TH Nürnberg Georg Simon Ohm
Fakultät Informatik

Masterarbeit

Entwicklung einer OpenCL-Implementierung für die VideoCore IV GPU des Raspberry Pi

---

Autor	Daniel Stadelmann
Matrikelnummer	2427423
Abgabe am	01.11.2017
Betreuer	Prof. Dr. Ch. Schiedermeier
Zweitkorrektor	Prof. Dr. M. Teßmann

Inhaltsverzeichnis

1. Einleitung

Raspberry Pi ist eine Serie von kleinen Einplatinencomputern, die vor allem für schulische und experimentelle Zwecke ins Leben gerufen wurde. Eden Upton, einer der Mitgründer der Raspberry Pi Foundation, beantwortete 2012 auf einer Maker Fair die Frage, ob es eine Möglichkeit geben wird, auf die (im Gegensatz zu der CPU sehr viel leistungsstärkeren) GPU für nicht-grafische Berechnungen zugreifen zu können, mit der Antwort „dass es nicht OpenCL sein wird [...], aber es eine Möglichkeit geben wird, Code auf der GPU auszuführen" [Upt12, Video Minute 21:10]. Zusätzlich hat Broadcom, der Hersteller der auf den Raspberry Pi-Modellen verbauten VideoCore IV GPU, 2014 deren Dokumentation veröffentlicht [Upt14], woraufhin eine Vielzahl von Projekten entstanden sind, mit dem Ziel, allgemeine Berechnungen auf der GPU auszuführen. Diese basieren jedoch meistens auf Assembler oder sehr spezielle Anwendungsfällen. Ebenso stellt Broadcom, im Gegensatz zu anderen Sprachen für die Programmierung von GPUs, wie z. B. OpenGL ES, immer noch keine offizielle Bibliothek für die Unterstützung von OpenCL auf der VideoCore IV GPU bereit. In einem Blog Post schreibt Eric Anholt, der für Broadcom unter anderem am OpenGL Treiber für die VideoCore IV entwickelt, dass „die Hardware nicht wirklich leistungsfähig genug ist, über OpenCL angesprochen werden zu können" [Anh16]. Ebenso schreibt er, dass Überlegungen in diese Richtung durchgeführt wurden, jedoch zu dem Entschluss kamen, dass die Speicheranbindung nicht gut genug dafür ist. Somit bleibt, ohne eine einfache Anbindung zum Ausführen allgemeiner Berechnungen auf der GPU der Raspberry Pi-Modelle, diese vor allem bei eingebetteten Projekten (z. B. Motorensteuerung oder Automationstechnik) meist vollständig ungenutzt.

1.1 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, eine weitestgehend funktionsfähige Implementierung des OpenCL-Standards zu verfassen, die es erlaubt, OpenCL C-Code auf der VideoCore IV GPU der Raspberry Pi-Modelle auszuführen. Diese Implementierung soll mindestens eine auf OpenCL basierende Bibliothek oder Anwendung unterstützen, sodass diese erfolgreich zusammen mit der hier entstehenden OpenCL-Implementierung verwendet werden kann. Weiterhin wird die Performance der OpenCL-Implementierung für die VideoCore IV-Hardware mit der Performance der jeweiligen Host-CPU der Raspberry Pi-Modelle verglichen, um zu zeigen, ob

die Verwendung von OpenCL auf der VideoCore IV GPU einen Nutzen für die Ausführungsdauer von Programme bringt. Durch den Beweis der Funktionalität mithilfe der Unterstützung von auf OpenCL basierenden Anwendungen und den Beweis der Effizienz durch Vergleich von Benchmarkergebnissen mit der Host-CPU der Raspberry Pi-Modelle soll aufgezeigt werden, dass die Implementierung von OpenCL auf der VideoCore IV GPU möglich und deren Verwendung auch effizient ist.

1.2 OpenCL-Standard

„OpenCL (Open Computing Language) ist ein offener Standard für allgemeine parallele Berechnungen auf CPUs, GPUs und weiteren Prozessoren" [Gro12, Kapitel 1]. Einer der Hauptanwendungsbereiche für OpenCL ist das sogenannte GPGPU (General Purpose Computation on Graphic Processor Units), also das Ausführen nicht-grafischer Berechnungen auf Grafikprozessoren. Der Nutzen von GPGPU und somit auch die Motivation hinter OpenCL ist die Eigenschaft moderner Grafikkarten, dass diese zwar meist einen im Vergleich zu CPUs eingeschränkten Befehlssatz besitzen, jedoch sowohl eine deutlich größere Anzahl an Prozessorkernen (z. B. hat die Nvidia GTX 960, erschienen 2015, 1024 Rechenkerne [NVI17]) als auch die Fähigkeit besitzen, mehrere Daten in einer Instruktion zu verarbeiten. Diese Fähigkeit, SIMD (Single Instruction, Multiple Data) genannt, erlaubt es den Prozessoren, einen Befehl auf mehrere Werte parallel auszuführen. Durch die deutlich erhöhte Anzahl an Prozessorkernen und die SIMD-Eigenschaft eignen sich GPUs meist sehr gut für mathematische Berechnungen auf großen Datensätzen. Jedoch ist die Programmierung von Grafikkarten stark plattformabhängig und Schritte, die bei „normalen" für die CPU geschriebenen Programmen einfach zu handhaben sind, wie das Starten von Programmen und der Zugriff auf den Arbeitsspeicher, sind bei der Programmierung für GPUs sehr viel komplizierter. OpenCL bietet eine plattformunabhängige Lösung für die Programmierung von GPUs sowie das Übertragen von Daten zwischen den Grafikprozessoren und Hauptspeicher und das Starten und Steuern von auf Grafikkarten ausgeführten Programmen. Der OpenCL-Standard wird von der Khronos Group entwickelt und ist mitsamt offiziellen Erweiterungen und Programmierreferenzen auf der offiziellen Khronos-Website [Gro17a] zu finden. Die Khronos Group ist ein Industriekonsortium von Industriegrößen wie Nvidia, AMD, Apple, ARM und Intel, die für die Standardisierung von Schnittstellen vor allem im Bereich Multimediaprogrammierung zuständig ist und auch weitere bekannte Standards wie OpenGL, OpenGL ES, Vulkan und SPIR-V veröffentlicht [Gro17e].

Der OpenCL-Standard befindet sich derzeit (Stand: 21.10.2017) in der finalen Version 2.2 [Gro17b] und lässt sich in drei für diese Arbeit relevante Teile gliedern:

1.2.1 OpenCL-Laufzeitbibliothek

Die **OpenCL-Laufzeitbibliothek** ist eine durch C-Header definierte Bibliothek, die in ein Programm, das OpenCL verwendet, eingebunden wird und dient dem Zugriff auf die OpenCL-Implementierung. Sie beinhaltet alle Funktionen, die benötigt werden, um Berechnungen über OpenCL durchzuführen. Da es der OpenCL-Standard erlaubt, mehrere OpenCL-Implementierungen (z. B. zum Ansprechen mehrerer Geräte wie CPU und GPU) gleichzeitig zu verwenden, wird jede Instanz einer OpenCL-Implementierung durch eine **OpenCL-Plattform** repräsentiert. Eine OpenCL-Plattform kann mehrere **OpenCL-Geräte** besitzen, auf denen die Berechnungen durchgeführt werden (siehe Abbildung 1.1). OpenCL-Geräte können weiter in Recheneinheiten und einzelnen Kernen unterteilt werden. Die in dieser Arbeit verwendete VideoCore IV GPU wird aufgrund der geringen Anzahl an Kernen (siehe Abschnitt 1.3) als ein einziges OpenCL-Gerät mit einer Recheneinheit angesehen. Die Interaktion mit dem OpenCL-Gerät (z. B. der GPU) geschieht über **Befehlswarteschlangen** (Command Queues), in denen alle Kommunikation (z. B. Ausführung von Code, Kopieren von Speicherbereichen) eingereiht wird, sodass die OpenCL-Implementierung diese Befehle (**Events**) dann synchron oder asynchron abarbeiten kann und die Kommunikation mit den OpenCL-Geräten synchronisiert werden kann, um so einen threadsicheren Zugriff auf die OpenCL-Plattform zu erlauben. Eine ausführlichere Beschreibung der Plattform-, Ausführungs-, Speicher- und Programmier-Modelle kann im Kapitel 3 der OpenCL 1.2-Spezifikation nachgelesen werden. Die Funktionsspezifikationen für die OpenCL-Laufzeitbibliotheken befinden sich in den Kapiteln 4 und 5 der OpenCL-Spezifikation, deren Implementierung wird im Kapitel 4 dieser Arbeit erläutert. [Gro12, Kapitel 3]

1.2.2 OpenCL C-Programmiersprache

Der zweite Teil des OpenCL-Standards ist die **OpenCL C-Programmiersprache**. OpenCL C basiert auf der C99 Spezifikation und modifiziert diese, um einerseits die Begrenzungen von GPUs darzustellen, andererseits das Programmieren von SIMD-Instruktionen zu vereinfachen. So bietet OpenCL C einen zusätzlichen optionalen Fließkommatypen `half`, der 16-Bit einnimmt und der Datentyp `double` ist auch optional, muss also von einer Implementierung nicht unterstützt werden, falls diese keine 64-Bit Fließkommaarithmetik bietet. Ebenso gibt es für jeden unterstützten skalaren Typ (z. B. `uchar`, `char`) fünf Varianten, die Vektoren dieser Typen darstellen. So gibt es neben `int` noch die Vektortypen `int2`, `int3`, `int4`, `int8` und `int16` mit 2, 3, 4, 8 und 16 Elementen vom Typ `int`. Diese Vektortypen müssen von allen OpenCL-Implementierungen unterstützt werden, auch wenn die Hardware keine nativen Vektor-Operationen bietet, in welchen Fall Instruktionen auf den Vektortypen seriell auf den einzelnen Vektorelementen ausgeführt werden

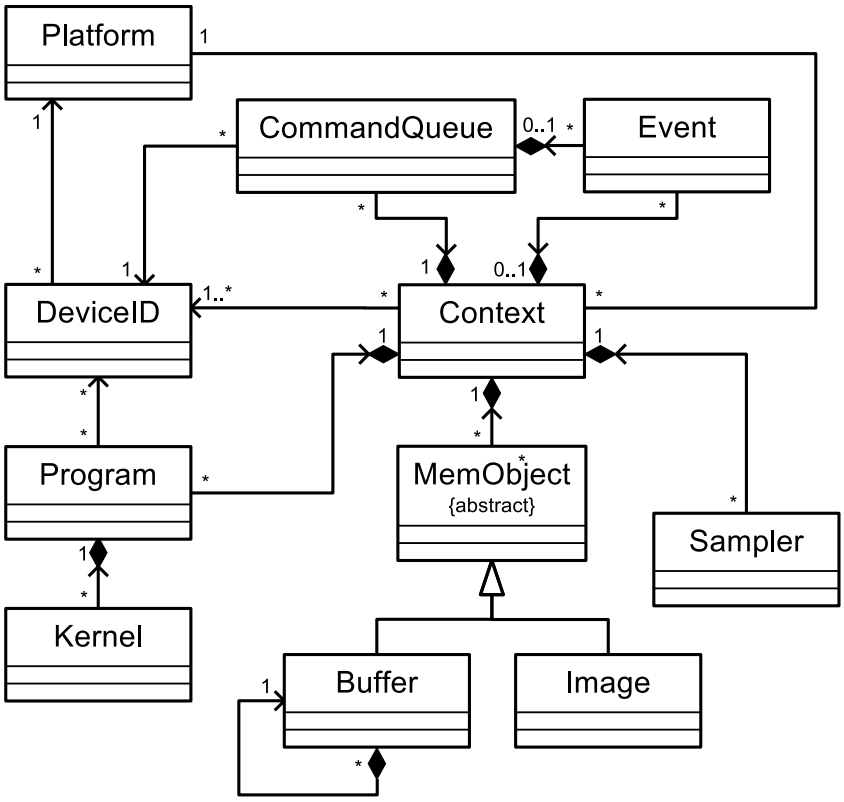

Abbildung 1.1: OpenCL Plattform Modell [Gro12, Abbildung 2.1]

müssen. Sämtliche arithmetischen Operatoren, die der C99 Standard für skalare Typen definiert, sind in OpenCL C auch für die jeweiligen Vektortypen definiert und werden komponentenweise berechnet. Zum Beispiel ist eine Addition `int3 c = a + b` zweier Vektortypen `int3 a, b` semantisch gleichbedeutend mit den Additionen aller Komponenten (siehe Listing 1.1). Ebenso sind arithmetische Ope-

```
1 int3 a, b, c;
2 c.x = a.x + b.x;
3 c.y = a.y + b.y;
4 c.z = a.z + b.z;
```

Listing 1.1: Komponentenweise Addition zweier Vektoren

rationen mit Vektortypen und skalaren Typen erlaubt, bei denen der skalare Wert implizit in einen Vektor der passenden Größe umgewandelt wird. OpenCL C definiert weitere Typen für den Zugriff auf Bildern und Sampler (Objekte, mit deren Hilfe Bilder gelesen werden können). Wie diese Typen umgesetzt werden, ist jedoch der OpenCL-Implementierung überlassen. Im Gegensatz zu C99 verbietet OpenCL C jedoch unter anderem Zeiger auf Funktionen zu erstellen sowie das Verwenden der C99 Standardbibliothek (z. B. `string.h`) und somit Zugriff auf sämtliche Hardware und Peripherie (z. B. Dateien, Zeitgeber und Signale). Ebenso sind re-

kursive Funktionen nicht erlaubt. Diese Einschränkung ermöglicht das Inlining aller Funktionen in die Hauptroutine (siehe Abschnitt 2.2.1), wodurch kein Stack benötigt wird. Die Hauptroutinen oder Einstiegspunkte, um OpenCL C-Code auf OpenCL-Geräten auszuführen, nennen sich **OpenCL-Kernel** und sind Funktionen im OpenCL C-Code, die mit dem Schlüsselwörtern `__kernel` oder `kernel` markiert sind. Kernels sind vergleichbar mit `main`-Funktionen in C-Sprachen, müssen jedoch den Rückgabetyp `void` haben. Alle Unterschiede von OpenCL C zu C99 sind im Kapitel 6 der OpenCL 1.2-Spezifikation zu finden und die Kompilierung der OpenCL C-Programmiersprache für die VideoCore IV Architektur wird im Kapitel 2 dieser Arbeit beschrieben. [Gro12, Kapitel 6]

1.2.3 OpenCL-Standardbibliothek

In OpenCL C geschriebener Code darf die C99 Standardbibliothek nicht verwenden (siehe Abschnitt 1.2.2). Der OpenCL-Standard definiert eine eigene Sammlung an vordefinierten Funktionen, die jedoch zum Teil einen anderen Schwerpunkt legt. In der OpenCL 1.2-Spezifikation wird die Standardbibliothek als Teil der OpenCL C-Programmiersprache beschrieben [Gro12, Abschnitt 6.12]. Neben den mathematischen Routinen, die auch im C99 Standard definiert sind, bietet die OpenCL-Standardbibliothek weitere Funktionen für geometrische Berechnungen (z. B. Euklidischer Abstand zweier Vektoren), Synchronisation der parallel ausgeführten Instanzen eines Kernels, sowie atomare Operationen auf gemeinsamen Speicher. Wo passend, sind diese Funktionen für skalare und Vektortypen definiert. Ebenso werden Funktionen zum Lesen und Bearbeiten von Bildern sowie Vergleich von Vektortypen spezifiziert. Da die OpenCL-Standardbibliothek von jeder OpenCL-Implementierung unterstützt werden muss, ist auch im Zuge dieser Arbeit eine Implementierung dieser Standardbibliothek entstanden, die im Kapitel 3 beschrieben ist.

