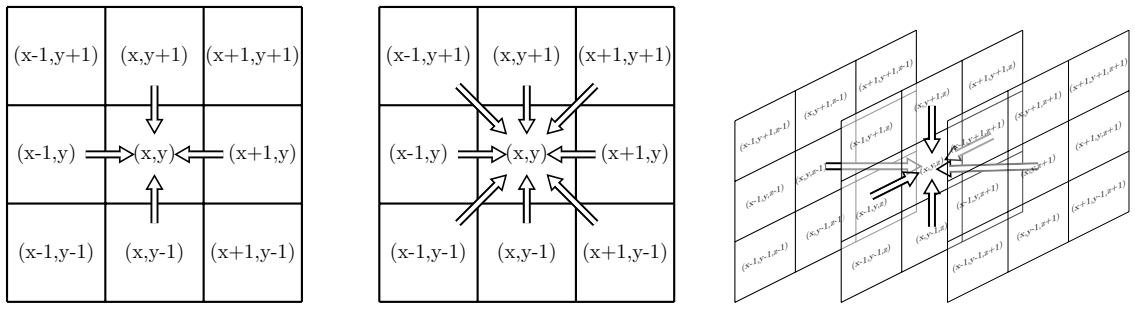
chen. Zusätzlich enthält sie unterschiedlich stark optimierte Versionen der Benchmarks, die z.B. genau auf die Fermi Architektur von NVIDIA [11] abgestimmt sind. In dieser Bachelorarbeit wird die 3D Jacobi Stencil Berechnung aus Parboil benutzt, um die Performance von SkelCL gegenüber OpenCL zu evaluieren.

```
1  #include <SkelCL/SkelCL.h>
2  #include <SkelCL/IndexVector.h>
3  #include <SkelCL/Zip.h>
4  int main(){
5    //Initialisiere SkelCL
6    skelcl::init();
7    //Erstelle Zip Skelett
8    skelcl::Zip<int(int, int)> add("int func(int x, int y){return x+y;}");
9    skelcl::Vector<int> A(1024);
10   skelcl::Vector<int> B(1024);
11   //Fuelle Vektoren mit Daten
12   fillVector(A.begin(), A.end());
13   fillVector(B.begin(), B.end());
14   //Fuehre Addition aus
15   skelcl::Vector<int> C = add(A, B);
16   printVector(C.begin());
17   return 0;
18 }
```

Listing 2: Beispielcode für eine Vektoraddition

3. 3D Jacobi Stencil Berechnung

Stencil Berechnungen sind eine Klasse von iterativen Berechnungen. Sie dienen als Vorlage für Anwendungen, die ähnliche Berechnungsmuster aufweisen, d.h. sie kapseln Berechnungsmuster und erleichtern dadurch deren Verwendung. Es gibt verschiedene Arten von Stencil Berechnungen, die meist auf zweidimensionalen (Matrizen) oder dreidimensionalen Rastern arbeiten. Die typische Abfolge von Stencil Berechnungen beginnt mit dem Aktualisieren jedes Rasterelements. Dafür wird eine bestimmte Rechenvorschrift auf alle Elemente oder auf eine bestimmte Gruppe von Elementen des Raster angewandt. Die neuen Werte dieser Elemente werden meist aus den Werten der umliegenden Elemente ermittelt. Da Stencil Berechnungen meist iterativ sind, wird nachdem alle Elemente aktualisiert wurden, die Berechnung wiederholt. Insgesamt wird das Raster n-mal aktualisiert, wobei n die Anzahl der Iterationen angibt. Um eine bestehende Stencil Berechnung für eigene Berechnungen zu nutzen, muss man die Rechenvorschrift zum Aktualisieren der Rasterelemente anpassen und erhält somit, ohne großen Aufwand, das gewünschte Ergebnis. Stencil Berechnungen lassen sich in verschiedene Kategorien einteilen. Diese hängen von der Anzahl der Elemente ab, die dazu nötig sind um den neuen Wert des zu aktualisierenden Elements zu bestimmen. Abbildung 6 zeigt verschiedene Klassen von Stencil Berechnungen. In Abbildung 6a ist eine 4-Punkt, 2D Stencil Berechnung zu sehen, wobei ihr Name auf die Anzahl der Elemente zurückzuführen ist, die dafür verwendet werden um den neuen Wert des zu aktualisierenden Elements zu bestimmen. Dies ist durch die Pfeile dargestellt, die auf das Element in der Mitte zeigen, wobei ein Pfeil für eine beliebige arithmetische Operationen stehen kann. Eine 8-Punkt, 2D Stencil Berechnung ist in Abbildung 6b dargestellt, wobei analog zur Vorherigen, 8 Nachbarelemente zur Errechnung des neuen Wertes, des zu aktualisierenden Elements, nötig sind. Eine Stencil Berechnung im dreidimensionalen Fall, ist in Abbildung 6c gezeigt. Hier handelt es sich um eine 6-Punkt, 3D Stencil Berechnung. Der einzige Unterschied zum zweidimensionalen Fall ist, dass zusätzlich Nachbarelemente auf der z-Achse, zur Berechnung des neuen Elements, herangezogen werden.



(a) 4-Punkt 2D Stencil Berechnung

(b) 8-Punkt 2D Stencil Berechnung

(c) 6-Punkt 3D Stencil Berechnung

Abbildung 6: Beispiele für Klassen von Stencil Berechnungen

3.1. 3D Jacobi Stencil

Die 3D Jacobi Stencil Berechnung wird auf einem dreidimensionalen Raster ausgeführt. Sie stellt einen iterativen Jacobi Löser dar, welcher die Wärmeleitungsgleichung auf einem strukturierten 3D Raster beschreibt [15]. Sie aktualisiert alle Elemente, die nicht auf einer der äußeren Ebenen liegen (dem Rand) nach der Vorschrift:

$$J_{(x,y,z)} = c_1\Big(J_{(x-1,y,z)}+J_{(x+1,y,z)}+J_{(x,y-1,z)}+J_{(x,y+1,z)}+J_{(x,y,z-1)}+J_{(x,y,z+1)}\Big)-c_0 J_{(x,y,z)}$$

Wobei J das Raster bezeichnet, $J_{(x,y,z)}$ das Element des Rasters an der Stelle (x, y, z) und c_0, c_1 skalare Faktoren. Da bei dieser Vorschrift das zu aktualisierende Element auch zur Berechnung des neuen Wertes herangezogen wird, gehört die 3D Jacobi Stencil Berechnung nicht zur Klasse der 6-Punkt 3D Stencil Berechnungen (Abbildung 6c), sondern zur Klasse der 7-Punkt 3D Stencil Berechnungen.

Jacobi Stencil werden hauptsächlich in der Physik eingesetzt, da sie die Wärmeleitungsgleichung numerisch lösen. Wärmeleitung zu untersuchen ist vor allem dort von Interesse, wo keine Wärmeleitung erwünscht ist, wie z.B. bei Isolierproblemen. Man möchte z.B. herausfinden wie viel Wärme durch eine Wand, eines bestimmten Materials, aus dem Inneren eines Hauses nach Außen gelangt. Hier beschreibt das Raster die Wand in ihrem Ursprungszustand, wobei durch das Iterieren, mit der Jacobi Stencil Berechnung, der zeitliche Verlauf der Wärmeleitung beschrieben wird. Die Faktoren c_0 und c_1 der Gleichung stellen die Materialeigenschaften des zu untersuchenden Objekts dar. Die Wärmeleitungsgleichung wird durch partielle Differentialgleichungen beschrieben, wobei das Lösen dieser Gleichungen sehr komplex werden kann. Daher werden diese Gleichungen meist (wie auch hier) durch numerische Verfahren gelöst.

3.2. OpenCL Implementierung

Die hier vorgestellte OpenCL Version, stammt aus der Benchmark Suite Parboil und wurde von Li-Wen Chang geschrieben. Sie liegt in unterschiedlich stark optimierten Versionen vor, wobei für den Vergleich mit der SkelCL Version die Basisversion, d.h. die Version ohne Optimierungen herangezogen wurde.

Das generelle Vorgehen bei dieser Berechnung ist als Ausschnitt in Listing 3 dargestellt. Zunächst wird das Startraster, in Zeile 5, aus einer Datei gelesen, welche aus $x \times y \times z$ Elementen besteht, wobei x, y, z die Größe des Rasters in der jeweiligen Dimension angeben. Dieses Raster wird beim Einlesen in ein Array (`h_A0`) gespeichert, wobei alle Zeilen einer Ebene, sowie die einzelnen Ebenen (die auf den z-Koordinaten) hintereinander geschrieben werden. Dadurch geht die dreidimensionale Struktur des Rasters verloren und es muss etwas Rechenaufwand, bei der Adressierung der einzelnen Rasterelemente über ihre Koordinaten, betrieben werden. Um ein Rasterelement $J_{(x,y,z)}$ zu erreichen, muss auf das Arrayelement `h_A0[x+(nx*y)+(ny*z)]` zugegriffen werden, wobei nx, ny die Größen des Rasters in x bzw. y Dimension bezeichnen und (x, y, z) die Koordinaten des Rasterelements. Außerdem wird ein weiterer Speicherbereich gleicher Größe alloziert

```
1  //Reserviere Speicher fuer Eingabe- und Ausgaberaster
2  float *h_A0    = (float*)malloc( sizeof(float) * nx*ny*nz);
3  float *h_Anext = (float*)malloc( sizeof(float) * nx*ny*nz);
4  //Schreibe Raster in h_A0
5  read_data(h_A0, ...);
6  //Berechne die globale und lokale Groesse
7  int tx = 256;
8  size_t block[3] = {tx, 1, 1};
9  size_t grid[3] = {(nx-2+tx-1)/tx*tx, ny-2, nz-2};
10 for(t=0;t<iteration;t++){
11    //Fuehre den Kernel mit h_A0 als Eingabe und h_Anext als Ausgabe aus
12    clStatus = clEnqueueNDRangeKernel(clCommandQueue,clKernel,3,NULL,
13                                      grid,block,0,NULL,NULL);
14    CHECK_ERROR("clEnqueueNDRangeKernel")
15    //Vertausche h_A0 und h_Anext
16    cl_mem d_temp = d_A0;
17    d_A0 = d_Anext;
18    d_Anext = d_temp;
19    clStatus = clSetKernelArg(clKernel,2,sizeof(cl_mem),(void*)&d_A0);
20    clStatus = clSetKernelArg(clKernel,3,sizeof(cl_mem),(void*)&d_Anext);
21 }
```

Listing 3: Codeausschnitt der OpenCL Version der 3D Jacobi Stencil Berechnung aus Parboil