1.2.4 Erweiterungen, Profile und Standard-Version

Neben dem offiziellen OpenCL-Standard werden noch eine Anzahl an offiziell unterstützten Erweiterungen definiert. Diese sind, wie auch die Liste aller anerkannten herstellerspezifischen Erweiterungen, auch auf der Khronos Website für OpenCL zu finden [Gro17a]. Die offiziellen Erweiterungen reichen von der Unterstützung von Berechnungen auf dem `half` Fließkommatyp (Erweiterung `cl_khr_fp16`) über das Teilen von Speicherbereichen mit OpenGL (Erweiterung `cl_khr_gl_sharing` sowie Direct 3D 10 (Erweiterung `cl_khr_d3d10_sharing`) bis zur Unterstützung eines Installable Client Driver (Erweiterung `cl_khr_icd`), der die gleichzeitige Verwendung mehrere OpenCL-Implementierungen ermöglicht und die Unterstützung von SPIR-V als Quellcode für OpenCL-Kernels (Erweiterung `cl_khr_il_`–

`program`) [Gro15]. `cl_khr_icd` und `cl_khr_il_program` werden von der in dieser Arbeit verfassten Implementierung unterstützt und sind in Abschnitt 4.3 beschrieben. Neben der Möglichkeit, die unterstützten Funktionen durch offizielle Erweiterungen auszuweiten besitzt der OpenCL-Standard auch sog. optionale Komponenten, die nicht implementiert werden müssen, falls z. B. die verwendeten Prozessoren diese nicht unterstützten, wobei jedoch die OpenCL-Implementierung immer noch standardkonform bleibt. Darunter zählen 64-Bit Fließkommazahlen und die Verarbeitung von Bildern jeglicher Art [Gro12, Tabelle 4.3].

Der OpenCL 1.2-Standard definiert zwei sogenannte Profile: Das `FULL_PROFILE` ist der Standard und beinhaltet alle im OpenCL 1.2-Standard beschriebenen Funktionen und Typen. Das `EMBEDDED_PROFILE` ist eine Abschwächung des Standards, indem manche Funktionen und Typen optional sind und Einschränkungen gelockert werden. So muss eine eingebettete OpenCL-Implementierung keine 64-Bit Typen (ganzzahlig oder Fließkommazahlen) unterstützen. Ebenso ist die Unterstützung mancher Bildtypen (z. B. dreidimensionale Bilder) und manche Bildoperationen optional (auch wenn die optionale Komponente der Bildverarbeitung unterstützt wird). Fließkommaberechnungen müssen denormale Zahlen (Fließkommazahlen sehr nah an null mit besonderer binären Repräsentation), NaNs und positives und negatives Unendlich nicht richtig handhaben, denormale Zahlen können mit null ersetzt werden und die Ergebnisse bei Berechnungen mit NaN und Unendlich sind nicht spezifiziert. Auch die numerische Genauigkeit, die im OpenCL 1.2-Standard im Kapitel 7 für alle mathematischen Routinen der Standardbibliothek den maximal erlaubten Fehler vom „exakten" Ergebnis festschreibt, ist gelockert, d. h. Berechnungen im eingebetteten Profil müssen weniger genau rechnen (siehe auch Abschnitt 3.1.2). [Gro12, Kapitel 10]

Wie bereits im Abschnitt 1.2 erwähnt, befindet sich der OpenCL-Standard derzeit in der finalen Version 2.2. Trotzdem wird in dieser Arbeit als OpenCL-Standard immer die OpenCL 1.2-Version genannt. Genauer gesagt, setzt die in dieser Arbeit erstellte OpenCL-Implementierung das `EMBEDDED_PROFILE` des OpenCL-Standards in der Version 1.2 um. Der Grund für die Umsetzung von OpenCL 1.2 anstatt der aktuellen Version 2.2 ist, dass OpenCL 2.0 und aufwärts eine Vielzahl an neuen Features einführt, die von dieser Implementierung (noch) nicht unterstützt werden können, wie das Starten von neuen Kerneln aus Kernel heraus. Ebenso ist die Unterstützung von 64-Bit Fließkommazahlen ab OpenCL 2.0 verpflichtend, auch für eingebettete Profile. Aus ähnlichen Gründen wird das eingebettete anstatt dem vollen Profil umgesetzt, da es der Implementierung erlaubt, 64-Bit Ganzzahlarithmetik nicht zu unterstützen. Da die VideoCore IV-Architektur nur 32-Bit breite Register sowie 32-Bit Berechnungen bietet (siehe Abschnitt 1.3.2), muss 64-Bit Arithmetik in Software emuliert werden. Dies wird aufgrund des großen Aufwands in dieser Arbeit nicht umgesetzt, kann jedoch in weiteren Entwicklungen nachgezogen werden.

1.3 VideoCore IV GPU

Der VideoCore IV-Prozessor ist ein energiesparender Grafikprozessor der Firma Broadcom, der in mobilen Endgeräten sowie in allen Modellen des Einplatinencomputers Raspberry Pi verbaut ist. Obwohl die verschiedenen Raspberry Pi-Modelle unterschiedliche CPUs besitzen, wird in allen der gleiche Grafikprozessor eingesetzt [Wik17f, Abschnitt Specifications]. Der VideoCore IV-Grafikprozessor bietet native Unterstützung für OpenGL ES 1.1 und 2.0, die OpenGL-Spezifikation für eingebettete Systeme, sowie OpenVG 1.1, eine Spezifikation für Vektorgrafiken. Der VideoCore IV-Prozessor besteht zum einen aus einer Dualcore VPU (Video Processing Unit), auf der ThreadX OS läuft und die Rechenaufgaben auf die eigentlichen Grafikprozessorkerne verteilt. Da die VPU nicht frei programmiert werden kann und die Kommunikation in der Raspberry Pi-Firmware gehandhabt wird, ist die VPU in dieser Arbeit nicht weiter von Bedeutung. Der wichtigere Teil der VideoCore IV sind die eigentlichen, frei programmierbaren, Grafikprozessorkerne, die sogenannten QPUs (Quad Processing Units, siehe Abschnitt 1.3.2). Wie man in Abbildung 1.2, der Architektur der VideoCore IV, im unteren Bereich sehen kann, werden bis zu vier Schichten (Slices) je vier QPUs unterstützt. Die tatsächliche Anzahl der Schichten ist vom Chip-Käufer (z. B. der Raspberry Pi Foundation) abhängig. Die VideoCore IV-Prozessoren der Raspberry Pi-Modelle besitzen drei Schichten je vier QPUs, also eine Summe von zwölf QPUs. [Bro13, Kapitel 1,2] [Her16]

1.3.1 Komponenten

Neben den Prozessorkernen beinhaltet die VideoCore IV auch noch eine Vielzahl von anderen Komponenten. Dazu gehören Komponenten wie der Tile Binner, der berechnet, in welchen Teilbereich der Anzeige sich Koordinaten befindet, die TMU (Texture and Memory Lookup Unit) zum Laden von Texturen aus dem Arbeitsspeicher sowie die PSE (Primitive Setup Engine) zum Laden von Vertices. Diese genannten Komponenten sind sehr spezifisch auf die Verarbeitung von grafischen Daten zugeschnitten und daher für diese Arbeit nicht weiter von Nutzen. Jedoch bietet die VideoCore IV-Architektur auch für GPGPU nützliche Komponenten an:

Die SFU (Special Functions Unit) kann mathematische Funktionen wie die Quadratwurzel ($sqrt(x)$), den Zweierlogarithmus ($log_2(x)$) sowie die Exponentialfunktion auf Basis zwei (2^x) und das multiplikative Inverse ($1/x$) für Fließkommazahlen annähern, jedoch mit einer sehr schlechten Genauigkeit von ungefähr elf Binärstellen (anstatt der 24 Stellen Genauigkeit des Datentyps `float`). Der Wert `NaN` für undefinierte Werte wird von der SFU nicht unterstützt, jedoch werden positiv und negativ unendliche Werte richtig repräsentiert. Ebenso benötigt die Berechnung dieser Funktionen drei Taktzyklen, in denen das Register, in das das Ergebnis ge-

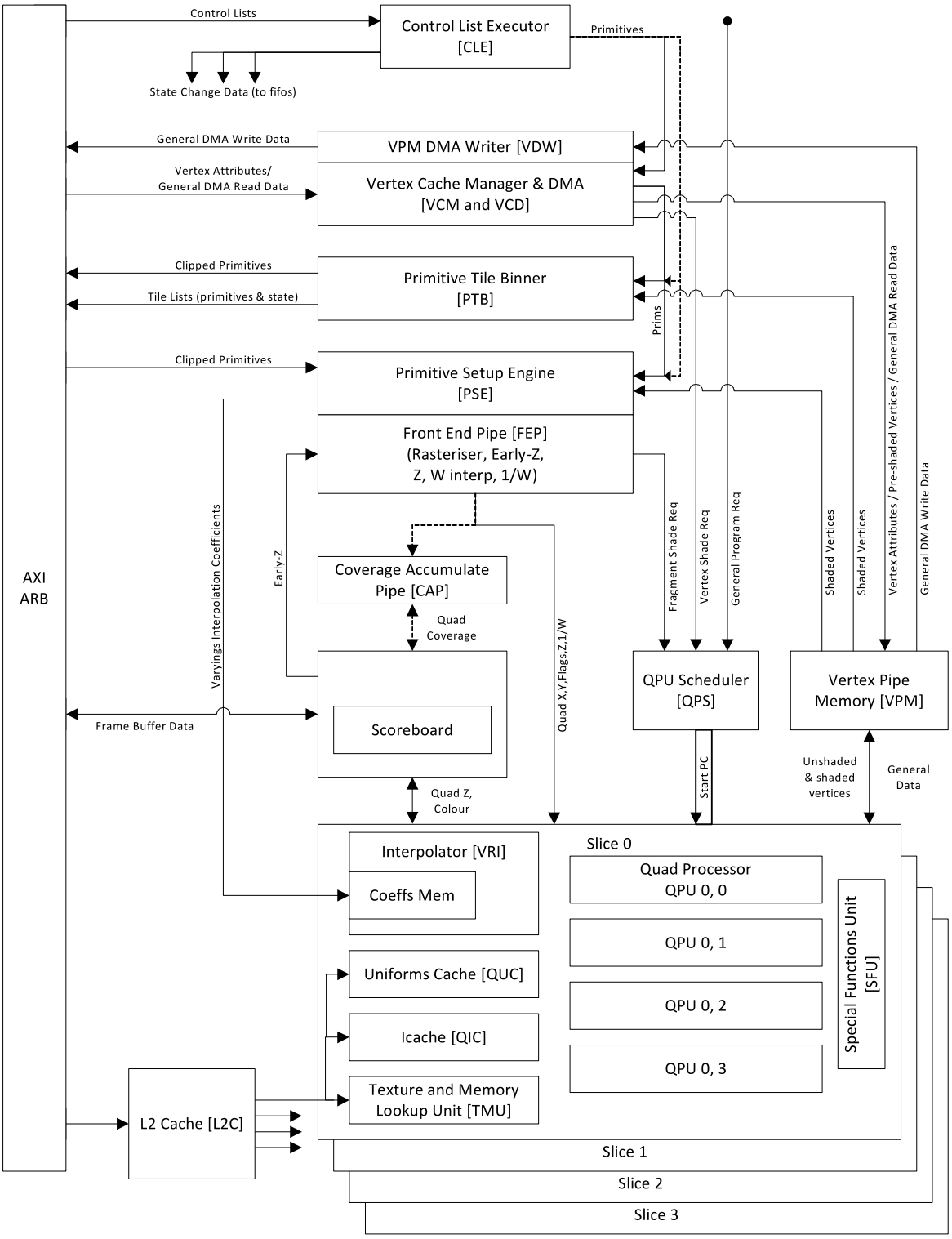

Abbildung 1.2: VideoCore® IV 3D System Block Diagram [Bro13, Abbildung 1]

speichert wird, nicht gelesen werden darf und auch die Register zum Setzen der Operanden für die SFU-Aufrufe (siehe Abschnitt 1.3.2) nicht beschrieben werden dürfen. Die VPM (Vertex Pipe Memory) dient als Zwischenspeicher für Zugriffe auf den Arbeitsspeicher. Hier können mehrere Werte gesammelt werden bevor diese über den VDW (VPM DMA Writer) in den Arbeitsspeicher geschrieben werden. Ebenso können über die Komponenten VCM und VCD (Vertex Cache Manager und Vertex Cache DMA) mehrere Datensätze aus dem Hauptspeicher in die VPM geladen werden, um dann einzeln von den QPUs gelesen zu werden. Außerdem bietet die VideoCore IV-Architektur eine mehrstufige Cache für Instruktionen und Daten. Während in der L2 Cache noch Instruktionen und Daten zusammen gespeichert werden, werden diese in den QIC (Instruction Cache) und QUC (Unforms Cache, zum Speichern der konstanten Parameter) aufgeteilt. Die Peripherie, die in Abbildung 1.2 außerhalb der Schichten (Slices) dargestellt ist, wie z. B. VPM, VDW, VCM und VCD, wird zwischen allen QPUs geteilt. Andere Peripherie, wie die SFU, die QIC und QUC werden nur innerhalb einer Schicht geteilt, d. h. es gibt für jede Schicht (je vier QPUs) eine Instanz dieser Komponenten. Um die Zugriffe auf die verschiedenen Komponenten zu synchronisieren, stehen den QPUs weiterhin ein globales Mutex sowie 16 globale Semaphore zur Verfügung. [Bro13, Kapitel 2]

1.3.2 Quad Processing Unit

Eine Quad Processing Unit ist ein frei programmierbarer Prozessorkern in der VideoCore IV GPU mit einer Taktrate von 250 MHz. Eine QPU besitzt einen 64-Bit Befehlssatz (siehe Abbildung 1.3), ist jedoch eine 32-Bit Architektur, d. h. die Größe eines Registerelements sowie eines Zeigers ist 32 Bit. Die QPU bietet 16-fachen virtuellen Datenparallelismus (SIMD), d. h. es werden virtuell immer 16 Werte mit einer Instruktion gleichzeitig behandelt, auch wenn nicht alle davon genutzt werden. Technisch ergibt sich der 16-fache virtuelle Parallelismus durch einen vierfachen physikalischem Parallelismus (sog. „quads", daher auch die **Quad** Processing Unit), der vier Taktzyklen hintereinander ausgeführt wird. Daraus ergibt sich auch, dass jede Instruktion aus dem Befehlssatz der QPU vier Taktzyklen benötigt. Da jedoch die darauf folgende Instruktion bereits auf dem ersten Quad ausgeführt werden kann, wenn die vorherige Instruktion auf dem zweiten Quad arbeitet, wird in jedem Taktzyklus eine neue Instruktion gestartet (siehe Tabelle 1.1). Der Einfachheit halber wird die vierstufige Instruktionspipeline bei Sprüngen nicht geleert, ebenso wird keine „branch prediction" – also das Vorhersagen, ob ein bedingter Sprung ausgeführt wird oder nicht – unterstützt. Daraus folgt, dass hinter jedem Sprungbefehl drei Instruktionen gewartet werden muss, bis der Sprung vollständig ausgeführt ist. Jede QPU besitzt zwei asymmetrische ALUs (Arithmetic Logic Units), mit unterschiedlichen Operationen für Addition und Multiplikation. Innerhalb einer einzelnen Instruktion können Operationen gleichzeitig auf beiden ALUs ausgeführt

	Takt 0	**Takt 1**	**Takt 2**	**Takt 3**	**Takt 4**	**Takt 5**
Quad 0	Instr0	Instr1	Instr2	...	...	...
Quad 1		Instr0	Instr1	Instr2	...	...
Quad 2			Instr0	Instr1	Instr2	...
Quad 3				Instr0	Instr1	Instr2

Tabelle 1.1: Abarbeitungsreihenfolge der Quads in den QPUs

Abbildung 1.3: QPU Instruction Encoding [Bro13, Abbildung 3]

werden. Durch die zwei asymmetrischen ALUs, die innerhalb einer Instruktion angesprochen werden können, den vierfachen physikalischen Datenparallelismus, die Taktrate von 250 MHz und die Gesamtanzahl von zwölf QPUs ergibt sich eine theoretische Maximalleistung der VideoCore IV GPU des Raspberry Pis von 24 GFLOPS oder 24 GIOPS ($24 * 10^9$ Fließkomma- oder Ganzzahlinstruktionen pro Sekunde). [Bro13, Kapitel 3]

Jede QPU besitzt zwei physikalische Registerbänke A und B mit jeweils 32 Registern sowie fünf speziellen Registern, den Akkumulatoren. Im Gegensatz zu den physikalischen Registern können die Akkumulatoren in zwei aufeinander folgenden Instruktionen geschrieben und wieder gelesen werden und werden daher für die Verwendung als temporäre Speicher bevorzugt. Sowohl die physikalischen Register als auch die Akkumulatoren beinhalten 16 Elemente (aufgrund der 16-fachen SIMD-Eigenschaft) je 32 Bit. Jedes Element eines Registers wird von dem Element des SIMD-Prozessors mit dem gleichen Element-Index bearbeitet. Dies entspricht

der komponentenweisen Berechnung in OpenCL (siehe Listing 1.1). Über spezielle Befehle der Multiplikations-ALU können die Vektorelemente frei rotiert werden, d. h. jede der möglichen Rotationen (0 bis 15 Schritte) ist möglich. Die in Abschnitt 1.3.1 genannten Hardwarekomponenten (z. B. SFU, Mutex, VPM) werden über spezielle Register angesprochen, wodurch keine zusätzlichen Instruktionen für den Zugriff auf die Peripherie benötigt werden. [Bro13, Abschnitt QPU Register Address Map]

Der Befehlssatz der QPU gliedert sich in vier unterschiedliche Arten von Befehlen, die sich in ihrer Bedeutung grundsätzlich unterscheiden: Die **ALU-Befehle** können beide ALUs gleichzeitig ansteuern, führen also die eigentlichen Berechnungen aus. Hiervon gibt es zwei Varianten. Während das Befehlsformat *alu* (siehe Abbildung 1.3) beide Eingangsregister *raddr_a* und *raddr_b* aus den beiden Registerbänken A und B liest, wird im Format *alu small imm* der Wert in *raddr_b* direkt als Argumentwert genommen (sog. Immediate Value). **Sprünge**, sowohl bedingt als auch unbedingt, werden über das Befehlsformat *branch* ausgeführt. Die Bedingung des Sprungs ist von den Statusregistern aller 16 SIMD-Elemente abhängig und kann Werte wie „alle Null-Statusbits gesetzt", „kein Carry-Flag gesetzt" oder „mindestens ein Negativ-Flag gesetzt" annehmen. Konstante Werte (die zu groß für das Small Immediate-Feld sind) können über die verschiedenen Variationen des **Lade**-Befehls geladen werden. Im Verlauf dieser Arbeit ist nur die Variante *load imm 32* von Interesse, die die Bits im Feld *immediate* in alle 16 Elemente der Ziel-Register *waddr_add* und *waddr_mul* kopiert. Das **Semaphore** Befehlsformat wird verwendet um eins der 16 Hardware-Semaphore (siehe Abschnitt 1.3.1) zu inkrementieren oder dekrementieren und blockiert, wenn das Semaphore unter den Wert null oder über den Wert 15 gesetzt werden soll. [Bro13, Abschnitt QPU Instruction Encoding]

Die beiden ALUs sind asymmetrisch, besitzen also unterschiedliche Operationen [Bro13, Abschnitt QPU Instruction Set]: Die „Additions"-ALU besitzt insgesamt 24 Operationen, darunter Addition, Subtraktion, Berechnung des Minimums/Maximums zweier Werte, Shifts und bitweises UND, NICHT, XOR und OR. Für einige dieser Operationen (Addition, Subtraktion, Maximum/Minimum) gibt es einen Operationscode für 32-Bit große ganze Zahlen sowie einen für die Berechnung mit 32-Bit Fließkommazahlen. Die „Multiplikations"-ALU besitzt nur acht Operationen, darunter eine Fließkommamultiplikation sowie eine 24-Bit Ganzzahlmultiplikation, die zwei 24-Bit Zahlen zu einer 32-Bit Zahl multipliziert. Andere häufig verwendete arithmetische Operationen wie die Division (mit ganzen Zahlen oder Fließkommazahlen), die 32-Bit Ganzzahlmultiplikation sowie das Berechnen des Restes (Modulo) werden von dem Befehlssatz der QPUs nicht angeboten und müssen daher in Software implementiert werden (siehe Abschnitt 2.2.4). Wenn in dieser Arbeit von „Maschinencode" oder „Programmierung der GPU" gesprochen wird, dann bezieht sich das auf den oben erwähnten Befehlssatz der QPUs.

1.3.3 Speicherzugriff

Im Gegensatz zu rechenstarken Desktop-Grafikkarten wie der Nvidia GTX 960 [NVI17] besitzt die VideoCore IV GPU keinen eigenen Grafikspeicher, auf dem Daten für Berechnungen zwischengespeichert werden können. Stattdessen können die QPUs über die Komponenten der VDW (VPM DMA Writer) sowie VCD (Vertex Cache DMA) (siehe Abschnitt 1.3.1) über DMA (Direct Memory Access) direkt auf den Arbeitsspeicher zugreifen. Um genug Speicher für grafische Anwendungen bereitzustellen, wird im Raspbian OS beim Booten in der Datei `/boot/config.txt` festgeschrieben, wie viel Arbeitsspeicher für die VideoCore IV reserviert wird. In diesem Speicherbereich werden z. B. OpenGL Shader oder Vertices für 3D Anwendungen für die Berechnung in den QPUs zwischengespeichert. Jedoch können die QPUs nicht nur auf den statisch reservierten Bereich, sondern auf den gesamten Arbeitsspeicher zugreifen, was zu einem Sicherheitsproblem führt. Die OpenGL-Implementierung in Mesa für die VideoCore IV GPU (siehe Abschnitt 1.4) bekämpft dieses Problem durch Überprüfung des Shader-Codes bei der Kompilierung [Anh16]. Da jedoch OpenCL-Kernel, im Gegensatz zu OpenGL-Shadern, die Speicherbereiche, auf denen sie ihre Berechnungen durchführen zur Laufzeit über die Kernel-Parameter übergeben bekommen, kann bei der Kompilierung ein Zugriff auf sicherheitskritische Bereiche nicht kontrolliert werden. Zur Lösung dieses Problems müssen daher vorerst alle Programme, die die OpenCL-Implementierung in dieser Arbeit verwenden, mit root-Rechten ausgeführt werden. Dies stellt sicher, dass der Benutzer bereits alle Rechte besitzt und kein Bedarf mehr besteht, über OpenCL-Kernels das System zu manipulieren.

1.4 Bestehende Arbeiten

In diesem Abschnitt gehe ich kurz auf eine Auswahl bereits existierender Programme und Bibliotheken ein, die sich auch mit den Themen „Unterstützung von OpenCL auf Raspberry Pi" und „GPGPU auf der VideoCore IV GPU" beschäftigen:

Es existieren bereits eine Vielzahl an Assemblern und Disassemblern, die zwischen Maschinencode für die VideoCore IV GPU und meist benutzerdefinierten Assemblersprachen konvertieren können. Einer dieser Assembler, der sehr umfangreich ist und sich derzeit noch in aktiver Entwicklung befindet, ist **vc4asm** [Mü14]. **vc4asm** beinhaltet auch eine Validierung, die den Maschinencode auf Fehler im Hinblick auf besondere Restriktionen mancher Befehle überprüft. Diese Validierung wird in dem im Zuge dieser Arbeit entstehende Compiler verwendet, um die richtige Beachtung dieser Restriktionen zu überprüfen und ist im Abschnitt 2.3.3 genauer beschrieben. Ebenso wird in dieser Arbeit der **vc4asm** Disassembler verwendet, um stichprobenartig überprüfen zu können, ob der richtige Maschinencode generiert

wird. Das Programmieren in diesen Assemblersprachen ist jedoch sehr aufwendig und fehleranfällig, wodurch sie sich nicht eignen um effizient und im großen Stil Code für die VideoCore IV GPU zu entwickeln.

Wie alle anderen GPUs, wird auch die VideoCore IV primär für Grafikberechnungen verwendet. Unter Linux (z. B. der Raspbian-Distribution, eines für den Raspberry Pi entwickelten Debian-Derivats) nutzen grafische Anwendungen die quelloffene Bibliothek Mesa. Mesa bietet eine plattformunabhängige Implementierung von Grafik-APIs wie unter anderem OpenGL, Vulkan und OpenVG [Mes17]. Zugriffe auf diese APIs werden über eine gemeinsame interne Struktur auf eins der bestehenden Backends für die jeweils aktive Grafikkarte umgeleitet. Seit Version 10.3 (August 2014) gibt es auch ein Mesa-Backend für die VideoCore IV GPU des Raspberry Pi [Wik17e, Abschnitt Broadcom]. Das **VideoCoreIV**-Backend für Mesa verwendet ein Kernel-Modul für den Zugriff auf die VideoCore IV-Hardware, das auch von der in dieser Arbeit verfassten OpenCL-Implementierung verwendet wird (siehe Abschnitt 4.2.4). Für manche Plattformen bietet Mesa auch eine Implementierung der OpenCL-API, die unter dem Namen Clover entwickelt wurde und mittlerweile als GalliumCompute in die Mesa-Bibliothek integriert ist. Die OpenCL-Schnittstelle ist jedoch für das **VideoCoreIV**-Backend nicht verfügbar. Wie bereits in Abschnitt 1.3.3 erwähnt, kontrolliert die Mesa-Implementierung für die VideoCore IV GPU den OpenGL-Shadercode bei der Kompilierung auf unerlaubte Speicherzugriffe, um z. B. eine Eskalation der Benutzerrechte durch überschreiben von Systemspeicher zu verhindern [Anh16].

Ebenso existiert bereits ein Anfang, ein LLVM-Backend für die QPUs der VideoCore IV GPU zu erstellen [Hal14b]. Dieses Backend wird seit 2014 nicht mehr weiter entwickelt und ist laut Angaben des Entwicklers Simon Hall sehr unvollständig. Im offiziellen Forum der Raspberry Pi Foundation beschreibt Hall seinen Fortschritt der Entwicklungen und nennt einige Mängel der Implementierung [Hal14a]: So nutzen Integer-Multiplikationen nur die ALU-Instruktion `mul24`, die nur die unteren 24 Bits multipliziert anstatt alle 32 Bits, Fließkomma-Division und Vergleiche mit Fließkomma-Zahlen sind nicht implementiert, genauso wie 8- und 16-Bit Integer Arithmetik. Ebenso bietet das Backend keine Implementierung für Built-In Funktionen (z. B. der OpenCL oder C Standardbibliotheken). Da dieses LLVM-Backend sehr unvollständig ist und seit längerem nicht mehr aktiv weiter entwickelt wird, eignet es sich auch nicht, um produktiv Code für GPGPU auf der VideoCore IV GPU zu schreiben.

Eine bereits bestehende Möglichkeit, OpenCL auf einer Vielzahl von Plattformen, einschließlich den Raspberry Pi-Modellen, auszuführen, ist die Bibliothek **pocl** [pd10]. **Pocl** nutzt das OpenCL-Frontend der LLVM-Kompilersuite, um OpenCL-Kernel für die Host-CPU zu kompilieren und dort auszuführen. Somit kann **pocl** OpenCL auf allen Plattformen unterstützen, für die es ein LLVM-Backend gibt.

Jedoch werden die Kernel auf der CPU und nicht auf der GPU ausgeführt, wodurch keine Leistungssteigerung erreicht wird. In dieser Arbeit wird **pocl** im Abschnitt 5.3.2 verwendet, um die Performance der Ausführung verschiedener Kernel auf der VideoCore IV GPU und der ARM Host-CPU des Raspberry Pi zu vergleichen.

1.5 Existierende OpenCL C-Compiler

Des Weiteren existieren bereits eine Hand voll Compiler für die OpenCL C-Programmiersprache. Der OpenCL-Standard schreibt vor, dass jede OpenCL-Implementierung einen solchen Compiler beinhaltet, um Kernel in einem der unterstützten plattformunabhängigen Formaten (OpenCL C-Sourcecode, SPIR, SPIR-V) ausliefern zu können, die dann bei Verwendung für die richtige Plattform kompiliert werden können. Deshalb beinhalten die OpenCL-Implementierungen von z. B. AMD, Nvidia, Intel plattformspezifische OpenCL C-Compiler, die jedoch meist proprietären Quellcode haben sowie aufgrund der spezifischen unterstützten Architektur für eine Implementierung auf den Raspberry Pi-Modellen nicht relevant sind. Für die Verwendung in dem in Kapitel 2 vorgestellten Compiler sind jedoch zwei offene Compiler mit OpenCL C-Frontend interessant:

1.5.1 LLVM

Wie bereits erwähnt, bietet die LLVM-Kompilersuite ein OpenCL C-Frontend. Dieses ist im Standard C/C++-Frontend namens CLang integriert und ist somit standardmäßig Teil aller LLVM-Installationen. Es gibt zwar kein funktionsfähiges LLVM-Backend für die VideoCore IV GPU, jedoch kann LLVM verwendet werden, um den Kernel-Sourcecode in eine besser maschinenlesbare Form (namens LLVM IR) zu bringen. Ebenso übernimmt LLVM bereits das Überprüfen auf Syntax-Fehler, sowie das Einbinden benötigter Funktionsdeklarationen und plattformunabhängige Optimierungen. Um den Code weiter verarbeiten zu können, kann LLVM dazu gebracht werden, die interne Repräsentation des verarbeiteten Codes, die LLVM IR (LLVM Intermediate Representation), in einem textuellen Format auszugeben. LLVM IR ist Assembler-ähnlich, beinhaltet also außer Sprüngen keinerlei Kontrollstrukturen mehr, sowie extra Instruktionen (Load, Store) für den Zugriff auf den Arbeitsspeicher und ist auf [LLV17b] ausführlich dokumentiert. LLVM IR ist SSA-basiert (Static Single Assignment Form), was bedeutet, dass jede Variable genau einmal an einer Stelle einen Wert zugewiesen bekommt und vor der Verwendung definiert sein muss, also Variablen niemals undefiniert sein oder überschrieben werden können [Wik17g]. Da die Ausgabe der LLVM IR vor dem Verlinken der Funktionen (und somit vor dem Aussondern der nicht-benötigten Funktionen) geschieht, werden in die ausgegebene Textdatei alle Funktionen geschrieben, die in der Quelldatei (und

allen eingebunden Headern) enthalten sind, was zu einer sehr großen Menge an zu verarbeitenden Text führen kann.

Ein Beispiel eines LLVM IR-Codes kann in Anhang A.2 gesehen werden. Dieser Code wird aus dem OpenCL-Kernel zur Berechnung der nächsten zehn Fibonaccizahlen ausgehend von zwei Parametern aus Anhang A.1 erzeugt. Anhang A.2 zeigt jedoch nur den für den Kernel wichtigen Teil des LLVM IR Codes. Das originale Ergebnis umfasst mehr als 83000 Zeilen und ist fast 5 MB groß.

1.5.2 SPIRV-LLVM

Khronos Group, das Standardkomittee hinter OpenCL, entwickelt an einer eigenen Version der LLVM-Kompilersuite [Gro17d], die compilierten Code in die Zwischensprache SPIR-V umwandelt. SPIR-V ist ein relativer neuer Standard (die Version 1.0 ist 2015 erschienen, die derzeit neuste Version 1.2 am 16. Mai 2017 [Gro17b]) der Khronos Group und wurde speziell als einheitliche Zwischensprache für auf GPUs ausgeführter Code (z. B. OpenGL, Vulkan oder auch OpenCL) entwickelt. SPIR-V ist im originalen Whitepaper als „Header und ein linearer Strom von Instruktionen" beschrieben und soll als plattformunabhängiges Format dienen, das aus mehreren Quellsprachen erzeugt werden kann und das geistige Eigentum der Entwickler schützen soll, was z. B. bei einer Auslieferung des OpenCL-Quellcodes nicht gegeben ist [Gro15]. Ähnlich wie LLVM IR ist SPIR-V SSA-basiert. Die Abbildung 1.4 zeigt die Einbindung von SPIR-V in mehreren Quellsprachen sowie Zielarchitekturen. Im Gegensatz zu der Ausgabe von LLVM IR der „normalen" LLVM-Kompilersuite, ist das ausgegebene SPIR-V ein Binärformat und beinhaltet nur die Funktionen, die von den Einstiegspunkten (OpenCL-Kernels) tatsächlich benötigt werden, was zu einer einfacheren Verarbeitung sowie sehr viel kleinere Dateigrößen führt.

Derselbe OpenCL-Kernel zur Berechnung der Fibonaccizahlen aus Anhang A.1 ergibt, mit SPIRV-LLVM kompiliert, den SPIR-V-Code in Anhang A.3. Zur besseren Lesbarkeit wurde der Code in Anhang A.3 in die textuelle Repräsentation von SPIR-V ungewandelt. Im Gegensatz zu dem Ergebnis der „normalen" LLVM-Kompilersuite hat der gesamte Code nur knapp mehr als 60 Zeilen und eine Größe von 1,3 KB.