(`h_Anext` in Zeile 3), in den das aktualisierte Raster geschrieben wird. In jeder Iteration wird ein Kernel gestartet, der das Eingaberaster aktualisiert, d.h. die vorgestellte Rechenvorschrift wird auf alle Elemente des Rasters, bis auf den Rand, angewandt und in das Ausgaberaster geschrieben (Zeile 12). Um dies iterativ durchzuführen, werden die beiden Speicherbereiche vertauscht (Zeilen 16 bis 20) und die Berechnung von Neuem ausgeführt. Dieses Vertauschen ist ein wichtiger Punkt in der Berechnung, denn hier werden die beiden Zeiger innerhalb des globalen Speichers der GPU vertauscht. Dies führt dazu, dass das Raster auf dem Speicher der GPU belassen werden kann und nicht zum Vertauschen auf den Host geladen werden muss, was einen hohe Datentransfer und somit eine schlechtere Performance zur Folge haben würde. Daher muss das Raster nur einmal auf den GPU Speicher geladen werden und nicht in jeder Iteration, vom Device zum Host und vice versa. Angeordnet werden die Work-Items dreidimensional, wie in Zeile 9 zu sehen ist, da das Raster ursprünglich eine dreidimensionale Struktur besessen hat. Allerdings werden, im Unterschied zur Größe des Raster, immer $nx \times ny \times nz$ Work-Items gestartet, wobei nx ein Vielfaches von 256 darstellt und ny, nz der Größe der jeweiligen Dimension, ohne den Randpunkten, entsprechen. Die Größe nx wurde so gewählt, damit das größtmögliche Raster, welches in dieser Implementierung eines der Größe $512 \times 512 \times 64$, also eines mit 2^{24} Elementen, noch in die maximale Anzahl an Work-Groups eingeteilt werden kann. Diese Größe $512 \times 512 \times 64$ wurde gewählt, da sie z.Z. bei den meisten GPUs der maximale Anzahl an Work-Items in den Dimensionen x, y, z entspricht. Da $\frac{2^{24}}{2^{16}} = 256$, wobei 2^{16} die maximale Anzahl an Work-Groups dar-

stellt, werden die Work-Items in Work-Groups der Größe $256 \times 1 \times 1$ zusammengefasst, wodurch sich das Anordnen von nx Work-Items in x-Dimension erklärt, wobei nx ein Vielfaches von 256 darstellt. Dieses Anordnen ist in Zeile 9 bzw. bei der Ausführung in Zeile 12 zu sehen.

Im Kernel (Listing 4) werden alle Elemente, die nicht auf dem Rand des Rasters liegen, aktualisiert. Dies wird dadurch erreicht, dass die globalen IDs in jeder Dimension zuvor inkrementiert werden, um die ersten Elemente in jeder Dimension, welche auf dem Rand liegen, auszuschließen. Da in der zweiten und dritten Dimension, wie zuvor erwähnt, nur $y-2$ bzw. $z-2$ Work-Items gestartet wurden, werden mit dieser Anordnung auch direkt die letzten Elemente in diesen Dimensionen, die auf auf dem Rand liegen, ausgeschlossen. Nun muss noch mit `if(i<nx-1)` in Zeile 8 überprüft werden, ob das zu bearbeitende Element, in der ersten Dimension, auf dem Rand des Rasters liegt. Durch die Speicherung des Rasters in einem eindimensionalen Array, ist die Adressierung der einzelnen Rasterelemente über ihre Koordinaten, wie schon eingangs erwähnt, nicht mehr trivial. Hierfür wurde ein Makro eingefügt, welches die Umrechnung von Koordinaten in Indizes im Array übernimmt.

```
1  #define Index3D(_nx,_ny,_i,_j,_k) ((_i)+_nx*((_j)+_ny*(_k)))
2  __kernel void naive_kernel(float c0,float c1,__global float* A0,
3                             __global float *Anext,int nx,int ny,int nz){
4    //Lasse alle Elemente bei denen eine Koordinate 0 ist aus
5    int i = get_global_id(0)+1;
6    int j = get_global_id(1)+1;
7    int k = get_global_id(2)+1;
8    if(i<nx-1){ //Aktualisiere die Elemente mit x-Dimension == x nicht
9      Anext[Index3D (nx, ny, i, j, k)] = c1 *
10       ( A0[Index3D (nx, ny, i, j, k + 1)] +
11         A0[Index3D (nx, ny, i, j, k - 1)] +
12         A0[Index3D (nx, ny, i, j + 1, k)] +
13         A0[Index3D (nx, ny, i, j - 1, k)] +
14         A0[Index3D (nx, ny, i + 1, j, k)] +
15         A0[Index3D (nx, ny, i - 1, j, k)] )
16       - A0[Index3D (nx, ny, i, j, k)] * c0;
17   }
18 }
```

Listing 4: Kernelcode der OpenCL Version der 3D Jacobi Stencil Berechnung aus Parboil

3.3. SkelCL Implementierung

Um die verschiedenen Datentypen und Verteilungen, die SkelCL bietet, zu testen, wurden mehrere SkelCL Versionen der 3D Jacobi Stencil Berechnung implementiert. Diese werden im Folgenden dargestellt, wobei die letzte Version sich auf eine Erweiterung von SkelCL bezieht, die zur Zeit noch nicht veröffentlicht wurde. Die Auswertungen der unterschiedlichen Implementierungen sind am Ende des Kapitels zusammengefasst.

3.3.1. SkelCL Implementierung mit Vektoren

Diese Version der Jacobi Stencil Berechnung orientiert sich stark an der OpenCL Version und daher wurde die Konvertierung, des Rasters, in eine eindimensionale Datenstruktur übernommen. Im Gegensatz zu der OpenCL Version wurden jedoch SkelCL Vektoren und nicht einfache Arrays verwendet. Das Raster wird zunächst, wie in Listing 5 zu sehen, durch elementweises Einlesen aus einer Datei, in den Vektor `input` geschrieben (Zeile 7), wobei der Vorteil eines Vektors gegenüber eines Arrays unter anderem darin liegt, dass der Vektor seine Größe speichert. Ein spezieller Vektortyp den SkelCL bietet ist der `IndexVector`, welcher bei der Definition automatisch mit den Werten von 0 bis zur angegebenen Größe initialisiert wird. Da aus dem Kernel heraus die einzelnen Raste-

```
1  //SkelCL Initialisieren mit einem Device
2  skelcl::init(nDevices(1));
3  //Raster und Positionen anlegen
4  skelcl::IndexVector positions(nx*ny*nz);
5  skelcl::Vector<float> input(nx*ny*nz);
6  //Raster einlesen
7  readGrid(input.begin(), inputFilename, nx, ny, nz);
8  //Kernel in Map Skelett einlesen
9  skelcl::Map<float(skelcl::Index)> f(std::ifstream("kernel_vec.cl"));
10 //Verteilungen setzen
11 positions.setDistribution(skelcl::distribution::Copy(positions));
12 input.setDistribution(skelcl::distribution::Copy(input));
13 //Berechnungen ausfuehren
14 for (int iter=0; iter<iterations; iter++){
15    input = f(positions, input, c0, c1, nx, ny, nz);
16 }
```

Listing 5: Hostcode der SkelCL Version der 3D Jacobi Stencil Berechnung mit Vektoren

relemente identifizierbar sein müssen und diese nicht über die lokalen oder globalen IDs bestimmt werden können, wurde dieser `IndexVector` verwendet. Die Bestimmung der Koordinaten über die lokalen und globalen IDs funktioniert aus dem Grund nicht, da die Daten, in einer multi-GPU fähigen Version, über mehrere GPUs verteilt sind und die lokalen und globalen IDs dies nicht berücksichtigen können. Die Größe des `IndexVectors` wird also durch die Größe des Rasters bestimmt (Zeilen 4 und 5), wodurch klar wird, dass jedes Rasterelement einem Element aus dem IndexVektor zugeordnet ist. Außerdem bekommt der `IndexVector`, in dieser Version, die gleiche Verteilung wie das Raster, welche daher im Folgenden, bei `positions` und `input`, auf copy gesetzt wird (Zeilen 11 und 12). Dadurch ist es zwar nicht möglich diese Berechnung auf mehrere GPUs zu verteilen, allerdings muss, wie in der OpenCL Version, das Raster nicht in jeder Iteration vom Device heruntergeladen (und wieder herauf) werden. Um die Berechnung durchzuführen, wird das Map Skelett verwendet, wodurch für jede Position ein Work-Item gestartet wird. Wie in Zeile 15, im zweiten Argument des Skeletts zu sehen ist, wird das Raster dem Kernel übergeben und im globalen Speicher gehalten, wodurch es für alle Work-Items verfügbar ist. Durch die Position, die in jedem Work-Item unterschiedlich

ist, kann der neue Wert für das jeweilige Rasterelement bestimmt werden. Damit sind alle Initialisierungen erledigt und die Berechnung wird mit dem Map Skelett mehrmals (`iterations`-Mal) ausgeführt.

```
1  float func(int position, global float *input, float c0, float c1,
2             int nx, int ny, int nz){
3    //Ermittle die x, y und z Koordinate
4    int x = position%nx;
5    int y = position/ny%ny;
6    int z = position/(nx*ny);
7    //Pruefe ob sich Element auf Aussenseite befindet
8    bool outside = x==0 | x==nx-1 | y==0 | y==ny-1 | z==0 | z==nz-1;
9    //Wenn Element auf der Aussenseite, aktualisiere nicht
10   if(!outside)
11     return c1 *
12       (
13        input[position     +1] +
14        input[position     -1] +
15        input[position    +nx] +
16        input[position    -nx] +
17        input[position +nx*ny] +
18        input[position -nx*ny]
19         ) - input[position] * c0;
20   return input[position];
21 }
```

Listing 6: Kernelcode der SkelCL Version der 3D Jacobi Stencil Berechnung mit Vektoren

Im Kernel erscheint der übergebene `IndexVector`, durch das Map Skelett, als Integer, welcher die Position im Raster beschreibt (das erste Argument der Kernelfunktion, Listing 6). Durch das zweite Argument `global float *input` ist das Raster für jedes Work-Item vollständig sichtbar. Da in SkelCL die Work-Items nicht dreidimensional angeordnet werden und es keine Datentypen gibt, die eine dreidimensionale Struktur bieten, ist es in dieser Version nicht möglich die Identifizierung der Elemente, die auf dem Rand des Rasters liegen, über die x-, y- und z-Koordinaten vorzunehmen. Um zu bestimmen, ob ein Rasterelement auf dem Rand des Rasters liegt, werden zuerst seine x-, y- und z-Koordinaten errechnet (Zeilen 4 bis 6). Daraus kann bestimmt werden, ob ein Element auf dem Rand liegt und somit nicht aktualisiert werden muss oder ob es innerhalb des Rasters liegt und eine Aktualisierung vorgenommen werden muss. Hier zeigt sich ein Nachteil gegenüber der OpenCL Implementierung, denn durch die Bedingung `if(!outside)`, in Zeile 10, entsteht eine Divergenz in den Work-Items, d.h. mehrere Work-Items müssten gleichzeitig unterschiedliche Befehle abarbeiten, je nachdem ob die Bedingung wahr ist oder nicht. Da die SIMD-Architektur der GPU erzwingt, dass die unterschiedlichen Zweige nacheinander abgearbeitet werden, kann dies zu einer Verlangsamung der Berechnung führen. Zu beachten ist, dass die Bestimmung, ob ein Element auf dem Rand des Rasters liegt, keine Divergenz hervorruft, da ausschließlich lo-

gische Verknüpfungen benutzt werden. Außerdem besteht ein kleiner Unterschied in der Rückgabe der aktualisierten Rasterelemente, denn während bei der OpenCL Version, die aktualisierten Elemente in ein zweites Raster geschrieben wurden, werden die aktualisierten Elemente einfach vom Kernel zurückgegeben, was eine intuitivere Programmierung ermöglicht.