Wie bereits erwähnt, ist SPIR-V ein linearer Strom von Instruktionen, wobei sich in der binären Darstellung jede Instruktion als eine Folge von 32-Bit Wörtern beschreiben lässt. Dies bedeutet, dass jegliche Information eines SPIR-V-Moduls (ein Modul ist eine Quelldatei) als Instruktion dargestellt wird. Wie man im Codebeispiel in Anhang A.3 sehen kann, folgen (bis auf die Header-Zeile) alle Zeilen dem gleichen Format. Dies gilt auch für die binäre Repräsentation, die vom Aufbau her gleich dem Textformat ist, nur dass die Namen (wie z. B. `Capability` oder `FuncParamAttr`)

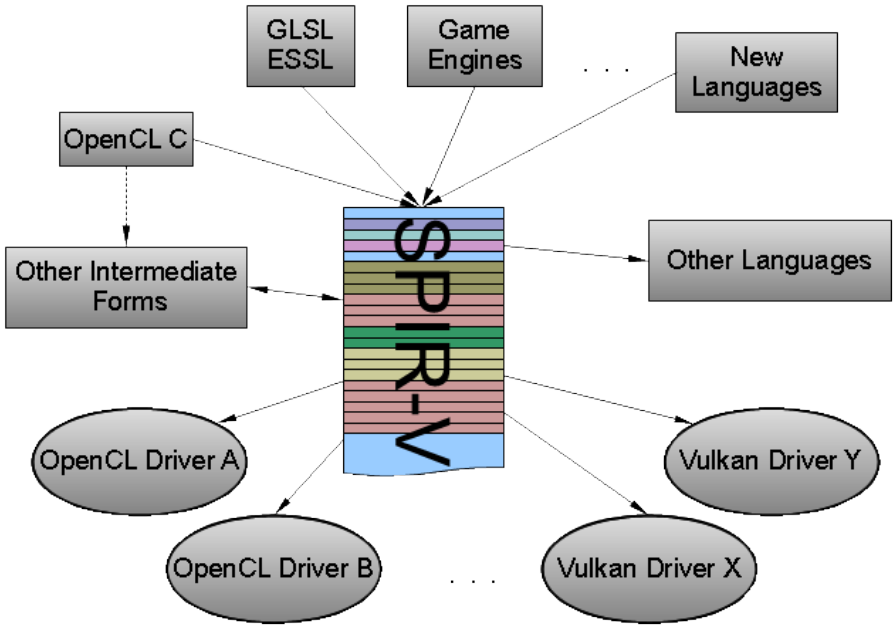

Abbildung 1.4: SPIR-V als gemeinsame Zwischensprache für mehrere Quellsprachen und Treiber [Gro15]

als fest spezifizierte Ganzzahlkonstanten gespeichert sind, sowie es keine Leerzeichen oder Zeilenumbrüche zwischen den Werten und Instruktionen gibt. Das erste Wort (32-Bit) jeder Instruktion besteht aus der Anzahl der Wörter der Instruktion, sowie den Opcode, der die Art der Instruktion festlegt. Darauf folgt die ID des Ergebnistyps und die ID des Ergebnisses, falls es ein solches gibt, sowie weitere Operanden, deren Bedeutung vom Opcode abhängig ist. So definiert die Instruktion `4 Constant 3 16 2` eine Konstante des Typs 32-Bit Integer (der vorher für die ID 3 festgelegt wurde), die die ID 16 und den Wert 2 besitzt. Ähnlich definiert die Zeile `5 Function 2 6 0 5` eine Funktion vom Typ `void` mit der ID 6 sowie weiteren, nicht weiter relevanten Attributen. Alle darauf folgenden Instruktionen gehören zu dieser Funktion, bis zu dem Opcode `FunctionEnd`. Danach kann wieder mit `Function` eine weitere Funktion definiert werden.

1.6 Aufbau der Arbeit

Die in dieser Arbeit erstellte OpenCL-Implementierung heißt **VC4CL**, was für **V**ideo**C**ore **4** Open**CL** steht. Diese Implementierung ist in folgende drei Projekte aufgeteilt, die mit dem Inhalt der jeweiligen Kapitel 2, 3 und 4 korrespondieren:

VC4CC ist die **V**ideo**C**ore **4** **C**ompiler **C**ollection und ist die Implementierung des OpenCL C-Compilers (siehe Abschnitt 1.2.2), die in Kapitel 2 beschrieben ist. Auch wenn die VC4CL OpenCL-Implementierung für die Raspberry Pi-Modelle geschrieben ist, lässt sich der VC4CC Compiler auf allen Systemen kompilieren und ausführen, für die ein moderner C++11-Compiler verfügbar ist. Ebenso kann VC4CC so konfiguriert werden, dass es auf einem leistungssärkeren System für die ARM-Architektur des Raspberry Pi kompiliert wird (mithilfe des Raspberry Pi cross-compilers [Fou17b]).

Die in Kapitel 3 beschriebene Implementierung der OpenCL-Standardbibliothek (siehe Abschnitt 1.2.3) heißt **VC4CLStdLib**, in Anlehnung an `stdlib.h` für die C-Standardbibliothek. VC4CLStdLib ist eine reine Header-Sammlung, wird also nicht in eine extra Bibliothek kompiliert, sondern wird von VC4CC bei der Kompilierung der OpenCL-Kernel mit eingebunden.

Die hostseitige OpenCL-Laufzeitbibliothek (siehe Abschnitt 1.2.1) wird im Projekt **VC4CL** implementiert und ist in Kapitel 4 beschrieben. Die Laufzeitbibliothek lässt sich ähnlich wie der VC4CC-Compiler auf einer Vielzahl an Systemen kompilieren, sowie für den Raspberry Pi cross-kompilieren, jedoch kann diese nur auf Raspberry Pi Modellen ausgeführt werden, da sie auf die Hardware der VideCore IV GPU (siehe Abschnitt 1.3) zugreift.

Alle drei Projekte werden mit der Entwicklungsumgebung Eclipse und dem CDT (Eclipse C/C++ Development Tools) entwickelt. Für das Buildsystem wird CMake verwendet, ein plattformunabhängiges Buildsystem für C/C++ Projekte, das die wichtigsten Plattformen (Windows, Linux, Mac OS) unterstützt, indem es die eigene Konfiguration vor der Kompilierung in eine Konfiguration eines jeweils nativen Buildsystems umwandelt (z. B. Unix Makefiles oder Visual Studio Projekt-Konfigurationen).

Der Quellcode der drei Teilprojekte sowie des zusätzlich zu Demonstrationszwecken erstellten Projektes **GameOfLife** (siehe Abschnitt 5.2.2) ist in digitaler Form den gebundenen Ausgaben dieser Arbeit beigelegt. Die in dieser Arbeit beschriebenen Implementierungen beziehen sich auf den Codestand, wie er auf den beigelegten CDs zu finden ist. Die jeweils aktuelle Version aller vier Projekte kann in den Github Repositories `https://github.com/doe300/VC4C`, `https://github.com/doe300/VC4CLStdLib`, `https://github.com/doe300/VC4CL` und `https://github.com/doe300/GameOfLife` eingesehen werden.

2. Compiler

Die **VC4CC** (**V**ideo**C**ore **4** **C**ompiler **C**ollection, angelehnt an die **G**NU **C**ompiler **C**ollection) ist ein Compiler, der OpenCL C-Quellcode sowie SPIR-V-Zwischencode in Maschinencode für die VideoCore IV-Architektur umwandeln kann. Somit implementiert der VC4CC-Compiler den Teil des OpenCL-Standards, der sich mit der OpenCL C-Programmiersprache beschäftigt (siehe Abschnitt 1.2.2). Aufgebaut ist der Compiler in drei Ebenen: Das Frontend (Abschnitt 2.1) ist für das Einlesen der unterstützten Quellsprachen und das Umwandeln in die interne Repräsentation zuständig. Im Middleend (Abschnitt 2.2) werden Umformungen nicht-unterstützter Befehle in unterstützte Varianten, sowie eine Reihe an Optimierungen ausgeführt. Das Backend (Abschnitt 2.3) weist den Variablen Registern zu und wandelt den Code in Maschinencode zur Ausführung auf den QPUs um. Abbildung 2.1 zeigt den schematischen Aufbau des Compilers. Die an der jeweiligen Stelle verwendeten Programmiersprachen oder Repräsentationen des zu kompilierenden Programms sind als UML Notizen dargestellt. Externe Programme, wie die beiden unterstützten Compiler LLVM und SPIRV-LLVM (siehe Abschnitt 2.1.1) und der vc4asm Assembler zum Validieren des generierten Codes (siehe Abschnitt 2.3.3) sind mit grauen Hintergrund hinterlegt.

Die Prioritäten, nach denen der VC4CC Compiler geschrieben ist, lassen sich wie folgt beschreiben:

1. Erzeugung richtigen Codes: Für die (bisher) unterstützten Programm-Quellcodes soll richtiger Code erzeugt werden. Zum einen, weil es zur Laufzeit die Ergebnisse nicht weiter kontrolliert werden und eine OpenCL-Anwendung dann falsche Werte berechnet. Zum anderen aber auch, weil falscher Maschinencode, z. B. fehlerhafte Instruktionen oder Instruktionen in der falschen Reihenfolge sehr schnell die VideoCore IV GPU aufhängen kann, was wiederum dazu führen kann, dass der gesamte Raspberry Pi einfriert.

2. Unterstützung vieler Quellprogramme: Wie bei jedem anderen Compiler auch, ist es ein Ziel, alle (oder zumindest so viel wie möglich) gültige Programm-Quellcodes in korrekte Maschinencode-Programme umwandeln zu können.

3. Erzeugung von schnellem Code: Der generierte Maschinencode soll nicht nur richtige Ergebnisse zurückgeben, sondern diese auch in möglichst wenig Zeit berechnen. Hier geht es nicht darum, den „optimalen" Code zu generieren.

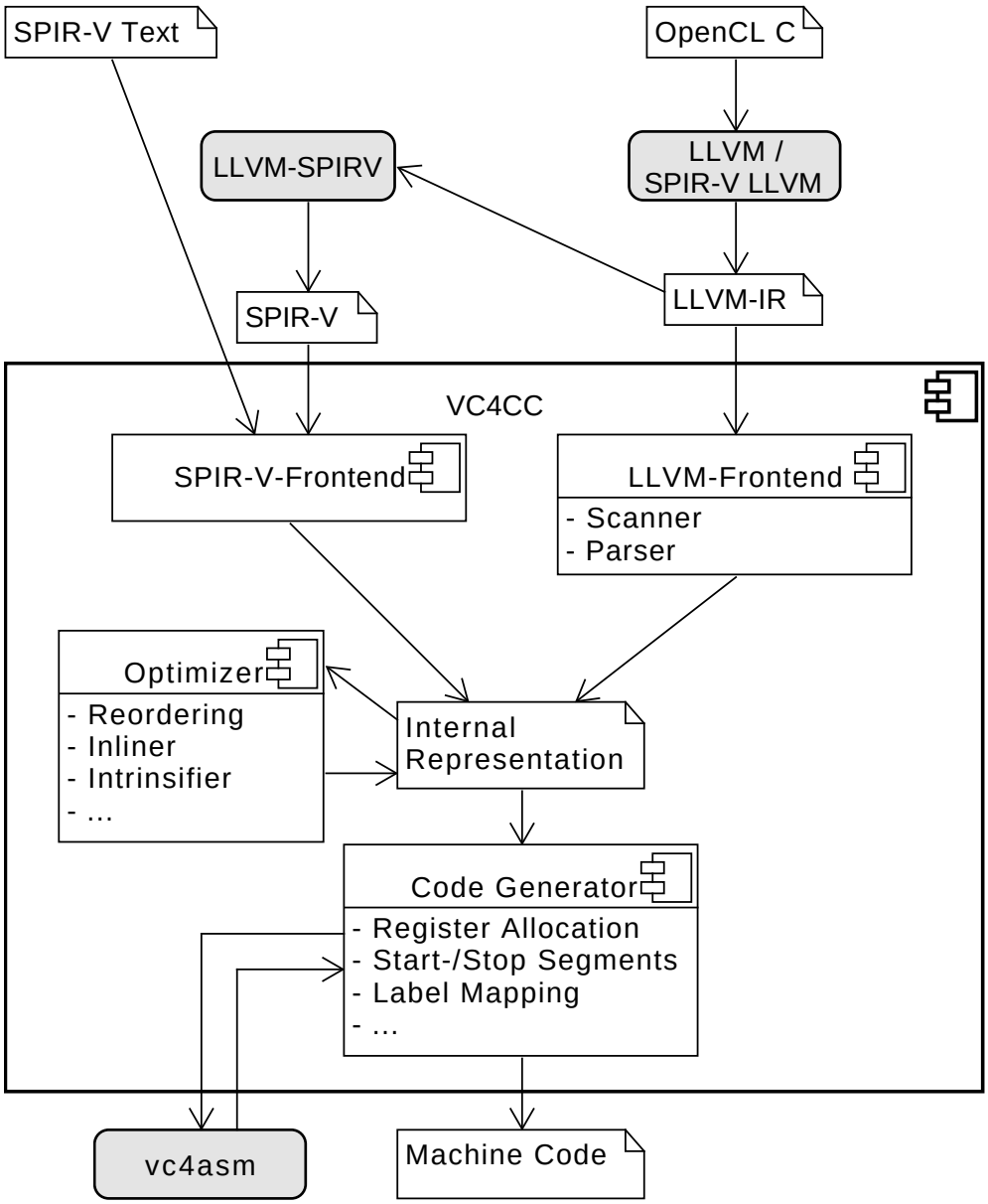

Abbildung 2.1: Schematischer Aufbau des VC4CC Compilers

Stattdessen wird versucht, unnötige oder doppelte Operationen zu eliminieren, sowie den Code so umzuwandeln, dass zumindest eine Auswahl der Besonderheiten der VideoCore IV-Architektur zur Generierung effizienteren Maschinencodes genutzt werden kann.

In OpenCL wird jedoch, im Gegensatz zu anderen Programmiersprachen wie C/C++, der Quellcode meist erst zur Laufzeit der Anwendung für die jeweilige Zielarchitektur kompiliert (siehe Abschnitt 4.2.2). Dies bedeutet, dass ein OpenCL-Compiler auch noch die zusätzliche Vorgabe besitzt, schnell zu kompilieren. Da die beste Optimierung des Maschinencodes nicht viel Zeit erspart, wenn die Kompilierung dieses Programms vorher sehr viel länger dauert, muss bei jedem Optimierungsschritt überlegt werden, mit welchem (zeitlichen) Aufwand bei der Kompilierung der Gewinn einer schnelleren Ausführung erkauft wird.

2.1 Frontend

Das Frontend eines Compilers hat die Aufgabe, Quellcode einzulesen, auf lexikalische, syntaktische und semantische Fehler zu überprüfen und in eine interne Repräsentation umzuwandeln, die von den weiteren Teilen des Compilers bearbeitet werden kann. Lexikalische Fehler können z. B. Tippfehler sein (`3.141.59265359` statt `3.14159265359` für die Konstante Pi), syntaktische Fehler sind z. B. falsch platzierte oder weg gelassene Schlüsselwörter oder Symbole mit besonderer Bedeutung (wie die geschweiften Klammern zur Markierung von Gültigkeitsbereichen oder das Semikolon als Abschluss einer Instruktion in C-ähnlichen Sprachen), während semantische Fehler inhaltlich falsche oder nicht ausführbare Instruktionen darstellen, wie das Multiplizieren eines Fließkommavektors mit einer Ganzzahlkonstante in OpenCL C. Neben der Fehleranalyse führt ein Compiler-Frontend gegebenenfalls vorher noch die Vorverarbeitung von Macros aus. [Wik17c, Abschnitt Front end]

Als gültige Quellsprachen unterstützt der in dieser Arbeit erstellte Compiler OpenCL C (siehe Abschnitt 1.2.2) sowie die Zwischensprachen LLVM IR und SPIR-V, die aus den jeweiligen in Abschnitt 1.5 genannten OpenCL C-Compiler erzeugt werden können (siehe auch Abbildung 2.1).

2.1.1 Vorverarbeitung

Der VC4CC-Compiler bietet zwei verschiedene Frontends für die Verarbeitung von LLVM IR (siehe Abschnitt 2.1.2) und SPIR-V (siehe Abschnitt 2.1.3) an. Die beiden Sprachen LLVM IR und SPIR-V wurden für die einfache maschinelle Lesbarkeit entwickelt und können somit einfach vom VC4CC eingelesen werden. OpenCL C hingegen besitzt, durch die von C vererbte verschachtelte Struktur, eine sehr

viel größere syntaktische Komplexität, was die Entwicklung eines Frontends zum Einlesen von OpenCL C sehr viel komplizierter macht. Um jedoch auch OpenCL C-Quellcode unterstützen zu können und so dem OpenCL-Standard konform zu werden, werden die in Abschnitt 1.5 genannten Compiler LLVM und SPIRV-LLVM verwendet, um den OpenCL C-Code in LLVM IR oder SPIR-V umzuwandeln. Wie bereits in Abschnitt 1.5 erwähnt, übernehmen diese Compiler die komplette Fehlerüberprüfung sowie eine Vielzahl an weiteren Aufgaben, wie Macro-Preprocessing oder manche Optimierungen. Damit der VC4CC-Compiler LLVM oder SPIRV-LLVM für die Vorverarbeitung verwenden kann, müssen deren Pfade bei der Konfiguration von VC4CC gesetzt werden.

Der grundlegende Vorgang der Vorkompilierung von OpenCL C-Quellcode ist für beide unterstützten Zwischensprachen gleich: Wenn der Compiler erkannt hat, dass es sich bei den Eingabedaten um OpenCL C-Quellcode handelt, wird der für die jeweilige Zwischensprache zuständige Compiler mit einer Vielzahl an Kommandozeilenparameter in einem Kindprozess gestartet, der OpenCL C-Quellcode in die Standardeingabe geschrieben und das Ergebnis aus der Standardausgabe gelesen. Ebenso wird die Standard-Fehlerausgabe überwacht, um Fehler in der Vorkompilierung an den Aufrufer weitergeben und die weitere Kompilierung abbrechen zu können. Als Kommandozeilenparameter wird unter anderem spezifiziert, dass es sich um eine 32-Bit Zielarchitektur handelt (um z. B. 32-Bit breite Zeigertypen zu erzeugen) und dass es sich um OpenCL C-Quellcode handelt, der mit der OpenCL C-Standardversion 1.2 kompiliert werden soll. Ebenso werden die meisten Optimierungen aktiviert, um diese nicht ebenfalls im VC4CC-Compiler implementieren zu müssen. Zusätzlich wird die Implementierung der OpenCL-Standardbibliothek aus Kapitel 3 in Form externer Header angegeben, damit diese bei der Vorkompilierung mit überprüft, eingebunden und optimiert wird. Zum Schluss wird noch definiert, dass der Quellcode als LLVM IR-Zwischencode wieder ausgegeben wird, also in keinen Maschinencode für irgendeine Architektur umgewandelt werden soll. Wird der Precompiler verwendet, um SPIR-V zu erzeugen, wird in einem zweiten Kindprozess der erzeugte LLVM IR-Code über das in SPIRV-LLVM mitgelieferte Programm `llvm-spirv` in SPIR-V-Code umgewandelt.

Wie bereits in Abschnitt 1.5.1 beschrieben, würde eine so erzeugter LLVM IR-Zwischencode eine sehr große Größe besitzen, da alle in allen eingebundenen Quelldateien enthaltenen Funktionen aufgelistet werden. Dies ist für die Verwendung von SPIR-V nicht der Fall, da bei der Umwandlung in SPIR-V alle nicht von den Kernel-Funktionen verwendeten Funktionen aussortiert werden. Jedoch nimmt in beiden Fällen die Vorkompilierung von OpenCL C-Quellcode in eine der unterstützten Zwischensprachen sehr viel Zeit in Anspruch. Die meiste Zeit davon (ungefähr 98% der Vorkompilierungszeit) wird im CLang-Frontend beim Einlesen der Quelldateien (mitsamt der VC4CL-Standardbibliothek) und vor allem bei der Makro-Verarbeitung

verbracht. Die Ausführungszeiten der einzelnen Schritte des LLVM-Compilers können mit dem Kommandozeilenparameter `-time` ausgegeben werden. Grund für diese lange Dauer ist es, dass die VC4CL-Standardbibliothek sehr viele Makros verwendet, um die meisten Funktionen automatisch zu generieren (siehe auch Abschnitt 3.2), die bei jeder Kompilierung von OpenCL C-Quellcode neu verarbeitet werden müssen. Abhilfe schafft das Konzept der „Precompiled Headers" [LLV17c], das es erlaubt, häufig verwendete Header-Dateien (wie die Implementierung der OpenCL C-Standardbibliothek) vorher zu kompilieren und als Compiler-interne Repräsentation des AST (Abstrakten Syntaxbaums) abzuspeichern. Diese Binärdatei kann dann anstatt der originalen Header-Dateien verwendet werden, um den Kompilierungsvorgang erheblich zu beschleunigen. Zusätzlich wird die ca. 13 MB große vorkompilierte Headerdatei nicht komplett eingelesen, sondern jeweils nur die benötigten Abschnitte. Dies lässt sich mit dem LLVM-Kommandozeilenparameter `-print-stats` zeigen, der statistische Informationen über die Kompilierung ausgibt. Listing 2.1 zeigt ein Beispiel der ausgegebenen Statistiken, beim dem nur 7% aller Identifier und 35% der Funktionsdeklarationen sowie kein einziges Makro aus der vokompilierten Headerdatei verwendet und somit überhaupt gelesen wird. Den Nutzen kann man auch an der Ausführungszeit des Precompilers sehen, die am Beispiel des Kernels zur Berechnung der Fibonaccizahlen aus Anhang A.1 von ungefähr 22 Sekunden auf ca. 500 ms verkürzt wurde.

```
*** AST File Statistics:
  1/220765 source location entries read (0.000453%)
  984/4660 types read (21.115879%)
  15651/44380 declarations read (35.265888%)
  217/2785 identifiers read (7.791741%)
  0/519 macros read (0.000000%)
  0/371988 statements read (0.000000%)
  0/519 macros read (0.000000%)
  0/5874 lexical declcontexts read (0.000000%)
  0/322 visible declcontexts read (0.000000%)
  217 / 225 identifier table lookups succeeded (96.444444%)
```

Listing 2.1: Analyse der verwendeten Elemente der vorkompilierten Standardbibliothek (Ausschnitt)

2.1.2 LLVM IR-Parser

Das LLVM IR-Frontend besteht, wie traditionsgemäß die Frontends der meisten Compiler, die textbasierten Quellcode (wie LLVM IR oder OpenCL C-Code) akzeptieren, aus einem Scanner und einem Parser. Der Scanner, der sich in `llvm/Scanner.cpp` befindet, ist sehr einfach gehalten und hat die Aufgabe, die Eingabe, die aus grafischen Charakteren besteht, in Token umzuwandeln. Diese Umwand-

lung geschieht Stück für Stück, d. h. der Parser fordert immer wieder ein neues Token an und der Scanner liest die Eingabe nur soweit, wie es nötig ist, dieses Token zu erstellen. Ein Token stellt einen Wert unterschiedlicher Typen dar. So können ganze und Fließkommazahlen, boolesche Konstanten und Zeichenketten (Strings) dargestellt werden. Wie in den meisten Compilern, behandelt der Scanner bereits Kommentare, indem diese einfach übersprungen werden und vordefinierte Schlüsselwörter (z. B. `define` zum Definieren einer Funktion) werden nicht gesondert behandelt, sondern als Token mit einer Zeichenfolge zurückgegeben, sodass der Parser diese passend interpretieren muss. Der Parser (`llvm/Parser.cpp`) übernimmt die hauptsächliche Arbeit des LLVM IR-Frontends, indem er, solange bis das Ende der Eingabe erreicht ist, das nächste Token liest und es im aktuellen Kontext interpretiert (z. B. als Schlüsselwort, Funktions- oder Variablenname) um somit die Struktur des Programms aufbauen zu können. Um die Grammatik der LLVM IR richtig analysieren und umwandeln zu können, wird die sehr umfangreiche offizielle Dokumentation der LLVM IR auf der offiziellen Website von LLVM verwendet [LLV17b].

Das LLVM IR-Frontend bietet mehrere Vor- und auch Nachteile im Vergleich zum SPIR-V-Frontend (siehe Abschnitt 2.1.3): Ein großer Vorteil ist, dass es sowohl mit SPIRV-LLVM (siehe Abschnitt 1.5.2) als auch der „normalen" LLVM-Kompilersuite (siehe Abschnitt 1.5.1), deren erzeugter Code sich leicht unterscheidet, verwendet werden kann. Vor allem die Verwendung mit der offiziellen LLVM-Version birgt den Vorteil, dass diese bereits in den Standard-Repositories des Raspbian Betriebssystems vorhanden ist und somit einfach installiert werden kann, während der SPIRV-LLVM Compiler erst aus dem Quellcode kompiliert werden muss, was auf den rechenschwachen Raspberry Pi-Modellen einige Zeit dauern kann. Jedoch wird, wie bereits beschrieben, im LLVM IR-Frontend textueller Code manuell eingelesen und verarbeitet, wodurch das Frontend fehleranfälliger und auch anfälliger gegenüber Änderungen im erzeugten LLVM IR-Zwischencode wird (z. B. durch neue LLVM-Versionen). Ebenso besitzt durch das textuelle Verarbeiten das LLVM IR-Frontend eine etwas schlechtere Performance als das SPIR-V-Frontend aus dem nächsten Abschnitt, weswegen man, wo möglich, die Verwendung des SPIR-V-Frontends bevorzugen sollte.

2.1.3 SPIR-V-Frontend

Das SPIR-V-Frontend ist einfacher aufgebaut als das LLVM IR-Frontend, in dem Sinne, dass kein Scanner und Parser implementiert werden musste. Stattdessen wird eine von Khronos entwickelte Bibliothek namens **SPIR-V Tools** [Gro17c] verwendet. SPIR-V Tools ist eine Werkzeugsammlung zum Einlesen, Ausgeben, Umwandeln und Validieren von SPIR-V im binären oder textuellem Format. Im SPIR-V-Frontend wird SPIR-V Tools verwendet, um den SPIR-V-Quellcode einzulesen. SPIR-V Tools

liest eine Instruktion nach der anderen aus der Eingabe und übergibt diese als Objekt eines vorgegebenen Typs an eine Callback-Methode, die die Instruktionen abhängig ihrer Art weiterverarbeitet. So wird z. B. für Typen sowie für Konstanten jeweils eine globale Tabelle erstellt, die die jeweilige ID dem gegebenen Typen oder der definierten Konstanten zuweist. Zusätzliche Informationen, wie die Namen für Kernel oder Parameter, werden ebenso je ID abgespeichert, um dann im weiteren Verlauf an das Objekt, welches für die gegebene ID erzeugt wird, übergeben werden zu können. SPIR-V Tools gibt die Instruktionen als linearen Strom aus, für die interne Verarbeitung wird aber eine Hierarchieebene (Funktion > Instruktionen einer Funktion) benötigt. Deshalb speichert das SPIR-V-Frontend beim Opcode `Function` die aktuelle Funktion und fügt alle folgenden Instruktionen, bis zu dem Opcode `FunctionEnd` dieser Funktion hinzu. Ebenso wird für jede Instruktion die Ergebnis-ID mit der Typ-ID des Ergebnisses verknüpft, um im nächsten Schritt die IDs mit den richtigen Typen assoziieren zu können. Über den Opcode `EntryPoint` (siehe Anhang A.3, Zeile 6) werden Funktionen bestimmt, die Einstiegspunkte sind, also Shader für OpenGL oder Kernel-Funktionen für OpenCL. Diese Information, sowie der Name des OpenCL-Kernels wird ebenso in der jeweiligen Funktion gespeichert.

In einem zweiten Durchlauf über die Instruktionen der generierten Funktionen werden diese Instruktionen in die interne Repräsentation konvertiert, die dann auch in den Abschnitten 2.2 für die Umformungen und Optimierungen sowie 2.3 für die Code-Generierung verwendet wird. Hierfür wird jeder Instruktion die vorher zusammengestellten Tabellen der Typen, Konstanten, ID-zu-Typ-Assoziationen und globalen Daten übergeben, aus denen die Operanden- und Ergebniswerte und deren Typen ermittelt werden. Die Aufteilung des Frontends in zwei Schritte ist nötig, da aufgrund von Rückwärtssprüngen (in Kombination mit Phi-Nodes, siehe Abschnitt 2.2.2) oder Funktionsaufrufen von Funktionen, die in der Eingabedatei weiter hinten definiert werden, Instruktionen auf IDs zugreifen können, die noch nicht definiert sind. Jedoch muss in der internen Repräsentation jeder Wert und dessen Typ klar definiert sein.

Auch das SPIR-V-Frontend hat Vor- und Nachteile gegenüber dem LLVM IR-Frontend aus Abschnitt 2.1.2: Zum einen ist es, wie bereits erwähnt, performanter. Ebenso ist es vollständiger, d. h. es unterstützt mehr der möglichen Befehle. Da die verwendete SPIRV-Tools-Bibliothek von Khronos selbst gewartet und aktiv weiterentwickelt wird, wird auch eine Kompatibilität mit der neusten SPIR-V-Version, sowie der meisten offiziellen Erweiterungen, garantiert. SPIR-V unterstützt neben OpenCL noch weitere andere Quellsprachen wie OpenGL und das relativ neue Vulkan, was zukünftige Weiterentwicklungen der Zwischensprache verspricht. Jedoch benötigt die Verwendung des SPIR-V-Frontends einen erhöhten Einrichtungsaufwand, da der SPIRV-LLVM-Compiler manuell installiert werden muss. Ebenso ist

das Tool `llvm-spirv` – Teil des SPIRV-LLVM-Compilers – nicht vollständig, d. h. es kann nicht alle gültigen LLVM IR-Instruktionen erfolgreich in SPIR-V-Instruktionen umwandeln. So bringen gewisse Instruktionen, wie `llvm.bswap` oder Structs mit Vektoren als Inhalt, das Programm zum Abstürzen. In solchen Fällen muss dann auf das LLVM IR-Frontend zurückgegriffen werden, das diese Instruktionen – zumindest teilweise – unterstützt. Dies geschieht automatisch, d. h. der VC4CC Compiler kompiliert einen OpenCL C-Quellcode über das LLVM IR-Frontend, wenn die Kompilierung über das SPIR-V-Frontend fehlschlägt.

2.2 Middleend

Wie bereits in Abschnitt 2.1 erwähnt, ist es die Aufgabe des Frontends, den eingelesenen Quellcode auf syntaktische und semantische Fehler hin zu überprüfen und in eine interne Repräsentation umzuwandeln. Das Middleend (eine Bezeichnung des mittleren Segments zwischen dem Frontend und dem Backend eines Compilers) führt eine Vielzahl von Umformungen auf der internen Repräsentation aus. Zu diesen Umformungen gehören nötige Schritte, um z. B. von der Zielarchitektur nicht unterstützte Instruktionen auf unterstützte Instruktionen mit der gleichen Semantik umzuwandeln. Darunter zählen das Inlining in Abschnitt 2.2.1, die Phi-Elimination in Abschnitt 2.2.2, die Behandlung von intrinsischen Funktionen und nicht-unterstützter Operatoren (Abschnitte 2.2.3 und 2.2.4). Ein weiterer Teil des Middleends dient der reinen Optimierung, d. h. die Instruktionsfolge vor und nach der Anwendung eines Optimierungsschrittes ist semantisch gleichbedeutend. Ebenso können beide Befehlsabläufe (vor und nach einer Optimierung) auf der Zielhardware ausgeführt werden, jedoch löst die optimierte Instruktionsfolge ihre Aufgabe „besser", indem sie z. B. weniger Instruktionen benötigt oder die Registerzuordnung (siehe Abschnitt 2.3.2) erleichtert. Zu den Optimierungsschritten zählen unter anderem die Optimierung der Speicherzugriffe sowie das Umsortieren und das Kombinieren von Instruktionen (Abschnitte 2.2.5, 2.2.6 und 2.2.7). Die Umformungen geschehen schrittweise, d. h. jede Umformung wird für alle (zutreffenden) Instruktionen der derzeit behandelten Funktion durchgeführt, bevor die nächste Umformung gestartet wird. In den folgenden Abschnitten werden eine Auswahl an interessanter Umformungen näher beschrieben. Die weiteren Umformungen (die alle von der Quelldatei `optimization/Optimizer.cpp` aus gestartet werden) beinhalten z. B. das Eliminieren von Sprüngen auf die nächste Instruktion, oder das Vereinfachen von bestimmten Operationen (wie die Umwandlung von `out = or in, 0` nach `out = in`) und sind meist trivial gelöst. Die in diesem Abschnitt beschriebenen die Umformungs- und Optimierungsschritte werden in der gleichen Reihenfolge aufgeführt, in der sie vom Compiler abgearbeitet werden.