3.3.2. SkelCL Implementierung mit Matrizen

Im Gegensatz zu der ersten Version mit Vektoren, ist diese Version (Listing 7), die Matrizen benutzt, multi-GPU fähig. Dies liegt jedoch nicht an der Verwendung von Matrizen anstatt Vektoren, sondern an der verwendeten Verteilung. In dieser Version gibt

```
1  //Benutze alle GPUs
2  skelcl::detail::DeviceProperties d=allDevices();
3  d.deviceType(device_type::GPU);
4  skelcl::init(d);
5  //Erstelle Matrix mit Positionen sowie Eingabe- und Ausgaberaster
6  skelcl::Matrix<pos_t> positions({ny*nz, nx});
7  initPositions(positions.begin(), nx, ny, nz);
8  skelcl::Matrix<float> output({nz*ny, nx});
9  skelcl::Matrix<float> input({ny*nz, nx});
10 //Erstelle Zeiger auf Eingabe- und Ausgaberaster damit diese vertauscht
       werden koennen
11 skelcl::Matrix<float> *iPtr = &input;
12 skelcl::Matrix<float> *oPtr = &output;
13 //Lese Raster
14 readGrid(input.begin(), inputFilename, nx, ny, nz);
15 //Erstelle Map
16 skelcl::Map<float(pos_t)> f(std::ifstream("kernel_mat.cl"));
17 //Setzte Verteilungen
18 positions.setDistribution(skelcl::distribution::Block(positions));
19 input.setDistribution(skelcl::distribution::Copy(input));
20 output.setDistribution(skelcl::distribution::Block(output));
21 for (int iter=0; iter<iterations; iter++){
22     //Berechne mit *iPtr als Eingabe- und *oPtr als Ausgaberaster
23     *oPtr = f(positions, *iPtr, c0, c1, nx, ny, nz);
24     //Vertausche die beiden Zeiger
25     Matrix<float> *tmp;
26     tmp=iPtr;
27     iPtr=oPtr;
28     oPtr=tmp;
29     //Passe die Verteilungen an
30     (*iPtr).setDistribution(distribution::Copy(*iPtr));
31     (*oPtr).setDistribution(distribution::Block(*oPtr));
32 }
```

Listing 7: Hostcode der SkelCL Version der 3D Jacobi Stencil Berechnung mit Matrizen

es zwei Speicherbereiche, im globalen Speicher, in denen das Raster gespeichert wird, wobei es sich bei dem einen um das Eingaberaster und bei dem anderen um das Ausga-

beraster handelt. Zuerst wird eine Liste aller gefundenen GPUs erstellt (Zeilen 2 und 3) und diese an `skelcl::init` (Zeile 4) übergeben. Dadurch werden von SkelCL nur die GPUs für die nachfolgende Berechnung verwendet. Im Folgenden werden die Matrizen für die beiden Raster und die Positionen erstellt (Zeilen 8 und 9). Hier zeigt sich ein Unterschied zur zuvor vorgestellten Version, denn es werden zusätzlich, in den Zeilen 11 und 12, zwei Zeiger angelegt (`iPtr` und `oPtr`) die auf das Eingaberaster `input` bzw. auf das Ausgaberaster `output` zeigen. Nach dem Einlesen des Rasters, wird analog zur Version mit Vektoren, das Map Skelett (Zeile 16) verwendet. Die Verteilungen sind in dieser Version etwas anders, denn das Eingaberaster besitzt zwar wieder die Verteilung Copy (zu sehen in Zeile 19), aber die beiden anderen Matrizen werden block verteilt, um den Datentransfer gering zu halten (Zeilen 18 und 20), wodurch diese Berechnung multi-GPU fähig wird. Nun wird in jeder Iteration das Eingaberaster komplett auf alle Devices hochgeladen. Die Positionen und das Ausgaberaster werden allerdings, da sie

```
1  float func(pos_t position, float_matrix_t input, float c0, float c1,
2             int nx, int ny, int nz){
3    //Ermittle die x, y und z Koordinate
4    int x = position.x;
5    int y = position.y%ny;
6    int z = position.y/ny;
7    //Pruefe ob sich Element auf Aussenseite befindet
8    bool outside = x==0 | x==nx-1 | y==0 | y==ny-1 | z==0 | z==nz-1;
9    if(!outside)
10     return c1 *
11       (
12        get(input, position.x-1,  position.y) +
13        get(input, position.x+1,  position.y) +
14        get(input, position.x,  position.y-1) +
15        get(input, position.x,  position.y+1) +
16        get(input, position.x, position.y-ny) +
17        get(input, position.x, position.y+ny)
18         ) - get(input, position.x, position.y) * c0;
19   return get(input, position.x, position.y);
20 }
```

Listing 8: Kernelcode der SkelCL Version der 3D Jacobi Stencil Berechnung mit Matrizen

block verteilt sind, auf die verschiedenen Devices aufgeteilt. Daher besitzt jedes Device einen anderen Teil an Positionen und somit auch einen anderen Teil an Elementen, die es zu berechnen hat. Ist die Aktualisierung auf allen Devices vollständig, können die einzelnen Blöcke, aufgrund der Verteilung des Ausgaberasters, zusammengesetzt werden. Durch Vertauschen der zusätzlichen Zeiger (25 bis 31) wird erreicht, dass das jetzige Ausgaberaster in der nächsten Iteration als Eingaberaster dient und vice versa. Hier muss noch beachtet werden, dass die Verteilungen, die mit dem Zeigertausch erhalten geblieben sind, neu gesetzt werden müssen, da sonst das neue Eingaberaster block und das neue Ausgaberaster copy verteilt wären.

In Listing 8 ist der dazugehörige Kernelcode vorgestellt. Hier werden die Koordinaten des Work-Items bestimmt, um damit zu prüfen, ob das Work-Item das Element aktualisieren soll oder ob es auf dem Rand des Rasters liegt. Dies gestaltet sich etwas einfacher als in der vorherigen Version, da durch die Verwendung eines `struct`, welcher zwei integer Werte für die x- und y-Koordinaten hält, dieser in den Work-Items zur Verfügung steht. Auf die Benutzung der `IndexMatrix`, einer Matrix die SkelCL bereitstellt und genau diese zuvor beschriebene Aufgabe löst, wurde an dieser Stelle verzichtet, da die Performance sehr unter der Verwendung der `IndexMatrix` litt. Außerdem steht durch die Übergabe einer Matrix (`float_matrix_t input`) die Funktion `get` (Zeilen 12 bis 19) zur Verwendung bereit, mit der auf bestimmte Koordinaten innerhalb der Matrix zugegriffen werden kann. Hierdurch wird der Kernelcode deutlich besser lesbar, denn es muss nicht mehr, außer bei den Elementen die auf der z-Achse die Nachbarn des zu bearbeitenden Elements sind, ein Offset berechnet werden, sondern es kann einfach auf die Nachbarn des Elements zugegriffen werden. Dies macht deutlich, dass SkelCL zwar einen einfachen Umgang mit zweidimensionalen Datenstrukturen ermöglicht, allerdings bei dreidimensionalen etwas getrickst werden muss, da sie auf die zweidimensionalen abgebildet werden müssen. Eine entsprechende Funktionalität für dreidimensionale Strukturen ist wünschenswert.

3.3.3. SkelCL Implementierung mit MapOverlap

Zur Zeit wird als Erweiterung von SkelCL, ein Skelett namens *MapOverlap* entwickelt. Ziel von MapOverlap ist die Vereinfachung der Bearbeitung von zweidimensionalen Daten, bei denen es Datenabhängigkeiten, innerhalb der Eingabedaten, bei den Berechnungen gibt. Das MapOverlap Skelett kann zur Implementierung von 2D Stencil Be-

```
1  skelcl::init();
2  //Raster und Positionen anlegen
3  skelcl::Matrix<float> input({ny*nz, nx});
4  //Raster einlesen
5  readGrid(input.begin(), inputFilename, nx, ny, nz);
6  //Kernel in MapOverlap Skelett einlesen, als Overlap alle direkten
       Nachbarelemente
7  //Bei Zugriff aus dem Raster heraus gibt -1 zurueck
8  skelcl::MapOverlap<float(float)>f(std::ifstream("kernel_mo.cl"),
9                                    1, SCL_NEUTRAL, -1);
10 //Verteilungen setzen
11 input.setDistribution(distribution::Overlap(input, 1));
12 //Berechnungen ausfuehren
13 for (int iter=0; iter<iterations; iter++){
14    input = f(input, c0, c1);
15    input.copyDataToHost();
16 }
```

Listing 9: Schema für Hostcode mit MapOverlap für 2D Raster

rechnungen verwendet werden. Dabei kann durch einen Parameter angegeben werden, in welchem Radius Nachbarelemente, aus dem Kernel heraus, zugänglich sein sollen. Außerdem bietet es eine elegante Behandlung für den Fall, dass auf Elemente außerhalb der übergebenen Matrix zugegriffen wird. Hierfür gibt es zwei Möglichkeiten, entweder wird ein neutraler, vorher festgelegter Wert zurückgegeben, oder es wird der nächstgelegene Wert aus der Matrix zurückgegeben. Um dieses Skelett multi-GPU fähig zu machen, wurde zusätzlich eine neue Verteilung eingeführt. Bei der *Overlap* Verteilung ist es möglich anzugeben, wie viele Zeilen der Matrix sich bei der Verteilung der Daten überlagern sollen. Die Matrix wird zwar geteilt um sie auf mehrere Devices übertragen zu können, allerdings überlagern sich diese Teile um eine vorher definierte Anzahl an Zeilen.