2.2.1 Inlining

Inlining ist der Vorgang, einen Funktionsaufruf durch den Inhalt der aufgerufenen Funktion zu ersetzen, sodass die Ausführung des Inhalts einer aufgerufenen Funktion semantisch äquivalent zu dem originellen Funktionsaufruf ist. Viele Compiler bieten Inlining als Optimierung an, da es den Overhead eines Funktionsaufrufs (z. B. Anlegen eines neuen Stack-Frames und Sichern der Register für Stack-basierte Architekturen, sowie Sprünge im Code und somit eventuelle Cache-Misses) beseitigt. Wenn an mehreren Stellen Funktionsaufrufe zu den gleichen Funktionen in die aufrufende Funktion eingefügt werden, erhöht sich jedoch die Größe des Codes, weshalb die meisten Compiler, auch bei aktivierter Inline-Optimierung, nicht alle Funktionsaufrufe ersetzen. Ein großes Problem stellen hierfür rekursive Funktionen dar, bei denen die Größe des Codes sehr schnell sehr viel größer wird oder, wenn der Compiler das Abbruchkriterium nicht erkennen kann, sogar ins unendliche steigen kann. Daher werden meist auch rekursive Funktionen nicht oder nur teilweise (z. B. eine gewisse Anzahl an Rekursionsschritten) ersetzt.

In diesem OpenCL C-Compiler ist die Inline-Optimierung ein essenzieller Schritt und kann auch nicht deaktiviert werden. Stattdessen wird Exhaustive Inlining (vollständiges Inlining) ausgeführt, was bedeutet, dass alle Funktionsaufrufe durch die Ausführung des Inhalts der aufgerufenen Funktion ersetzt werden. Dies wird ermöglicht, da der OpenCL-Standard rekursive Funktionen in der OpenCL C-Sprache für Kernels verbietet [Gro12, Abschnitt 6.9]. Exhaustive Inlining vereinfacht die weitere Behandlung des Codes an mehreren Stellen. Zum einen wird bei der Ausführung kein Stack benötigt, da es nur eine aufgerufene Funktion und damit nur ein „Stack-Frame" gibt. Das ist auch der Hauptgrund für das Exhaustive Inlining, da die Verwendung eines Stacks einen effizienten Zugriff auf einen RAM voraussetzt, was für die VideoCore IV GPU nicht gegeben ist (siehe Abschnitt 1.3). Des Weiteren wird die Register-Zuordnung (siehe Abschnitt 2.3.2) stark vereinfacht, da bei Funktionsaufrufen nicht mehr darauf geachtet werden muss, welche Register von der aufgerufenen Funktion verwendet werden und auch keine Register auf dem Hauptspeicher gesichert werden müssen (ähnlich einem Stack-Frame). Der dritte Vorteil ist, dass nach dem Inlining aller Funktionen in die aufrufende Kernel-Funktion nur noch die Kernel-Funktionen beachtet werden müssen. Da es der OpenCL-Standard ebenso verbietet, Zeiger auf Funktionen zu erstellen [Gro12, Abschnitt 6.9], und es keine Funktionsaufrufe mehr gibt, können alle nicht-Kernelfunktionen verworfen werden.

Ein Teil des Inlinings wird bereits in der Vorkompilierung (siehe Abschnitt 2.1.1) durch das Anschalten der Optimierungen mit dem Compilerflag `-O3` ausgeführt. Hier gilt jedoch, wie bereits beschrieben, dass der jeweilige Compiler selbst entscheidet, welche Funktionen er in den Aufrufer einsetzt und welche nicht. Die Funktionalität des vollständigen Inlinings der restlichen Funktionsaufrufe befin-

det sich in der Datei `optimizations/Inliner.cpp` und besteht aus folgenden Schritten:

1. Finden der innersten Funktion
2. Einsetzen in Funktionsaufruf
   (a) Kopieren der tatsächlichen Parameter in die formalen Parameter
   (b) Kopieren der Parameter und lokalen Variablen in die aufrufende Funktion
   (c) Kopieren der Instruktionen in die aufrufende Funktion
3. Funktionsaufruf durch Sprungziel ersetzen

(1) Je Kernel-Funktion werden, beginnend mit dieser, alle Funktionsaufrufe gesucht. Wird ein Funktionsaufruf gefunden, wird die aufgerufene Funktion betreten und von dort aus rekursiv weiter gesucht, bis eine Funktion gefunden wird, die keine Funktionsaufrufe mehr besitzt. (2) Diese Funktion wird an die Stelle des vorher gefundenen Funktionsaufrufs eingesetzt. Dafür werden folgende Schritte ausgeführt: (2.a) Je Parameter des Funktionsaufrufs wird in die aufrufende Funktion eine `move`-Instruktion eingefügt, die der den tatsächlichen Parameterwert des Aufrufs (z. B. ein Register, oder eine Konstante) in den formalen Parameter der aufgerufenen Funktion kopiert. (2.b) Die Definitionen der Parameter und der lokalen Variablen der aufgerufenen Funktion werden in die Liste der lokalen Variablen der aufrufenden Funktion kopiert. Wie auch bei Schritt (2.a) werden die Parameter und lokalen Variablen der aufgerufenen Funktion umbenannt, um innerhalb des Kernel eindeutig zu sein (siehe Listing 2.3). (2.c) Die Instruktionen der aufgerufenen Funktion werden in die aufrufende Funktion eingefügt. Hier werden `return`-Instruktionen besonders behandelt: Wenn die aufgerufene Funktion einen Wert zurückgibt, wird für die `return`-Instruktion eine `move`-Instruktion erzeugt, die (umgekehrt zu Schritt (2.a)) den Rückgabewert in die Zielvariable des Funktionsaufrufs kopiert. Ebenso werden für alle `return`-Instruktionen ein Sprung hinter den Funktionsaufruf erstellt. (3) Da die Instruktionen der aufgerufenen Funktion vor dem Funktionsaufruf eingefügt worden sind, kann im letzten Schritt der originale Funktionsaufruf durch ein Sprungziel ersetzt werden, dass das Verlassen der inneren Funktion (z. B. durch `return`) darstellt. Es wird ab Schritt (1) wiederholt, solange, bis die aktuelle Funktion die Kernel-Funktion ist und diese keine Funktionsaufrufe mehr besitzt.

Listings 2.2 und 2.3 zeigen ein Beispiel eines Inline-Vorgangs einer Inkrementierfunktion. Zur besseren Lesbarkeit wurde für das Beispiel LLVM IR anstatt der internen Repräsentation gewählt, obwohl die Inline-Optimierung auf dem Code in der internen Repräsentation durchgeführt wird. In Listing 2.3 kann man gut erkennen, wie die lokalen Variablen umbenannt werden (siehe Zeile 2). Indem das

```
1 define i32 @increment(i32 %val) {
2   %1 = add i32 %val, i32 1
3   ret i32 %1
4 }
5
6 define void @kernel(i32 %arg) {
7   %1 = call i32 @increment(i32 %arg)
8   ; [...]
9 }
```

Listing 2.2: Beispiel eines Inlinings in LLVM IR (vorher)

```
1 define void @kernel(i32 %arg) {
2   %1.%val = mov i32 %arg    ; Copy real parameter into
     formal parameter (2.a)
3   %1.%1 = add i32 %1.%val, i32 1  ; Insert instructions in
     caller (2.c)
4   %1 = mov i32 %1.%1  ; Replace return with move (2.c)
5   br label %1.after  ; Insert branch after return (2.c)
6 %1.after:  ; Replace function-call with label (3)
7   ; [...]
8 }
```

Listing 2.3: Beispiel eines Inlinings in LLVM IR (nachher)

Ziel des Funktionsaufrufs (die Variable `%1`) als Präfix an den Variablennamen angehängt wird, wird die Eindeutigkeit jedes geänderten Variablennamen garantiert. Bei Funktionen ohne Rückgabewert wird stattdessen der Funktionsname und eine Zufallszahl als eindeutiges Präfix verwendet. Das Ergebnis in Listing 2.3 besitzt noch überflüssige Instruktionen (wie das Kopieren von `%arg` in Zeile 2, sowie der Sprung auf die nächste Instruktion in Zeile 5). Diese werden jedoch in den weiteren Optimierungen entfernt, sodass am Ende nur noch eine Instruktion übrig bleibt: `%1 = add i32 %arg, i32 1`.

2.2.2 Phi Elimination

Die SSA-Form der Zwischensprachen LLVM IR und SPIR-V (siehe Abschnitte 1.5.1 und 1.5.2) hat viele Vorteile für Compiler-Optimierungen, wie z. B. das schnelle Erkennen ungenutzter Werte, Konstanten mit gleichen Werten oder doppelte gleichwertige Berechnungen. Jedoch führt SSA ein Problem ein, nämlich den Umgang mit Variablen, die in alternativen Ausführungszweigen zugewiesen werden. Listing 2.4 zeigt ein Beispiel für einen solchen Code. Aufgrund der SSA-Form müssen z. B. in LLVM IR die beiden Zuweisungen in eine zusammengefasst werden. Für einfache Fälle bietet sowohl LLVM IR als auch SPIR-V eine `select`-Instruktion,

die abhängig einer Bedingung dem Ziel verschiedene Werte zuweisen kann (ähnlich dem Ternären Operator `?:`). Für komplexere Fälle (wie in Listing 2.4) gibt es hier die sog. Phi-Funktion. Diese weist dem Ziel einen Wert zu, je nachdem von welchem Basisblock zu der Phi-Funktion gesprungen wurde, d. h. welcher Basisblock vor dem Basisblock der Phi-Funktion ausgeführt wurde. Die Phi-Funktion für das Code-Beispiel aus Listing 2.4 in LLVM IR kann in Listing 2.5 gesehen werden. Hierbei stellen die Paare in den eckigen Klammern jeweils der zugewiesene Wert und das Sprungziel des vorhergehenden Basisblocks dar und die Phi-Funktion in Zeile 11 weist dem Ergebnis den Wert `true` zu, wenn der bedingte Sprung in Zeile 4 direkt zum Sprungziel `%11` gesprungen ist, ansonsten den Wert von `%10`, falls der Basisblock `%8` ausgeführt wurde. [Wik17g]

```
1 void func(int r, int y) {
2     int l = y || r;
3 }
```

Listing 2.4: Beispiel der Zuweisung einer Variable aus alternativen Ausführungszweigen

```
1  define void @func(i32, i32) {
2  %2:
3    ; [...]
4    br i1 %7, label %11, label %8
5
6  %8:
7    ; [...]
8    %10 = icmp ne i32 %9, 0
9
10 %11:
11   %12 = phi i1 [ true, %2 ], [ %10, %8 ]
12   ; [...]
13 }
```

Listing 2.5: LLVM IR-Code zu Listing 2.4 (Ausschnitt)

Da Phi-Funktionen nicht im Befehlssatz der VideoCore IV-Architektur (siehe Abschnitt 1.3) vorhanden sind, müssen diese in eine Reihe anderer Operationen umgewandelt werden. Dies geschieht im Optimierungsschritt „Phi Elimination" und in der Datei `optimization/Eliminator.cpp`: Als Erstes werden alle Phi-Funktion innerhalb eines Kernels gesucht. Für jede Phi-Funktion wird für jeden vorhergehenden Basisblock eine `mov`-Instruktion des jeweiligen Wertes zu dem Ziel der Phi-Funktion erstellt und die Phi-Funktion gelöscht. Für das Beispiel aus Listing 2.5 würden folgende Zuordnungen erstellt werden: `%2` zu `%12 = mov i1 true` und `%8` zu `%12 = mov i1 %10`. Diese Zuordnungen werden in einer Liste gesammelt.

Im nächsten Schritt werden für alle Basisblöcke, für die es Zuordnungen gibt, diese an das Ende des Basisblocks angefügt. Als Ergebnis werden, anstatt der Zuweisung einer Variable über Phi-Funktionen abhängig vom vorherigen Basisblock, die Zielvariablen am Ende der vorherigen Basisblöcke zugewiesen (siehe Listing 2.6). Ab diesem Zeitpunkt besitzt der Code keine SSA-Form mehr, was jedoch für diesen Compiler keine größeren Auswirkungen hat. Diese Vorgehensweise wird auch in den Backends von LLVM verwendet und kann dort in der Datei `lib/CodeGen/PHIElimination.cpp` eingesehen werden [Gro17d].

```
1  define void @func(i32, i32) {
2  %2:
3    ; [...]
4    %12 = mov i1 true
5    br i1 %7, label %11, label %8
6
7  %8:
8    ; [...]
9    %10 = icmp ne i32 %9, 0
10   %12 = mov i1 %10
11
12 %11:
13   ; [...]
14 }
```

Listing 2.6: Ergebnis der Phi-Elimination für das Beispiel aus Listing 2.4 (Ausschnitt)

2.2.3 Intrinsic Functions

Intrinsische Funktionen (vom englischen „intrinsic functions", was „innere Funktionen" bedeutet), oft auch built-ins („eingebaute Funktionen") genannt, sind spezielle Funktionen höherer Programmiersprachen, deren Implementierung meist nur dem Compiler bekannt ist und dieser zur Kompilierzeit die Funktionsaufrufe auf intrinsische Funktionen durch die Implementierung ersetzt. Solche Funktionen werden oft verwendet, um Maschinenbefehle direkt für die höhere Sprache verfügbar zu machen, z. B. für stark optimierten Code. Ein anderer Anwendungsfall sind grundlegende mathematische Routinen (wie die Potenzfunktion oder den Logarithmus), die je nach Zielarchitektur mit anderen Code implementiert werden, um die beste Performance zu erreichen. [MSD17]

Auch der VC4CC-Compiler unterstützt eine Vielzahl an intrinsischen Funktionen. Diese sind in der Implementierung der OpenCL-Standardbibliothek (siehe Kapitel 3) in der Header-Datei `_intrinsics.h` definiert und werden im Compiler in einem speziellen Optimierungsschritt in der Datei `intrinsics/Intrinsics.cpp`

durch die jeweilige Implementierung ersetzt:

Die meisten intrinsischen Funktionen werden direkt auf einzelne (oder wenige) ALU-Operationen abgebildet. So gibt es unter anderem intrinsische Funktionen für die ALU-Operationen `fmin/fmax`, `min/max` und `shr/shl`. Für diese Art von Intrinsics wird der Funktionsaufruf direkt durch eine ALU-Operation ersetzt und die Funktionsparameter als Operanden verwendet.

Eine weitere Gruppe von intrinsische Funktionen stellen die Aufrufe der SFU (Special Functions Unit) dar. Wie bereits in Abschnitt 1.3.1 beschrieben, werden Berechnungen über die SFU durchgeführt, indem der Operand (z. B. die Zahl, aus der die Quadratwurzel berechnet werden soll) in ein spezielles Register (`sfu_recip`, `sfu_rsqrt`, `sfu_exp` oder `sfu_log`) geschrieben wird und drei Zyklen später das Ergebnis aus einem anderen speziellen Register `r4` gelesen werden kann. Die SFU-Intrinsics werden daher durch ein Kopieren des Operanden in das entsprechende SFU-Register, gefolgt von zwei `nop`s (also Instruktionen, die nichts tun) sowie eine einem `mov` zum Kopieren des Ergebnisses aus `r4` in das Ziel des Funktionsaufrufs ersetzt.