Das MapOverlap Skelett lässt sich sehr gut auf die 2D Stencil Berechnungen Anwenden, die analog zu den 3D Berechnungen funktionieren. In Listing 9 ist ein schematischer Hostcode für die 2D Jacobi Stencil Berechnung, mittels des MapOverlap Skeletts und der Overlap Verteilung, angegeben. Hier fällt direkt auf, dass es nicht mehr nötig ist, einen `IndexVector` oder eine `IndexMatrix` mit Positionen zu erzeugen, da direkt aus dem Kernel die umliegenden Elemente adressiert werden können. In Zeile 8 wird bei

```
1  float func(float* input, float c0, float c1){
2    //Pruefe ob sich Element auf Aussenseite befindet
3    bool outside = get(input, 0, 1) == -1 | get(input, 0,-1) == -1
4                 | get(input, 1, 0) == -1 | get(input,-1, 0) == -1;
5    //Wenn Element auf der Aussenseite, aktualisiere nicht
6    if(!outside)
7      return c1 *
8        (
9         get(input, 0, 1) +
10        get(input, 0,-1) +
11        get(input, 1, 0) +
12        get(input,-1, 0)
13         ) get(input, 0, 0) * c0;
14   return get(input, 0, 0);
15 }
```

Listing 10: Schema für Kernelcode mit MapOverlap für 2D Raster

der Erzeugung des MapOverlap Skeletts, nach dem Quellcode des Kernels, der Radius als Argument angegeben. Außerdem wird mit `SCL_NEUTRAL` bestimmt, dass bei einem Zugriff außerhalb der Matrix, der angegebene Wert -1 zurückgegeben wird. Bei dem Festlegen der Verteilung (Zeile 11), wird der Radius als weiteres Argument mitgegeben, womit erreicht wird, dass die einzelnen Teile, auf den Devices, sich jeweils um eine Zeile überlagern (engl. *overlap*). Bei der Ausführung der Iterationen müssen dann, nach jeder Aktualisierung des Rasters, alle Daten zurück zum Host kopiert (Zeile 15) und dann wieder zu den Devices hochgeladen werden. Dies liegt daran, dass die sich überlagernden Zeilen, für welche die Verteilung sorgt, nicht aktualisiert werden, da ihre neuen Werte evtl. auf einem anderen Device berechnet wurden.

Der Kernelquellcode (Listing 10) profitiert von der Verwendung des MapOverlap Skeletts. Es ist nun einfacher möglich, die umliegenden Elemente mittels `get(matrix,dx,dy)` zu adressieren, wobei `dx` und `dy` die Weite in x- bzw. y-Richtung angeben. Dies ist in den Zeilen 9 bis 14 zu sehen, wo die Werte der Nachbarelemente bezogen werden. Mit `get(matrix,0,0)` im Kernel, erhält man das zu aktualisierende Element, welches man vorher mit Hilfe der übergebenen Positionen berechnen musste. Die Abfrage ob man sich auf dem Rand des Rasters befindet, gestaltet sich zwar immer noch recht aufwendig, ist aber um einiges besser lesbar. Es müssen nicht mehr die Dimensionen des Rasters mitgegeben werden, da diese nicht mehr zum Bestimmen der Positionen und der Elemente auf dem Rand benötigt werden.

3.4. Evaluation

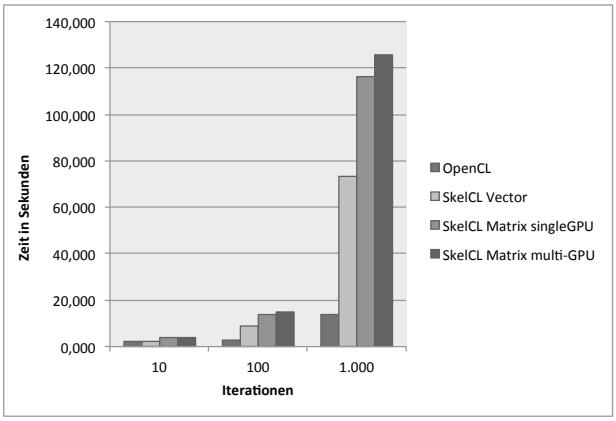
Als Testsysteme wurden ein System, welches mit einer 4 Kern Intel Xeon E5520 CPU mit 2,27GHz und 4 NVIDIA Tesla C1060 GPUs ausgestattet ist, verwendet (*System 1*). Zusätzlich wurde ein System, welches über eine 2 Kern Intel Core 2 6600 CPU mit 2,4GHz, zwei NVIDIA GeForce 9800 GX2 GPUs und eine NVIDIA GeForce GTX 480 GPU verfügt, verwendet (*System 2*). Die maximale Performance der verwendeten GPUs ist in Tabelle 2 dargestellt.

	max. Performace in GFLOPS, SP (Single Precision)
NVIDIA Tesla C1060	936
NVIDIA GeForce GTX 480	1345
NVIDIA GeForce 9800 GX2	2×576

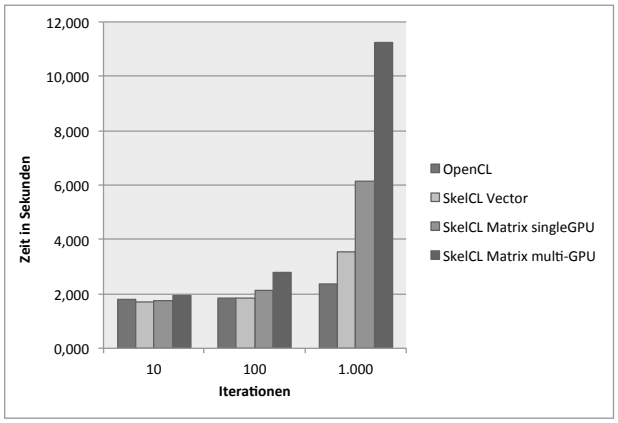
Tabelle 2: Max. Performance der verwendeten GPUs (siehe NVIDIA)

Auf beiden Systemen wurden Raster der Größe $128 \times 128 \times 32$ und $512 \times 512 \times 64$ verwendet. Es wurde mit 10, 100 und 1.000 Iterationen gemessen. Außerdem wurde die SkelCL Version mit Matrizen, auf einer GPU und auf allen GPUs der Systeme, ausgeführt, um die Skalierung auf mehreren GPUs zu beurteilen.

In Abbildung 7 sind die Laufzeiten der verschiedenen Versionen bei variierender Anzahl an Iterationen, auf System 1, dargestellt. In Abbildung 7a ist deutlich zu erkennen, dass die OpenCL Version, bei dem großen Raster, eine viel kürzere Laufzeit besitzt als die SkelCL Versionen. Die OpenCL Version benötigte, bei dem großen Raster, für 1.000 Iterationen ca. 13 Sekunden wohingegen die SkelCL Version, welche die Vektoren benutzt, 73 Sekunden, also eine ganze Minute länger, benötigte. Dies entspricht ungefähr dem Faktor 5,5, was vermuten lässt, dass von SkelCL Datentransfers initiiert werden, obwohl sie bei der Vektor Version nicht gewünscht sind. In der SkelCL Vektor Version wurden, wie in Unterunterabschnitt 3.3.1 beschrieben, die Verteilungen der Daten während der Iterationen nicht verändert, was eigentlich zu dem Effekt führen sollte, dass wie in der OpenCL Version, die Daten nur einmal auf die GPU geladen werden müssen und dort



(a) Raster der Größe $512 \times 512 \times 64$

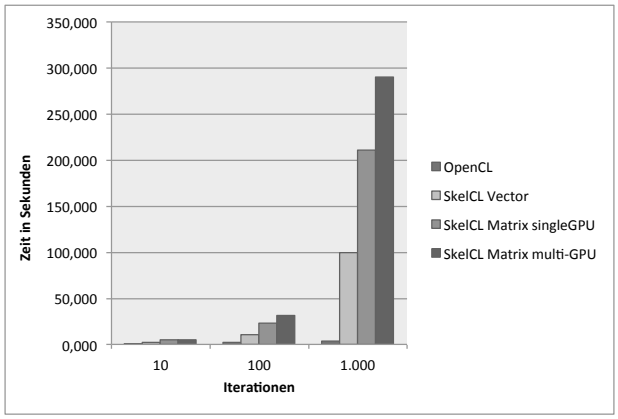


(b) Raster der Größe $128 \times 128 \times 32$

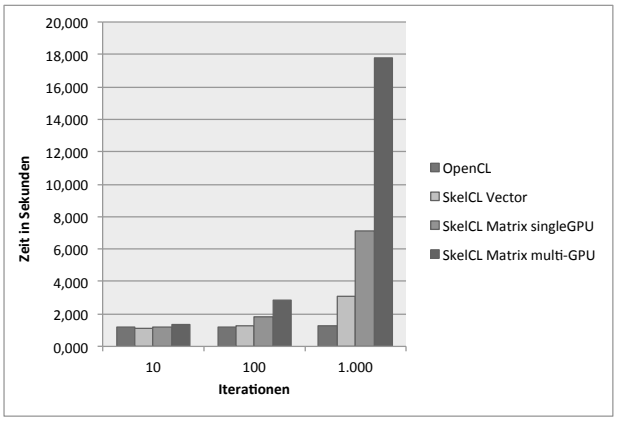
Abbildung 7: Laufzeit der Implementierungen auf System 1

bleiben können bis alle Iterationen durchgeführt sind. Dies scheint jedoch nicht der Fall gewesen zu sein, da die Laufzeiten der OpenCL Version und der Vektor Version sehr weit auseinanderdriften. Die Matrixversionen sind mit ca. zwei Minuten Laufzeit sehr weit abgeschlagen. Auch scheint die Verwendung von mehreren GPUs, für die Berechnung, keine Geschwindigkeitsvorteile zu bringen, was wahrscheinlich daran liegt, dass das Raster copy Verteilt ist und das komplette Raster in jeder Iteration auf das Device hoch und zurück wieder zum Host geladen werden muss. Bei dem kleineren Raster scheinen die Unterschiede erst bei einer hohen Anzahl an Iterationen ins Gewicht zu fallen, denn bei 10 bzw. bei 100 Iterationen, liegen die Laufzeiten der OpenCL und der SkelCL Vektor Version noch dicht beieinander (Abbildung 7b). Lediglich bei 1000 Iterationen ist die OpenCL Version eine gute Sekunde schneller als die SkelCL Version mit Vektoren, was jedoch nur einem Faktor von etwas mehr als 1,5 entspricht. Die Matrizen Versionen der SkelCL Implementierungen sind extrem langsam und besitzen, bei der multi-GPU Variante, mit 11 Sekunden, eine fast 5 fach so lange Laufzeit wie die OpenCL Version. Wobei deutliche Unterschiede zwischen der SkelCL Matrix Version existieren, welche eine GPU und der die alle verfügbaren GPUs benutzt. Dies ist darauf zurückzuführen, dass bei der single-GPU Version das Raster zwar auch in jeder Iteration auf das Device und wieder zurück geladen werden muss, aber dies nur auf eine GPU geschehen muss und nicht wie in der multi-GPU Version auf alle GPUs. Daher müssten die Datentransfers bei der multi-GPU Version auf diesem System, vier mal so hoch sein, wie bei der single-GPU Version.