Weitere intrinsische Funktionen bieten Zugriff auf die Peripherie innerhalb der VideoCore IV GPU (siehe Abschnitt 1.3.1) sowie nicht-synchronisierten Zugriff auf den Arbeitsspeichern. Für die Inkrementierung und Dekrementierung der verfügbaren Hardware-Semaphoren gibt es jeweils eine Funktion `vc4cl_semaphore_increment` und `vc4cl_semaphore_decrement`. Diese nehmen das zu verändernde Semaphore (0 bis 15) als konstanten Parameter entgegen und werden vom Compiler durch die jeweilige Instruktion zum In- oder Dekrementieren konvertiert (siehe Abschnitt 1.3.2). Das einzige Hardware-Mutex der VideoCore IV GPU kann über ein spezielles Mutex-Register gesperrt und freigegeben werden. Dafür existieren die Funktionen `vc4cl_mutex_lock` und `vc4cl_mutex_unlock`, die in ALU-Operationen zum Schreiben des Mutex-Registers (für das Sperren) und Lesen aus dem Mutex-Register (für das Entsperren) umgewandelt werden. Da sich alle QPUs die Hardware für den Speicherzugriff (VDW, VCD, VPM, siehe Abbildung 1.2) teilen, muss der Zugriff auf den Arbeitsspeicher bei jedem Lesen oder Schreiben synchronisiert werden. Dafür wird automatisch jeder Speicherzugriff mit dem Hardware-Mutex gesichert, indem dieses vor dem Zugriff gesperrt und danach wieder freigegeben wird. Ansonsten könnte es vorkommen, dass eine QPU auf eine Zieladresse schreibt, die zwischenzeitlich von einer anderen QPU gesetzt worden ist, und somit sich falsch verhält. Um jedoch für besondere Fälle (z. B. atomaren Speicherzugriff, siehe Abschnitt 3.2.2) unsynchronisierten Zugriff auf den Arbeitsspeicher zu bieten, gibt es die intrinsischen Funktionen `vc4cl_dma_read` und `vc4cl_dma_write` die auf die gleichen Instruktionen wie der „normale" synchronisierte Speicherzugriff abgebildet werden, jedoch ohne das Hinzufügen der Mutex-Operationen. Diese müssen, um trotzdem die Richtigkeit des ausgeführ-

ten Codes sicherzustellen, manuell hinzugefügt werden, können jedoch jetzt noch andere Instruktionen mit einschließen.

Die OpenCL-Standardbibliothek (siehe Abschnitt 1.2.3 und Kapitel 3) bietet besondere Funktionen, mit denen jedes Work-Item (einzelne parallele Ausführung eines Kernels) auf Informationen über sich selber und die übergeordnete Work-Group (Gruppe von parallelen Ausführungen eines Kernels) zugreifen kann. Darunter fallen die Größe der Work-Group, die Anzahl der Dimensionen sowie der Work-Groups und die eindeutige ID des ausgeführten Work-Items (siehe [Gro12, Abschnitt 6.12.1]). Da diese Informationen erst beim Starten des Kernels und der Erstellung der Work-Group und der Work-Items bekannt wird, müssen sie bei der Ausführung der Work-Items als Parameter übergeben werden (siehe Abschnitt 2.3.1). Diese Parameter werden erst vom Compiler-Backend hinzugefügt, weshalb auch das Auslesen dieser Werte erst vom Compiler hinzugefügt werden kann. Ansonsten würde die Vorverarbeitung (Abschnitt 2.1.1) den Quellcode als fehlerhaft erkennen, aufgrund nicht definierter Werte. Die Implementierung der intrinsischen Funktionen zum Auslesen der Work-Item- und Work-Group-Information besteht aus einem einfachen Kopieren der Parameter in das Ziel der intrinsischen Funktion.

Weitere intrinsische Funktionen werden verwendet, um im OpenCL C-Standard illegale Typkonvertierungen durchzuführen oder die Implementierung von Code zu verstecken, der sich in OpenCL C nicht darstellen lässt. OpenCL erlaubt zwar, wie auch in C, skalare Zahlenwerte in andere Datentypen zu konvertieren (z. B. `int i; char c = i`). Jedoch ist die Konvertierung von Vektortypen verboten und muss über spezielle Funktionen der Standardbibliothek durchgeführt werden. Um diese Funktionen jedoch implementieren zu können, wird z. B. eine Konvertierung `int4 i; char4 c = i` benötigt. Diese wird in einer intrinsischen Funktion versteckt, wodurch ein OpenCL C-Compiler zwar den Funktionsaufruf überprüfen kann und trotzdem später vom VC4CC-Compiler ein optimierter Code erzeugt werden kann, der sich in OpenCL C nicht (effizient) darstellen lässt.

2.2.4 Operatoren

Wie bereits in Abschnitt 1.3.2 beschrieben, gibt es eine Reihe an wichtigen arithmetischen Operatoren, die nicht direkt auf ALU-Instruktionen abgebildet werden können. Diese müssen durch Kombination Hardware-unterstützter Instruktionen implementiert werden, was in der Datei `intrinsics/Operators.cpp` programmiert ist und als Teil des Intrinsivierungsschritt (Abschnitt 2.2.3) durchgeführt wird. Unter den in Software implementierten Operatoren befinden sich die 32-Bit Ganzzahlmultiplikation, die 32-Bit vorzeichenlose und vorzeichenbehaftete Division mit und ohne Berechnung des Rests sowie die 32-Bit Fließkommadivision. Für alle dieser Operationen wird vor der Verwendung der allgemeinen Lösung ausprobiert,

ob optimierte Berechnungen für spezielle Kombinationen von Operanden angewendet werden können. So werden z. B. Operationen mit zwei konstanten Werten im Compiler berechnet und das Ergebnis in die Zielvariable geschrieben. Ebenso werden Ganzzahlmultiplikationen oder -divisionen mit konstanten Zweierpotenzen durch Shifts ersetzt. Dies ist möglich aufgrund der Binärdarstellung, wodurch $(0011)_2 * 2^2 = (1100)_2$ ergibt, d. h. eine Multiplikation mit 4 entspricht einer Verschiebung aller Bits um 2 (der Exponent der Zweierpotenz 2^2) nach links. Durch diese Optimierung können die speziellen Fälle der Multiplikation oder Division mit konstanten Zweierpotenzen durch eine einzelne Hardwareinstruktion ersetzt werden. Ebenso kann die Ganzzahlmultiplikation direkt durch die unterstützte 24-Bit Ganzzahlmultiplikation ersetzt werden, wenn die Typen beider Operanden weniger als 24 Bit breit sind. Dies gilt z. B. für die Datentypen `short` (16 Bit) und `char` (8 Bit). Für die Fließkommadivision kann die Optimierung verwendet werden, dass eine Division durch eine beliebige konstante Zahl mit der Multiplikation der Inverse dieser Zahl ersetzt werden kann, die wiederum als Maschinenbefehl verfügbar ist ($x/y = x * (1/y)$). Es kann auch die SFU verwendet werden, um das Inverse einer Zahl zur Laufzeit zu berechnen (siehe Abschnitt 1.3.1). Diese Optimierung benötigt vier Taktzyklen (SFU-Aufruf, zwei Wartezyklen und die Fließkommamultiplikation), kann jedoch nur angewendet werden, wenn die zu ersetzende Division die Verwendung von ungenauen Verfahren erlaubt, z. B. durch ein dementsprechendes Attribut.

Die allgemeine Implementierung der vorzeichenlosen Ganzzahlmultiplikation macht sich die bereits genannte Eigenschaften ganzer Zahlen zunutze, dass eine Multiplikation oder Division mit einer Zweierpotenz als einzelne Shift-Maschinenbefehle abgebildet werden kann, genauso wie die Eigenschaft des Distributivgesetzes ($a * (b + c) = a * b + a * c$). Des Weiteren besitzt die VideoCore IV-Architektur den Maschinenbefehl `mul24`, der zwei 24-Bit Ganzzahlen multiplizieren und als Ergebnis eine 32-Bit Ganzzahl zurückliefern kann. Kurz zusammengefasst werden 32-Bit breite ganze Zahlen multipliziert, indem diese in 16-Bit breite Zahlen aufgeteilt, diese miteinander multipliziert und anschließend wieder aufaddiert werden. Der Pseudocode für diese Berechnung ist in Listing 2.7 zu finden und lässt sich auch anhand einer mathematischen Formel beweisen:

$$
\begin{aligned}
res &= c11 + c10 + c01 + c00 \\
&= a_{lower} * b_{lower} + (a_{lower} * b_{upper}) * 2^{16} + (a_{upper} * b_{lower}) * 2^{16} + (a_{upper} * b_{upper} * 2^{32}) \\
&= a_{lower} * (b_{lower} + b_{upper} * 2^{16}) + a_{upper} * (b_{lower} * 2^{16} + b_{upper} * 2^{16}) * 2^{16} \\
&= a_{lower} * b + a_{upper} * b * 2^{16} \\
&= (a_{lower} + a_{upper} * 2^{16}) * b \\
&= a * b
\end{aligned}
\tag{2.1}
$$

```
1  //split operands
2  int aUpper = a >> 16; //upper 16
3  int aLower = a && 0xFFFF; //lower 16 bits
4  int bUpper = b >> 16; //uper 16 bits
5  int bLower = b && 0xFFFF; //lower 16 bits
6  //calculate partial products
7  int c11 = aLower * bLower;
8  int c10 = aLower * bUpper;
9  int c01 = aUpper * bLower;
10 int c00 = aUpper * bUpper;
11 //align partial products
12 c10 = c10 << 16;
13 c01 = c01 << 16;
14 c00 = c00 << 32;
15 //sum up parts
16 int res = c11 + c10;
17 res = res + c01;
18 res = res + c00;
```

Listing 2.7: Berechnung der 32-Bit-Ganzzahlmultiplikation

Hierbei sind x_{lower} die unteren 16 Bits (also $x \bmod 2^{16}$) und x_{upper} die oberen 16 Bits (also $x/2^{16}$), wodurch $x_{lower} + x_{upper} * 2^{16} = x \bmod 2^{16} + (x/2^{16}) * 2^{16} = x \bmod 2^{16} + (x - x \bmod 2^{16}) = x$. $(x/2^{16}) * 2^{16}$ ergibt $(x - x \bmod 2^{16})$ und nicht x, da die unteren 16 Bits bei einer Ganzzahldivision verworfen werden. Die Berechnungen in den Zeilen 10, 14 und 18 des Code-Listings 2.7 können weggelassen werden, da das Ergebnis eines nicht-zyklischen Shifts einer 32-Bit-Zahl um 32 Stellen in Zeile 14 immer null ergibt, somit der Wert aus Zeile 10 auch nie benötigt wird und in Zeile 18 immer null dazu addiert werden würden. Diese Implementierung resultiert in elf Instruktionen, die benötigt werden, um eine vorzeichenlose 32-Bit Ganzzahlmultiplikation durchzuführen. Für eine vorzeichenbehaftete Ganzzahlmultiplikation werden die Absolutwerte beider Operanden mithilfe der vorzeichenlosen Multiplikation multipliziert und das Vorzeichen des Ergebnisses wieder umgedreht, falls genau einer der Operanden negativ ist. Dadurch benötigt die allgemeine Ganzzahlmultiplikation weitere 16 Instruktionen.

Für die allgemeine vorzeichenlose Ganzzahldivision wird die sog. „Long Division" (deutsch: Schriftliche Division) implementiert, die auch für die manuelle Division mit Stift und Papier verwendet wird. Bei der schriftlichen Division wird von links beginnend stellenweise der größte ganzzahlige Vielfache des Nenners gesucht, der in den aktuell betrachteten Bereich des Zählers passt. Diese Zahl wird vom Zähler abgezogen und der ganzzahlige Faktor des Vielfachen wird an das Ergebnis angehängt. Passt in die letzten Stellen des Zählers kein ganzzahliges Vielfaches des Nenners mehr hinein, bleibt der Rest als Rest der Division stehen. Listing

2.8 beschreibt die binäre schriftliche Division in Pseudocode: Als Erstes wird die Division durch null überprüft und eine dementsprechende Handhabung eingeleitet. Daraufhin wird bitweise vom höchsten Bit des Zählers beginnend, dieses an den Rest angefügt und überprüft, ob der Rest größer als der Nenner ist. Ist der Rest größer als der Nenner, wird der Rest um den Wert des Nenners verringert und der das letzte Bit des Quotienten auf Eins gesetzt. Dies wird bis zum niedrigsten Bit des Nenners wiederholt. Am Ende steht im Quotienten `Q` der ganzzahlige Teiler und im Rest `R` der Rest der Division.

```
1  -- Calculates N divided by D, Q is the quotient and R the
       remainder
2  if D = 0 then error(DivisionByZeroException) end
3  Q := 0                  -- Initialize quotient and remainder
       to zero
4  R := 0
5  for i := n - 1 .. 0 do  -- Where n is number of bits in N
6    R := R << 1           -- Left-shift R by 1 bit
7    R(0) := N(i)          -- Set the least-significant bit of
       R equal to bit i of the numerator
8    if R >= D then
9      R := R - D
10     Q(i) := 1
11   end
12 end
```

Listing 2.8: Pseudocode der binären Long Division [Wik17d]

Es gibt eine Vielzahl an effizienteren Algorithmen für die vorzeichenlose binäre Ganzzahldivision (siehe [Rod08]), die jedoch nicht für die VideoCore IV-Architektur angewendet werden können. Manche dieser Algorithmen verwenden eine Lookup-Tabelle, um das Ergebnis der Division zweier bekannter Bitmuster nachzuschlagen anstatt auszurechnen. Aufgrund der geringen Registeranzahl würde eine solche Lookup-Tabelle aus dem Arbeitsspeicher geladen werden müssen, was wiederum sehr ineffizient ist. Andere Algorithmen (wie das Newton-Raphson-Verfahren) beruhen auf einer effizienten Ganzzahlmultiplikation, z. B. als einzelne Hardwareinstruktion, die ebenso im Befehlssatz der QPUs nicht verfügbar ist. Ebenso gibt es Algorithmen, die temporäre Register benötigen, die die doppelte Bitbreite besitzen wie der Datentyp, auf dem gerechnet wird. Auch dies ist nicht möglich, da alle Register 32 Bit breit sind und somit mit diesen Algorithmen keine 32-Bit Division implementiert werden kann. Da der Rest der Division automatisch mit geführt wird, wird derselbe Algorithmus für die Berechnung des Quotienten und des Restes zweier ganzer Zahlen verwendet. Es wird jeweils nur ein anderer temporärer Wert (Quotient oder Rest) zurückgegeben. Insgesamt benötigt die Implementierung der vorzeichenlosen Ganzzahldivision für 32-Bit Zahlen 226 Taktzyklen und ist somit

sehr ineffizient (siehe auch Abschnitt 5.3.3).

Die Berechnung der vorzeichenbehafteten Ganzahldivision basiert ebenfalls auf demselben Algorithmus. Genauer gesagt benutzt die vorzeichenbehaftete Division und Berechnung des Rests den gleichen Code wie die vorzeichenlose Variante. Vorher werden die Operanden, Nenner und Zähler, in positive Zahlen umgewandelt. Nach der Berechnung wird das Vorzeichen des Ergebnisses wieder umgedreht, falls genau einer der Operanden negativ ist. Dies ist möglich, da $x/-y = -x/y = -(x/y)$ und $-x/-y = x/y$, wodurch einerseits eine Division zweier negativer Zahlen der Division der absoluten Werte dieser Zahlen entspricht und andererseits bei der Division mit einer negativen Zahl das Vorzeichen weggenommen und wieder an das Ergebnis angefügt werden kann, wodurch die eigentliche Division immer vorzeichenlos durchgeführt werden kann.

Ein weiterer nicht in Hardwareinstruktionen unterstützter Operator ist die Fließkommadivision. Diese wird mit dem Newton-Raphson Verfahren implementiert. Wie auch für die Ganzzahldivision gibt es weitere Algorithmen, um die Fließkommadivision zu implementierten. Jedoch ist das Newton-Raphson-Verfahren genügend effizient, sodass vorerst kein Bedarf für ein komplexeres Verfahren besteht. Im Newton-Raphson-Verfahren (siehe Pseudocode in Listing 2.9) wird der Divisor $P = 1/D$ angenähert, sodass $Q = N/D = P * N$ berechnet werden kann. Die initiale Schätzung für $P_0 \approx 1/D$ wird über die SFU berechnet. Die Berechnung des multiplikativen Inverses über die SFU ist zwar zu ungenau zur direkten Bestimmung von $1/D$, jedoch genau genug für die initiale Schätzung (siehe Abschnitt 1.3.1). Anschließend wird die Schätzung von P iterativ verbessert (siehe Listing 2.9, Zeile 3). Aufgrund der quadratischen Konvergenz des Newton-Raphson-Verfahrens werden für die Berechnung mit Fließkommazahlen mit „normaler" Genauigkeit (`float`) fünf Iterationen benötigt ($2^5 = 32 > 24$ Binärstellen der Mantisse des 32-Bit Fließkommatypen). Nach den fünf Iterationen wird das Ergebnis $Q = P_5 * N$ berechnet. Die Implementierung des Newton-Raphson Verfahrens benötigt (mit Kontrolle auf Division durch null und `NaNs`) insgesamt 24 Taktzyklen und ist damit viel effizienter als die Ganzzahldivision. [Zha99, Abschnitt 5.2]

```
1 // Handle D, N being zero/NaN
2 P_0 ≈ 1/D
3 P_{i+1} = P_i(2 − D * P_i)   // Repeat n times
4 Q = P_∞ * N
```

Listing 2.9: Pseudocode der Newton-Raphson Fließkommadivision [Zha99]

2.2.5 Speicherzugriff optimieren

Die VideoCore IV GPU kann über DMA auf den Hauptspeicher der Host-CPU zugreifen. Dies geschieht jedoch nicht direkt von der QPU aus, sondern über eine weitere Komponente, die VPM (Vertex Pipe Memory), die als Zwischenspeicher oder Cache für alle Speicherzugriffe dient (siehe Abschnitt 1.3.1). Somit müssen für jeden Speicherzugriff (lesend oder schreibend) die DMA für die Kommunikation zwischen VPM und Hauptspeicher und der Zugriff der QPUs auf die VPM konfiguriert werden. Im Allgemeinen sieht der Speicherzugriff aus der VideoCore IV GPU heraus folgendermaßen aus: Auf der linken Seite sind die Schritte zum Lesen aus dem Arbeitsspeicher und auf der rechten Seite die zum Schreiben aufgelistet.