Auf dem System 2 treten diese Unterschiede noch drastischer in Erscheinung. Schon bei 100 Iterationen trennt die Laufzeiten der OpenCL Version und der SkelCL Vektor Version, bei dem großen Raster, der Faktor ~7, bei 1.000 ist es ein Faktor >26, die multi-GPU Matrizen Version, ist mit der 84 fachen Laufzeit der OpenCL Version, sehr weit abgeschlagen (Abbildung 8a). Möglicherweise überträgt das System 2 die Daten nicht so



(a) Raster der Größe $512 \times 512 \times 64$



(b) Raster der Größe $128 \times 128 \times 32$

Abbildung 8: Laufzeit der Implementierungen auf System 2

schnell zu den GPUs wie das System 1, was die enormen Laufzeiten der SkelCL Versionen erklären würde. Bei dem kleinen Raster liegen die Werte zwar dichter beieinander und sind bei 10 und bei 100 Iterationen fast identisch, allerdings zeigt sich bei 1.000 Iterationen wieder das selbe Laufzeitverhalten, wie auf System 1, denn die OpenCL Version ist schon fast um den Faktor 2,5 schneller als die SkelCL Vektor Version (Abbildung 8b). Die Laufzeiten der Matrizen Versionen sind um ein Vielfaches höher als bei den anderen Versionen.

Bei der Anzahl der LoC (*Lines of Code*), also der Länge des Quelltextes, zeigt SkelCL einen Vorteil. Die SkelCL Versionen sind nur etwa halb so lang wie die OpenCL Version, was darauf hindeutet, dass der erforderliche Programmieraufwand, bei den SkelCL Versionen, deutlich geringer ist.

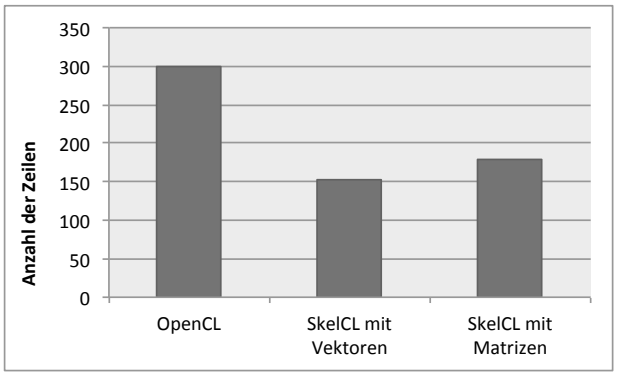


Abbildung 9: Anzahl der Codezeilen

Die Ergebnisse lassen darauf schließen, dass SkelCL für diese Art von Berechnung noch nicht geeignet ist, da dieses Ersparnis von der Hälfte des Quelltextes nicht eine Laufzeiterhöhung von bis zu dem 84 Fachen aufwiegt. Dies könnte zum einen an der nicht vorhandenen Unterstützung für dreidimensionale Datentypen liegen oder an der iterativen Berechnung die vorgenommen wird. SkelCL bietet in der jetzigen Version für die 3D Jacobi Stencil Berechnung nicht die erforderlichen Datentypen und auch nicht das erforderliche algorithmische Skelett, um diese Berechnung mit einem geringen Implementierungsaufwand und einer hohen Performance umzusetzen.

4. B+Baum Traversierung

B Bäume [2] sind Datenstrukturen, die erstmals 1970 von Rudolf Bayer vorgestellt wurden. Sie wurden mit dem Ziel entwickelt die Verwaltung von Plattenspeicher zu beschleunigen und finden sich heutzutage hauptsächlich in Datenbanken und Dateisystemen wieder. Da sie immer vollständig balanciert sind und nicht ausarten können wie z.B. Binärbäume (die zu Listen werden können), liegen alle Suchanfragen in B-Bäumen in der selben Zeitkomplexität. Außerdem speichert er Daten (sog. *Records*), sortiert nach Schlüsseln. Die formale Definition für einen B Baum nach [2] sieht wie folgt aus:

Definition 1 (B Bäume) *Sei $h \geq 0 \in \mathbb{Z}$, $k \in \mathbb{N}$. Ein gerichteter Baum T liegt in der Klasse $\tau(k, h)$ der B Bäume, wenn T entweder leer ist ($k = 0$) oder die folgenden Eigenschaften besitzt.*

1. *Jeder Pfad von der Wurzel zu einem der Blätter besitzt die selbe Länge h, die auch die Höhe des Baumes T genannt wird.*

2. *Jeder Knoten, außer der Wurzel und den Blättern, besitzt mindestens $k+1$ Söhne. Die Wurzel ist entweder ein Blatt oder besitzt mindestens zwei Söhne.*

3. *Jeder Knoten besitzt maximal $2k+1$ Söhne.*

Der in der Definition genannte Parameter k, wird als die Ordnung des Baumes bezeichnet. Normalerweise zeigen die Schlüssel, in den Knoten von B Bäumen, auf Daten wodurch die B Bäume als Suchbäume für diese fungieren. Die vollständige Balancierung eines B Baumes wird durch entsprechend definierte Einfüge- und Löschoperationen sichergestellt. Die Balancierung gewährleistet, dass jede Suchanfrage in $\mathcal{O}(\log(h))$ durchgeführt werden kann, da die Knoten maximal bis zu den Blättern durchsucht werden, die genau h weit von der Wurzel entfernt liegen. In Abbildung 10 ist eine Instanz eines B Baumes mit der Ordnung $k = 2$ dargestellt. Hier ist zu sehen, dass die Knoten auf mindestens $k+1$ und maximal $2k+1$ Söhne verweisen, wobei dies nicht auf die Wurzel und auf die Blätter zutrifft.

B+Bäume sind spezielle B Bäume bei denen im Gegensatz zu den B Bäumen, die Verweise auf die Records nur in den Blättern und nicht zusätzlich in den Knoten vorhanden sind. Dadurch wird innerhalb der Knoten mehr Platz für Schlüssel verfügbar, welcher dazu genutzt werden kann die Ordnung des Baumes zu erhöhen, was einen positiven Effekt auf die Zugriffszeit der Records bringen kann. Außerdem ist es einfach möglich die Records in einem flachen Speicher zu organisieren und Bereichssuchen durchzuführen, also nach Schlüsseln zu suchen, die zwischen zwei Werten liegen, was der Grund dafür ist, dass sie sich sehr gut für die Implementierung von Datenbanken eignen. Die Suchanfragen in B+Bäumen besitzen immer den gleichen zeitlichen Aufwand $\Theta(\log(h))$, da jeder B+Baum, beim Suchen, bis zu den Blättern durchlaufen werden muss.

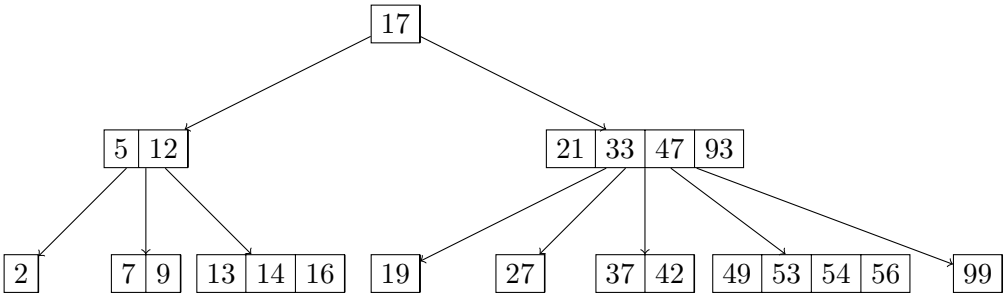


Abbildung 10: Beispiel eines B Baumes der Ordnung 2

4.1. Implementierung

Die Implementierung des B+Baumes stammt von Amittai F. Aviram [1] und wurde verwendet, um wie in [7] vorgestellt, die B+Baum Traversierung zu parallelisieren. Der Grundgedanke hinter der Parallelisierung der B+Baum Traversierung ist, Suchanfragen zu bündeln und so nach mehreren Schlüsseln parallel zu suchen.

Der B+Baum in der zugrundeliegenden Implementierung besteht aus mehreren `nodes` (Listing 11, Zeilen 1 bis 8). Jeder `node` kann für einen Knoten oder für ein Blatt stehen, was durch ein Flag (`is_leaf`) unterschieden wird. Der Unterschied zwischen einem Knoten und einem Blatt liegt in dem Zeigerarray `pointers`. Wenn es sich bei dem `node` um einen Knoten handelt, verweisen die Zeiger auf weitere `nodes`, wobei der erste Zeiger auf ein `node` verweist, dessen Schlüssel (`keys`) alle kleiner sind, als der erste Schlüssel im verweisenden Knoten. Der zweite Zeiger zeigt auf ein `node`, bei dem die Schlüssel alle größer sind als der erste Schlüssel des Verweisenden `nodes` aber kleiner gleich dem zweiten sind. Die Anzahl der Schlüssel in einem Knoten wird durch `num_keys` angegeben, wobei die Anzahl der gültigen Zeiger in einem Knoten immer `num_keys`+1 entspricht. Handelt es sich bei dem `node` um ein Blatt, entsprechen die Indizes der Schlüssel den Indizes der Zeiger, sie bilden also Paare und diese Zeiger verweisen auf die Daten (hier durch `records` repräsentiert), welche über die Schlüssel erreicht werden. Der letzte Zeiger in jedem Blatt (also der an Stelle $k-1$), zeigt auf das Blatt, welches sich rechts neben dem ausgehenden Blatt befindet. Ist dieses Blatt bereits das am weitesten rechts stehende, verweist der Zeiger auf `NULL`. Insbesondere ist das Array von Zeigern, in Blättern, immer von der selben Größe wie das Array von Schlüsseln. Des Weiteren zeigt jeder `node` auf seinen Vorgänger`node`. Die `records` bestehen aus einem integer Wert, allerdings ist es einfach möglich in den `records` beliebige Daten zu halten. Um den Baum nun mittels der GPU zu durchsuchen, wird hierfür die Baumstruktur, die durch die Verkettung der einzelnen `nodes` entstanden ist, in einen zusammenhängenden Speicherbereich transformiert. Dafür existiert die Struktur `knode` (Listing 11, Zeilen 13 bis 19), von der *Anzahl der nodes*-viele, in einen zusammenhängenden Speicherbereich angelegt werden. Der Unterschied zu den `nodes` ist, dass die Größe der `knodes` immer gleich ist, da jeweils für die Schlüssel und Zeiger, die in diesem Fall die Indizes der `knodes` sind auf die gezeigt wird, die maximale mögliche Größe (`DEFAULT_ORDER`+1) alloziert (reserviert)

```
1  typedef struct node {
2          void ** pointers;
3          int * keys;
4          struct node * parent;
5          bool is_leaf;
6          int num_keys;
7          struct node * next; // Used for queue.
8  } node;
9
10 typedef struct record {
11         int value;
12 } record;
13 typedef struct knode {
14         int location;
15         int indices [DEFAULT_ORDER + 1];
16         int  keys [DEFAULT_ORDER + 1];
17         bool is_leaf;
18         int num_keys;
19 } knode;
```

Listing 11: Datenstrukturen auf denen die B+Baum Traversierung arbeitet

wird. Diese Transformation der Daten kostet Zeit und daher eignet sich diese Methode der Parallelisierung, auf der GPU, nur für gebündelte Suchanfragen [7]. Außerdem sollte der B+Baum nicht zu oft verändert werden, denn nach jeder Änderung muss er wieder in die zusammenhängende Datenstruktur transformiert werden.

4.2. OpenCL Implementierung

Für die Traversierung wird zuerst der B+Baum in die vorher beschriebene Kernelstruktur transformiert, woraufhin OpenCL initialisiert wird. Im Folgenden wird eine GPU als Device ausgewählt, weswegen die vorliegende B+Baum Traversierung nicht multi-GPU fähig ist. Nachdem die Command Queue erstellt wurde, wird das Program erstellt und anschließend kompiliert. Für die Argumente des Kernels werden Speicherbereiche reserviert, die als Kernelargumente gesetzt werden.

Bevor der Kernel ausgeführt werden kann, müssen die Anzahl der zu startenden Work-Items, ihre Anordnung, sowie die Größe der Work-Groups und deren Anordnung, festgelegt werden. In der Implementierung wird, für jeden Schlüssel nachdem der Baum durchsucht wird, eine Work-Group gestartet, die für jeden Schlüssel eines `nodes`, ein Work-Item startet. Sei *count* die Anzahl der Schlüssel, nach denen der Baum durchsucht werden soll und weiterhin *order* die Ordnung des B+Baumes, dann werden genau $count * order$ Work-Items gestartet. Die Work-Groups werden eindimensional angeordnet und fassen jeweils *order*-viele Work-Items zusammen. Daraus ergibt sich des Weiteren, dass genau *count*-viele Work-Groups gestartet werden (eine pro zu Schlüssel).

Der Kernel ist in Listing 12 zu sehen. Es werden zunächst die globale und die lokale ID ermittelt, mit denen die Position des Work-Items und die Zuordnung des Work-Items zu seiner Work-Group bestimmt werden (Zeile 7 und 8). Die Traversierung startet in Zeile 10 mit dem Beginn der Schleife, die dazu dient die `nodes` der Höhe nach zu durchlaufen. Innerhalb der Schleife wird der gesuchte Schlüssel, mit jeweils zwei benachbarten Schlüsseln des `nodes`, welcher gerade durchsucht wird, verglichen. Hier ist zu beachten, dass dieser Vergleich (Zeile 12) von allen Work-Items einer Work-Group parallel durchgeführt wird. Ist eine Stelle gefunden, an der die Abfrage zutrifft, wird geprüft ob in der drunterliegenden Ebene des B+Baumes ein weiterer `node` liegt, indem gesucht werden muss (Zeile 14). Diese Abfrage wird von genau einem Work-Item einer Work-Group durchgeführt, da die vorherige Abfrage nur auf ein Nachbarpaar von Schlüsseln zutreffen kann. Damit alle Work-Items einer Work-Group die selben Informationen darüber besitzen, an welcher Stelle dieses Nachbarpaar gefunden wurde, wird eine Synchronisation durchgeführt (Zeile 18). Es wird sichergestellt, dass lediglich das erste Work-Item einer Work-Group, auf den gemeinsamen Speicher zugreift (Zeile 18), da es sonst zu race-conditions kommen könnte.

Nachdem die Höhe des Baumes durchlaufen wurde, existiert pro Work-Group ein `knode` indem sich der Schlüssel befinden könnte. Dies wird wieder parallel von allen Work-Items einer Work-Group durchsucht (Zeile 26). Wird der Schlüssel gefunden, so wird in den Ausgabespeicher `ansD` geschrieben (Zeile 27). Diese Schreiboperation führt immer nur ein Work-Item einer Work-Group aus, da der Schlüssel an nur einer Stelle im `knode`, also von einem Work-Item, gefunden werden kann. Wurde der Schlüssel nicht gefunden, wird kein Ergebnis geschrieben.

4.3. SkelCL Implementierung

Ein wichtiger Unterschied zwischen der SkelCL Implementierung und der OpenCL Implementierung besteht darin, dass pro Schlüssel, nach dem gesucht werden soll, lediglich ein Work-Item gestartet wird, im Gegensatz zu k Work-Items bei der OpenCL Implementierung. Dies liegt daran, dass es in SkelCL zur Zeit noch nicht möglich ist, die ID der Work-Groups zu bestimmen, wenn mehrere Devices zur Berechnung genutzt werden. Auf jedem Device, werden die Work-Groups unabhängig voneinander nummeriert, d.h. es ist nicht möglich, über die lokale oder globale ID, die erste Work-Group auf einem Device von der ersten Work-Group auf einem anderen Device zu unterscheiden. Allerdings ist diese Einschränkung, für die Traversierung eines B+Baumes, nicht weiter hinderlich, es entstehen sogar, wie wir später sehen werden, durch die Lösung, nur ein Work-Item pro Schlüssel zu starten, Vorteile. Innerhalb der Funktion `kernel_gpu_skelcl_wrapper(...)` wird zuerst die Initialisierung von SkelCL durchgeführt, wobei als Devices, für die weitere Benutzung, alle verfügbaren GPUs ausgewählt werden (Zeilen 7 bis 9). Danach werden die Eingabedaten, die in der OpenCL Version direkt dem Kernel übergeben wurden, in SkelCL Vektoren gespeichert (Zeilen 11 bis 15). Daraufhin werden die Verteilungen der Eingabedaten festgelegt. Bei den Eingabedaten, d.h. den Daten die dem Kernel übergeben werden, handelt es sich zum einen um den transformierten B+Baum (`vecKnodes`),

```
1   __kernel void findK(long height, __global knode *knodesD,
2                       long knodes_elem, __global record *recordsD,
3                       __global long *currKnodeD, __global long *offsetD,
4                       __global int *keysD, __global record *ansD)
5   {
6   // private thread IDs
7   int thid = get_local_id(0);
8   int bid = get_group_id(0);
9   // processtree levels
10  for(int i = 0; i < height; i++){
11    // if value is between the two keys
12    if( (knodesD[currKnodeD[bid]].keys[thid]) <= keysD[bid]
13         && (knodesD[currKnodeD[bid]].keys[thid+1] > keysD[bid])){
14      if(knodesD[offsetD[bid]].indices[thid] < knodes_elem){
15        offsetD[bid] = knodesD[offsetD[bid]].indices[thid];
16      }
17    }
18   __syncthreads();
19    // set for next tree level
20    if(thid==0)
21      currKnodeD[bid] = offsetD[bid];
22    __syncthreads();
23  }
24  //At this point, we have a candidate leaf node which may contain
25  //the target record.  Check each key to hopefully find the record
26  if(knodesD[currKnodeD[bid]].keys[thid] == keysD[bid]){
27    ansD[bid].value =
28                recordsD[knodesD[currKnodeD[bid]].indices[thid]].value;
29  }
30  }
```

Listing 12: Kernelcode der OpenCL B+Baum Traversierung

um die `records` welche die Daten repräsentieren auf welche die Schlüssel verweisen (`vecRecords`) und den Schlüsseln, nachdem der Baum durchsucht werden soll (`vecKeys`). Der Vektor `vecKeys` wird, wie in Zeile 17 zu sehen, block verteilt, da die einzelnen Suchen (nach den verschiedenen Schlüsseln) unabhängig voneinander sind, sie werden also auf die zuvor ausgewählten Devices aufgeteilt. Da auf jedem Device der komplette Baum und die `records` verfügbar sein müssen, werden diese beiden Vektoren (`vecKnodes` und `vecRecords`) copy verteilt (Zeilen 18 und 19). Nun ist die Verteilung der Daten festgelegt, es fehlt allerdings noch die Wahl des passenden Skeletts um die Traversierung durchzuführen. Hierfür wurde, wie in Zeile 21 zu sehen ist, das Map Skelett gewählt, da die Traversierung eine Funktion darstellt, die auf jeden Schlüssel angewandt wird. In Zeile 23 wird der Kernel ausgeführt, wobei eine neuer Vektor mit den Ausgabedaten (`vecAns`) zurückgegeben wird. Die Ausgabedaten werden in den Zeilen 26 und 27 in das Array `ans` kopiert.

Der Kernel der SkelCL Version (Listing 14) sieht dem Kernel der OpenCL Version sehr

```
1  using namespace skelcl;
2  void kernel_gpu_skelcl_wrapper(record *records, long records_elem,
3          knode *knodes, long knodes_elem, int order, long maxheight,
4          int count, int *keys, record *ans)
5  {
6    //Waehle alle GPUs
7    detail::DeviceProperties d=allDevices();
8    d.deviceType(device_type::GPU);
9    skelcl::init(d);
10   //Kopiere Daten in SkelCL Vektoren
11   Vector<record> vecRecords(records, records + records_elem);
12   Vector<knode> vecKnodes(knodes, knodes + knodes_elem);
13   Vector<int> vecKeys(count);
14   for(int i=099;i<count;i++)
15     vecKeys[i]=keys[i];
16   //Setze die Verteilungen
17   vecKeys.setDistribution(distribution::Block(vecKeys));
18   vecRecords.setDistribution(distribution::Copy(vecRecords));
19   vecKnodes.setDistribution(distribution::Copy(vecKnodes));
20   //Erstelle Berechnung mittels Map Skelett
21   Map<record(int)> trav(std::ifstream("./kernel/kernel_gpu_skelcl.cl"));
22   //Fuehre die Traversierung aus
23   Vector<record> vecAns = trav(vecKeys, maxheight, order,
24                                vecKnodes, knodes_elem, vecRecords);
25   //Kopiere Ergebnisse in Ausgabespeicher
26   for(unsigned int i=0; i<vecAns.size(); i++)
27     ans[i].value=vecAns[i].value;
28 }
```

Listing 13: Ausschnitt des Hostcodes der SkelCL B+Baum Traversierung

ähnlich, da das Grundprinzip der beiden Kernel das Selbe ist. Zunächst fällt auf, dass das Ausgabedatum (`ans`) im Kernel erzeugt und initialisiert wird. Außerdem fällt auf, dass zwar der weitere Kernel die gleichen Operationen darstellt wie jener der OpenCL Version, die Synchronisierung allerdings nicht vorhanden ist. Da in der SkelCL Version für jeden Schlüssel nur ein Work-Item gestartet wird, werden nicht mehr alle Nachbarpaare in den `knodes` mit dem Schlüssel, nach dem gesucht wird, parallel verglichen. Dies führt zwar zu einem niedrigeren Parallelitätsgrad als bei der OpenCL Version, allerdings kann es nicht mehr vorkommen, dass zwei Work-Items in die selbe Speicherzelle schreiben wollen und somit ist keine Synchronisation zwischen den Work-Items erforderlich. Ein Vorteil der daraus entsteht, ist dass die B+Baum Traversierung damit multi-Device fähig wird, da der Kernel nun nicht mehr in Abhängigkeit von der Work-Group ID arbeitet, wie es in der OpenCL Version der Fall ist (Listing 12, Zeile 7). Der übrige Kernelcode ist ansonsten äquivalent zu dem der OpenCL Version.

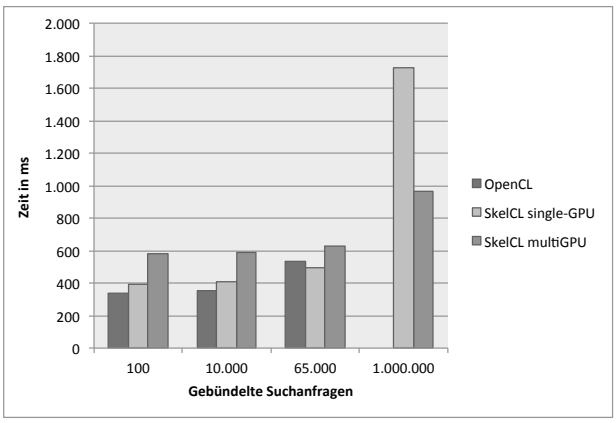
```
record func(int key, long height, int order, global knode *knodes,
            long knodes_elem, global record *records){
  long currKnode = 0;
  long offset = 0;
  record ans;
  ans.value = -1;
  for(int i=0; i<height; i++){
    for(int thid=0; thid < order; thid++)
      if(knodes[currKnode].keys[thid] <= key
         && knodes[currKnode].keys[thid+1] > key)
                if(knodes[offset].indices[thid] < knodes_elem)
                    offset = knodes[offset].indices[thid];
      currKnode=offset;
  }
  for(int thid=0; thid<order; thid++)
    if(knodes[currKnode].keys[thid] == key)
      ans.value = records[knodes[currKnode].indices[thid]].value;
  return ans;
}
```

Listing 14: Kernelcode der SkelCL B+Baum Traversierung

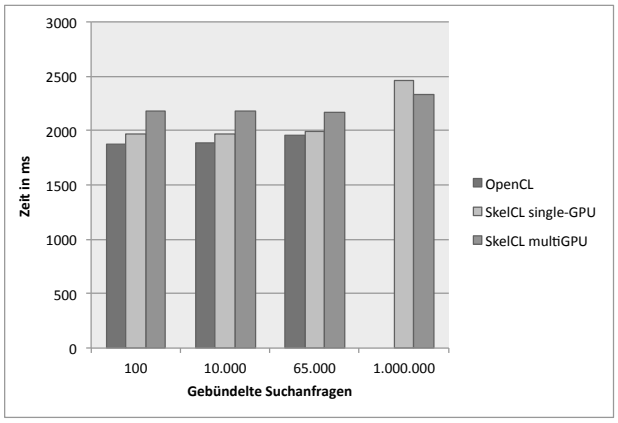
4.4. Evaluation

Die Testumgebungen für die Erhebung der Daten sehen wieder wie folgt aus. Zum einen wurde ein System mit einer 4 Kern Intel Xeon E5520 CPU mit 2,27GHz verwendet welche zudem über 4 NVIDIA Tesla C1060 GPUs verfügt (*System 1*). Zum anderen wurde ein System mit einem 2 Kern Intel Core 2 6600 mit 2,4GHz CPU und zwei NVIDIA GeForce 9800 GX2 GPUs sowie einer NVIDIA GeForce GTX 480 GPU verwendet (*System 2*).
Es wurden auf beiden Systemen B+Bäume mit jeweils 10 Millionen Records traversiert, wobei als Ordnungen 508, der Standardwert der OpenCL Implementierung, sowie 128, der Wert, der laut [7] die beste Performance liefert, benutzt wurden. Außerdem wurde die Anzahl der Elemente in einer Suchanfrage variiert, wobei hier zu beachten ist, dass die OpenCL Implementierung nach maximal $2^{16} - 1 = 65535$ Elementen gleichzeitig suchen kann, da sie für jedes Suchelement eine Work-Group startet und die maximale Anzahl der Work-Groups auf den Testsystemen bei $2^{16} - 1$ liegt. Als Größen für die Suchanfragen wurden 100, 10.000, 65.000 und 1.000.000 gewählt. Bei der SkelCL Version wurde zum einen nur eine GPU und zum anderen alle verfügbaren GPUs verwendet.

In Abbildung 11 ist die Laufzeit der Implementierungen auf System 1 dargestellt, auf der linken Seite mit einem B+Baum der Ordnung 508 und auf der rechten mit einem B+Baum der Ordnung 128. Es ist deutlich zu erkennen, dass die Zeiten, in Abbildung 11a, der OpenCL und der SkelCL Version, welche nur eine GPU verwendet, sehr nahe beieinander liegen. Die multi-GPU, SkelCL Version ist bei wenigen Suchanfragen etwas langsamer als die beiden anderen Versionen, wobei sie sich, mit steigender Anzahl der gebündelten Suchanfragen, den Laufzeiten der anderen Versionen annähert. Dies



(a) B+Baum mit Ordnung 508 und 10Mio. Records

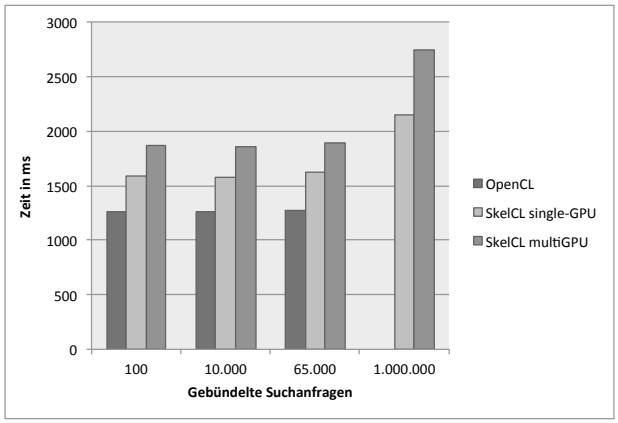


(b) B+Baum mit Ordnung 128 und 10Mio. Records

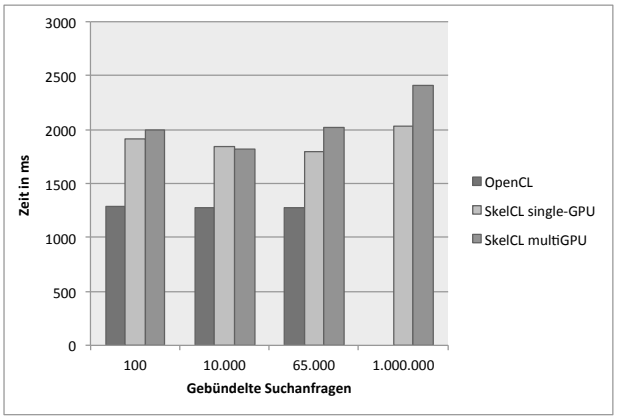
Abbildung 11: Laufzeit der Implementierungen auf System 1

liegt vermutlich daran, dass für die Verteilung der Daten auf die verschiedenen GPUs zusätzlicher Aufwand (*Overhead*) betrieben werden muss, der bei wenigen Suchanfragen nicht durch den Geschwindigkeitsvorteil, der mehreren GPUs, ausgeglichen werden kann. Allerdings zeigt sich bei der Zeit für 1 Million Suchanfragen, dass die multi-GPU Variante fast doppelt so schnell ist, wie die Version mit einer GPU. Hier ist der Overhead für die multi-GPU Fähigkeit, im Vergleich zur Zeit die benötigt wird um die Suchanfragen abzuarbeiten, sehr gering und fällt daher bei vielen Suchanfragen nicht mehr so stark ins Gewicht. Auch zu erkennen ist, dass die Wahl der Ordnung des B+Baumes einen großen Unterschied der Laufzeit zufolge haben kann, denn obwohl die Ordnung 128 nach [7] die beste Performance erzielen soll, ist dies auf den Testsystemen der gegenteilige Fall. Hier ist in Abbildung 11b zu sehen, dass die Implementierungen alle eine längere Laufzeit besitzen, als die mit der Ordnung 508. Es sind sogar alle Laufzeiten, bei einer Ordnung von 128, langsamer als bei einem Baum mit der Ordnung 508, wobei die Laufzeit auf den Bäumen mit Ordnung 128 annähernd konstant zu sein scheint. Dies scheint darauf zurück zu führen zu sein, dass durch die geringere Ordnung von 128 erheblicher Mehraufwand betrieben werden muss, welcher die Laufzeitvorteile überwiegt. Außerdem ist zu erkennen, dass bei einer gebündelten Suchanfrage von 1 Million Elementen, die beiden SkelCL Versionen eine ähnliche Laufzeit besitzen, was nicht intuitiv erscheint.

Auf dem System 2 sehen die Laufzeiten der unterschiedlichen Versionen und Aufrufe, denen auf System 1 sehr ähnlich. Alle Laufzeiten sind um einen konstanten Faktor höher, was vermutlich drauf zurückzuführen ist, dass zwar die GPUs des System 2 eine höhere maximale Performance liefern, als die des System 1, aber die CPU, welche die Daten auf die Devices verteilt, auf dem System 1 eine höhere maximale Performance besitzt als die des System 2. Auch ist zu erkennen, dass auf dem System 2, die single-GPU Version schneller ist als die multi-GPU SkelCL Version (Abbildung 12a). Dies könnte daran liegen, dass die verwendete GPU in der single-GPU (GeForce GTX 480) Version



(a) B+Baum mit Ordnung 508 und 10Mio. Records



(b) B+Baum mit Ordnung 128 und 10Mio. Records

Abbildung 12: Laufzeit der Implementierungen auf System 2

schneller ist als die beiden anderen GPUs (GeForce 9800 GX2) des System 2. Wenn nun die Daten in gleich große Teile auf die drei GPUs aufgeteilt werden, wird der Performanceunterschied der verwendeten GPUs nicht berücksichtigt, woraufhin die erste GPU schneller mit ihren Berechnungen fertig ist als die beiden anderen und daher, bis die anderen GPUs mit ihren Berechnungen fertig sind, wartet.

In Abbildung 13 ist die Anzahl der LoC (*Lines of Code*, Codezeilen) der beiden Implementierungen gegenübergestellt. Hier lässt sich gut erkennen, welchen Vorteil SkelCL bei der Implementierung von parallelen Anwendungen bietet. Die OpenCL Implementierung wurde in 430 LoC geschrieben, bei der SkelCL Version wurden nur 70 LoC benötigt. Dies bringt auf der einen Seite einen zeitlichen Vorteil bei der Implementierung selber, denn 70 Zeilen Code lassen sich einfach schneller schreiben als 430. Auf der anderen Seite ist ein kürzerer Code besser les- und dadurch auch wartbar. Die Anzahl der LoC

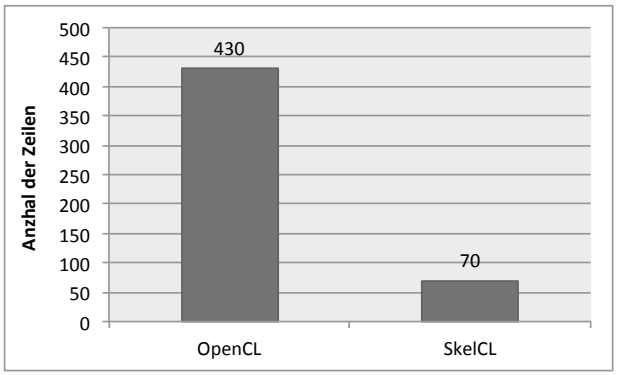


Abbildung 13: Anzahl der Codezeilen

ist natürlich nur ein ungefähres Maß für den Programmieraufwand, denn sie setzt sich aus mehreren Faktoren zusammen, wie etwa aus der Formatierung des Codes oder den Fähigkeiten des Programmierers. Allerdings ist der Unterschied in der Anzahl der LoC so groß (ca. Faktor 6), dass sich hierdurch schon Rückschlüsse auf den benötigten Programmieraufwand ziehen lassen. Der kürzere SkelCL Quelltext erklärt sich zum einen durch die Initialisierung von OpenCL, welche SkelCL dem Programmierer vollständig abnimmt und zum anderen durch die Verwendung der algorithmischen Skelette, denen die eigentliche Berechnung übergeben wird. Zu bemerken ist außerdem, dass bei der B+Baum Traversierung, die multi-GPU Fähigkeit der SkelCL Version keinen Mehraufwand bei der Programmierung erforderte und somit auch die Anzahl der LoC nicht beeinflusste, es mussten lediglich die Verteilungen der verschiedenen Daten angepasst werden.

5. Fazit

Das Fazit zur Implementierung mit SkelCL sieht zweigeteilt aus. Auf der einen Seite ist SkelCL bei der Implementierung von parallelen Anwendungen sehr hilfreich, denn durch die Verwendung von algorithmischen Skeletten, muss der Programmierer sich nicht mehr überlegen, wie die Daten auf die Devices verteilt und die Berechnung parallelisiert werden, sondern kann seine Algorithmen leicht in die vorgegebenen Skelette einordnen. Dafür muss zwar ein Overhead in kauf genommen werden, der je nach Algorithmus auch hoch sein kann, allerdings kann das Zeitersparnis beim Implementieren diesen Overhead wieder aufwiegen. Bei den vorgestellten Anwendungen konnten die SkelCL Implementierungen mit bis zu 6 mal weniger Quelltext, als die äquivalenten OpenCL Implementierungen, geschrieben werden (siehe B+Baum Traversierung). Besonders bei der Entwicklung von multi-Device fähigen, parallelen Algorithmen kann SkelCL seine Stärken ausspielen, da man hier mit OpenCL die Daten manuell auf die Devices verteilen müsste, was sehr aufwendig ist. In SkelCL muss man hingegen nur eine geeignete Verteilung wählen und hat somit sein Ziel schon erreicht. Dies hat auch zur Folge, dass die Quelltexte von SkelCL Programmen viel kürzer sind als die von OpenCL Anwendungen, wodurch der Code auch besser les- und wartbar wird.
Auf der anderen Seite gibt es aber auch Berechnungen, die von der Verwendung von SkelCL (noch) nicht profitieren, wie bei der 3D Jacobi Stencil Berechnung gesehen wurde. Dies liegt unter anderem daran, dass SkelCL nicht über dreidimensionale Datentypen verfügt, mit denen die Daten auf die Devices verteilt werden können. SkelCL bietet ein großes Potential und stellt eine einfach zu benutzende und überwiegend performante Alternative zur nativen OpenCL Programmierung dar.

Im Folgenden werden einige mögliche Erweiterungen für SkelCL beschrieben, welche die Implementierung und Ausführung von SkelCL Programmen möglicherweise noch weiter erleichtern bzw. verbessern würde. Zum einen wäre eine dreidimensionale Datenstruktur, wie sie in der 3D Jacobi Stencil Berechnung gebraucht werden würde, eine vorteilhafte Erweiterung für SkelCL, bei der man ähnlich wie bei einer Matrix, die man als weiteres Argument an den Kernel übergibt, auf die Elemente mittels der Koordinaten zugreifen kann. Damit wäre es nicht mehr nötig Daten mit einer dreidimensionalen Struktur, in eine zwei- oder gar eindimensionale zu transformieren, sondern könnte intuitiver mit ihnen arbeiten. Außerdem könnte das MapOverlap Skelett, welches auf Matrizen, also auf zweidimensionalen Strukturen definiert ist, auf dreidimensionale Strukturen erweitert werden. Gerade bei der Verteilung der Daten auf die Devices ist es bei dreidimensionalen Datenstrukturen nicht einfach zu bestimmen, welche Elemente auf welchen Devices verfügbar sein müssen, da man umdenken muss, wenn dreidimensional angeordnete Daten in einer zweidimensionalen Datenstruktur abgelegt sind. Besonders die Stencil Codes, die in der Physik oft gebraucht werden, könnten durch die Erweiterung von MapOverlap auf dreidimensionale Datentypen profitieren.

Durch die Datentypen `IndexVector` bzw. `IndexMatrix`, werden die globalen IDs aus OpenCL für den Fall erweitert, dass die Daten auf mehrere Devices verteilt werden

sollen, was ein sehr nützliches Werkzeug bei der SkelCL Programmierung darstellt. In OpenCL werden die globalen IDs für jedes Device neu nummeriert, was eine Unterscheidung der Devices anhand der ID unmöglich macht. Jedoch wäre es zum Teil hilfreich, wenn dieser Mechanismus auch auf die Work-Group IDs bzw. auf die lokalen IDs übertragen werden würde. Dies würde die Übertragung von Alogorithmen und Anwendungen vereinfachen, die ursprünglich in oder für OpenCL entworfen wurden, aber in SkelCL implementiert werden sollen.

OpenCL und SkelCL bieten die Möglichkeit parallele Anwendungen für heterogene Systeme zu Programmieren, allerdings sind in heterogenen Systemen die Geschwindigkeiten der verfügbaren Devices meist unterschiedlich. Daher wäre es hilfreich, wenn SkelCL automatisch die Verteilung Block anhand der Leistungsfähigkeit der unterschiedlichen Devices vornehmen könnte. Dies ist zwar nicht einfach umsetzbar, da man aus den Device Informationen nicht immer ableiten kann, wie schnell bzw. wie viel schneller ein Device im Gegensatz zu einem anderen ist, allerdings wäre es von großem Vorteil, wenn bei einer Berechnung, bei der die Daten auf mehrere GPUs Block verteilt werden, die Daten in unterschiedlich große Teile geteilt werden, je nachdem wie schnell die verfügbaren GPUs sind. Dadurch würde verhindert werden, dass eine GPU die viel schneller ist als eine andere GPU des Systems, auch viel früher mit ihren Berechnungen fertig ist und auf die andere GPU wartet, wodurch wertvolle Rechenzeit der wartenden GPU verschwendet werden würde.

Literatur

[1] A. Aviram. Original B+ Tree source: Date: 26 June 2010, http://www.amittai.com/prose/bplustree.html.

[2] R. Bayer and E.M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[3] OpenMP Architecture Review Board. Openmp application program interface OpenMP Application Program Interface Version 3.1, July 2011.

[4] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44 –54, oct. 2009.

[5] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1 –11, dec. 2010.

[6] Murray I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[7] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating Braided B+Tree Searches on a GPU With Cuda. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011.

[8] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[9] Intel. Intel Core i7-3700 Desktop Processor Series. Technical report.

[10] A. Munshi et al. The OpenCL Specification. *Khronos OpenCL Working Group*, 2009.

[11] Nvidia. Fermi Compute Architecture Whitepaper.

[12] Nvidia. NVIDIA GeForce GTX 680 Whitepaper.

[13] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176 –1182, may 2011.

[14] M. Steuwer, P. Kegel, and S. Gorlatch. Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1858 –1865, may 2012.

[15] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, and W.W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

A. Quelltexte

Die in dieser Bachelorarbeit referenzierten Quelltexte sind auf der unten eingeklebten CD zu finden. Sie liegen im Verzeichnis `skelcl/examples/jacobi_stencil` bzw. `skelcl/examples/b+tree`. Die vollständige SkelCL Bibliothek ist im Ordner `skelcl` zu finden. Außerdem ist die Bachelorarbeit als PDF im Wurzelverzeichnis auf der CD gespeichert.

Hiermit versichere ich, dass die vorliegende Arbeit mit dem Thema, Performanceanalyse von SkelCL mittels B+Baum Traversierung und 3D Jacobi Stencil Berechnung, selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, den 8. März 2013

Patrick Schiffler

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Münster, den 8. März 2013

Patrick Schiffler