1. Konfiguration DMA
2. Setzen der Speicheradresse
3. Warten auf Abschluss
4. Konfiguration VPM
5. Lesen aus VPM

1. Konfiguration VPM
2. Schreiben in VPM
3. Konfiguration DMA
4. Setzen der Speicheradresse
5. Warten auf Abschluss

Bei den Konfigurationen der VPM und DMA wird jeweils der Datentyp (8-Bit, 16-Bit oder 32-Bit), die Größe des SIMD-Vektors (1, 2, 3, 4, 8 oder 16) sowie die Anzahl der zu lesenden oder schreibenden Vektoren angegeben. Durch das Setzen der Speicheradresse wird die jeweilige DMA-Operation angestoßen. Da diese länger als ein Taktzyklus benötigt, wird ein spezielles Register ausgelesen, das blockiert, bis die Operation abgeschlossen ist. Der Zugriff auf die VPM hingegen führt keine zusätzliche Verzögerung ein. Da OpenCL C-Kernel in beliebiger Reihenfolge auf die dem Kernel als Parameter übergebenen Speicherbereiche zugreifen können, wird standardmäßig für jeden geschriebenen (oder gelesenen) SIMD-Vektor ein extra Speicherzugriff erzeugt. D. h. jeder Wert, der aus dem Arbeitsspeicher gelesen (oder in den RAM geschrieben) wird, benötigt (mit Absicherung durch das Hardware-Mutex) acht Instruktionen, wovon eine mindestens fünf Taktzyklen blockiert [Sug16, Eintrag QV56]. Da jedoch einerseits in OpenCL-Kernel oft zusammenhängende Speicherbereiche hintereinander gelesen (oder geschrieben) werden (z. B. in Schleifen, die über ein Array iterieren) und andererseits die VPM bis zu 64 16-fache SIMD-Vektoren gleichzeitig in den Hauptspeicher schreiben und bis zu 16 16-fachen SIMD-Vektoren gleichzeitig aus dem Hauptspeicher lesen kann [Bro13, Tabellen 32 und 33], bietet es sich an, gleichartige Speicherzugriffe auf zusammenhängende Speicherbereiche in eine DMA-Operation zu vereinen. Gleichartig bedeutet hier, dass die gleiche Operation (Lesen oder Schreiben) auf den gleichen Datentypen (Typgröße in Bytes und Anzahl der SIMD-Vektorelemente) ausgeführt wird.

```
1 //addr is the address of some memory location of type int8*
2 int8 a = addr[0];
3 int8 b = addr[1];
4 int8 c = a + b;
5 int8 d = addr[2];
6 int8 e = addr[4];
7 int8 f = addr[5];
8 int8 g = addr[6];
```

Listing 2.10: Beispielcode für die Optimierung von Speicherzugriffen

Hierfür werden als Erstes Blöcke gleichartiger Speicherzugriffe auf zusammenhängende Speicherbereiche gesucht. D. h. es werden innerhalb eines Basisblocks aufeinanderfolgende Lese- oder Schreibinstruktionen gesucht, bei denen jede Lese- oder Schreibinstruktion auf den Speicherbereich direkt nach dem der vorherigen Instruktion zugreift. Dabei ist egal, auf welche Adresse die erste Lese- oder Schreibinstruktion eines solchen Blocks zugreift. Als „direkt danach" gilt die Speicheradresse, die genau um die Größe des gelesenen oder geschriebenen Datentypen größer ist als die vorherige Adresse. Für das Beispiel in Listing 2.10 bedeutet dies, dass zwei Speicheradressen aufeinander folgen, wenn sie einen Abstand von 32 Bytes (4-Byte Integer und 8-facher SIMD-Vektor) besitzen. Dabei wird bei jeder anderen Art von Speicherzugriff (z. B. eine Schreibinstruktion in einer Serie von Leseinstruktionen oder eine Leseinstruktion von einer anderen Speicheradresse oder der Speicherzugriff mit einem anderen Datentypen) der gefundene Block abgeschlossen und ein neuer begonnen. Für das Beispiel aus Listing 2.10 gibt es somit zwei Blöcke von Leseinstruktionen, die zusammengefasst werden können, das Lesen von `addr[0]` bis einschließlich `addr[2]` und das Lesen der Positionen vier bis sechs.

Wenn ein Block gleichartiger Speicherzugriff auf zusammenhängende Speicherbereiche beendet ist, z. B. durch andere Speicherzugriffe, durch das Ende des Basisblocks oder durch erreichen der maximalen Größe (16 für Lesezugriffe, 64 für Schreibzugriffe), werden die einzelnen Lese- oder Speicherinstruktionen darin kombiniert. Dafür werden alle bis auf eine Instruktion zum Setzen der Speicheradresse sowie zum Warten auf die Fertigstellung des DMA-Zugriffs entfernt. Bei Leseoperationen werden alle außer der jeweils ersten Instruktion entfernt, bei Schreibzugriffen alle außer der jeweils letzten, da der Schreibvorgang von der VPM in den Hauptspeicher erst erfolgen kann, wenn alle Daten in die VPM geschrieben sind. Ebenso werden die Konfigurationen für die jeweilige DMA-Operation sowie für den Zugriff auf die VPM angepasst, sodass die richtige Anzahl an Werten gelesen oder geschrieben werden. Im Beispiel aus dem Listing 2.11 werden z. B. die Konfigurationen für das Lesen aus der VPM als auch für den DMA-Zugriff so angepasst, dass drei Vektoren mit jeweils acht 32-Bit Elementen auf einmal gelesen werden, anstatt dem vorher für jeden Lesevorgang konfigurierten einzelnen Element.

```
1  //Block 1
2  mutex_aqcuire();
3  dma_setup(int, 8, 3);
4  dma_addr = addr;
5  dma_wait();
6  vpm_setup(int, 8, 3);
7  int8 a = vpm_read();
8  int8 b = vpm_read();
9  int8 c = a + b;
10 int8 d = vpm_read();
11 mutex_release();
12 //Block 2
13 mutex_aqcuire();
14 dma_setup(int, 8, 3);
15 dma_addr = addr + 4 * sizeof(int8);
16 dma_wait();
17 vpm_setup(int, 8, 3);
18 int8 e = vpm_read();
19 int8 f = vpm_read();
20 int8 g = vpm_read();
21 mutex_release();
```

Listing 2.11: Ergebnis der Optimierung von Speicherzugriffen des Codes aus 2.10

Das Ergebnis dieser Optimierung für das Codebeispiel aus Listing 2.10 kann in Listing 2.11 als Pseudocode gesehen werden. Insgesamt werden somit nur noch 21 Maschinenbefehle benötigt (die `dma_setup()` Pseudoinstruktion benötigt zwei Maschinenbefehle) anstatt der 49 Instruktionen des ursprünglichen Codes. Da das Schreiben in oder Lesen aus der VPM die Adresse automatisch inkrementiert, muss zwischen den `vpm_read()` Instruktionen die VPM-Adresse nicht explizit auf den nächsten Wert gesetzt werden. Ebenso muss nur noch zweimal auf die Fertigstellung der DMA-Operation gewartet werden. Um die optimierte Version jedoch weiterhin gegen gleichzeitige Speicherzugriffe durch verschiedene QPUs abzusichern, muss der gesamte Block durch das Hardware-Mutex abgesichert werden. Ein Nachteil dieser Optimierung ist es, dass das Zusammenfügen von Speicherzugriffen die Anzahl der Instruktionen zwischen Reservieren und Freigeben des Hardware-Mutex zum Teil sehr stark erhöht und somit den kritischen Abschnitt stark vergrößern kann. Dies führt dazu, dass die anderen QPUs in der Zwischenzeit länger untätig auf die Freigabe des Mutex warten müssen und somit zumindest ein Teil der eingesparten Instruktionen wieder verloren geht.

Eine weitere, bisher noch nicht implementierte, Optimierung kann die Anzahl der Zugriffe auf den Hauptspeicher über die DMA-Schnittstelle noch weiter reduzieren. Hierfür wird die VPM-Komponente als Cache für Speicherzugriffe verwendet.

Der Grundgedanke der Optimierung ist ähnlich der bereits beschriebenen Optimierung, jedoch auf globaler Ebenen (über den gesamten Programmverlauf). Ziel ist es, Speicherbereiche, die aus dem Kernelcode heraus gelesen werden einmalig in die VPM zu laden und bei allen folgenden Lesezugriffe aus der Kopie in der VPM heraus zu lesen. Für Schreibvorgänge in den Hauptspeicher gilt ein ähnliches Prinzip: Alle Werte werden in die VPM geschrieben und am Ende der gesamte Block veränderter Daten einmalig in den Arbeitsspeicher kopiert. Somit könnte die Anzahl der DMA-Operationen auf bis zu einer Lese- oder Schreiboperation pro verwendetem Speicherbereich reduziert werden. Damit diese Optimierung funktioniert, muss der verfügbare VPM-Speicher auf die jeweils gecachten Speicherbereiche im Hauptspeicher aufgeteilt werden. Eine Möglichkeit diese Optimierung umzusetzen wäre es, bei der Kompilierung die VPM in Blöcke aufzuteilen, die jeweils einem Speicherbereich auf dem RAM entsprechen. Hierfür muss jedoch die Größe der verwendeten Speicherbereiche zur Kompilierzeit bekannt sein, was sich in den meisten Fällen nicht herausfinden lässt. Ebenso teilen sich alle QPUs eine VPM und somit eine gemeinsame Cache, wodurch auf verschiedenen Regionen geschrieben werden muss und der Zugriff auf diese durch das verfügbare Hardware-Mutex abgesichert werden muss.

2.2.6 Reordering

Ein weiterer Optimierungsschritt, der auch in den meisten gängigen Compilern angewendet wird, ist das Umsortieren von Instruktionen innerhalb eines Basisblocks. Der Code dieser Optimierung steht im VC4CC-Compiler in der Quelldatei `optimization/Reordering.cpp` und dient dem Zweck, `nop`-Instruktionen zu eliminieren, indem diese durch „produktive" Instruktionen ersetzt werden. `nop`s können unter anderem durch einen SFU-Aufruf eingefügt werden, um auf das Ergebnis der SFU zu warten. Ebenso fügt ein vorheriger Optimierungsschritt künstlich `nop`-Instruktionen ein, um Read-after-Writes zu trennen. Ein Read-after-Write tritt auf, wenn eine Instruktion eine Variable schreibt und die nächste Instruktion diese Variable wieder liest. Diese Fälle werden durch ein `nop` getrennt, um die Variable auch auf physikalische Register abbilden zu können (da diese ja ein Lesen in der nächsten Instruktion, nachdem sie beschrieben werden nicht erlauben, siehe Abschnitt 1.3.2) und somit die Registerzuordnung (Abschnitt 2.3.2) mehr Optionen hat. Durch das Reordering werden die meisten der so eingefügten zusätzlichen Verzögerungsinstruktionen wieder ersetzt, wodurch der erzeugte Code nur begrenzt größer wird. Weiterhin kann das Reordering dazu verwendet werden, um das Kombinieren von Instruktionen (siehe Abschnitt 2.2.7) zu optimieren, indem versucht wird, Instruktionen soweit umzusortieren, dass möglichst viele der Instruktionen vor oder nach einer anderen Instruktion stehen, mit dieser sie kombiniert werden können. Diese Anwendung ist aufgrund der zusätzlichen Komplexität (und somit

zusätzlichen Kompilierzeit) derzeit noch nicht implementiert.

Das Vorgehen lässt sich grundlegend folgendermaßen formulieren: Für alle Basisblöcke innerhalb eines Kernels werden alle `nop`-Instruktionen gesucht. Daraufhin wird versucht, eine Instruktion weiter hinten im gleichen Basisblock zu finden, die die behandelte `nop`-Instruktion ersetzen kann, ohne die Semantik des Basisblocks zu verändern:

1. Initialisieren der gesperrten Werte
2. Für jede folgende Instruktion:
    a. Wird ein gesperrter Wert verwendet?
    b. Wird ein Signal ausgelöst oder die Status-Flags gesetzt oder abgefragt?
    c. Ist die Instruktion eine Delay-Instruktion?
    d. Wird das Hardware-Mutex gesetzt oder freigegeben?
    e. Wenn keins der oberen Kriterien zutrifft, breche ab
    f. Füge Ergebnis der Instruktion an gesperrte Werte hinzu
3. Verschiebe Kandidat an die behandelte Stelle

(1) Als Erstes werden die Werte (lokale Variablen oder Register) ermittelt, auf die ein möglicher Ersetzungskandidat nicht zugreifen darf. Dabei muss der Grund für das Einfügen der Delay-Instruktion (`nop`) beachtet werden: Wenn z. B. die Delay-Instruktion zwischen dem Setzen des SFU-Registers (siehe Abschnitt 1.3.1) und dem Auslesen des Ergebnisses steht, dann darf die ersetzende Instruktion weder auf SFU-Register noch auf die Ergebnisvariable des SFU-Aufrufs zugreifen. Wurde die Delay-Instruktion jedoch zum Trennen von Read-after-Write eingefügt, darf die ersetzende Instruktion das Ergebnis der vorherigen Instruktion (dessen Lesen vom Schreiben getrennt wurde) nicht lesen oder beschreiben. (2) So wird, beginnend mit der direkt folgenden Instruktion, bis zum Ende des Basisblocks, jede Instruktion analysiert, ob sie umsortiert werden kann. (2.a) Falls die aktuell analysierte Instruktion einen der gesperrten Werte als Eingabe verwendet, kann sie nicht nach vorne gezogen werden, da sonst die Operanden ungültig sind. (2.b) Ebenso werden, um den Optimierungsschritt einfach zu halten, alle Instruktionen, die Signale oder Status-Flags setzen oder verwenden, als Kandidaten abgelehnt, da ansonsten die Art des Signals sowie alle Instruktionen, die Status-Flags setzen analysiert werden müssen, ob diese die vom aktuellen Kandidaten gesetzten Flags überschreiben. (2.c) `nop` Delay-Instruktionen können auch nicht verwendet werden, da diese sonst den Grund für die Verzögerung an ihrer ursprünglichen Position verletzen würden. (2.d) Ebenso werden Zugriffe auf das Hardware-Mutex übersprungen, da diese sonst den kritischen Abschnitt verlängern oder verschieben würden. (2.e) Wurde eine

Instruktion gefunden, die keine der oberen Kriterien verletzt, kann diese für die Ersetzung verwendet werden und die Suche wird abgebrochen. (2.f) Verletzt die aktuelle Instruktion eine der Kriterien, kann sie nicht umsortiert werden, jedoch muss ihr Ergebnis mit in die Liste der gesperrten Werte angefügt werden. (3) Ist eine passende Instruktion gefunden, die keine der Kriterien verletzt, wird diese anstelle der `nop` Instruktion eingefügt. Durch diese Optimierung werden die meisten Delay-Instruktionen durch „produktive" Instruktionen ersetzt und somit die Ausführungszeit der kompilierten Programme verbessert.

2.2.7 Instructions Combiner

Die Architektur der VideoCore IV GPU hat die Besonderheit, dass jede QPU zwei asymmetrische ALUs besitzt, die gleichzeitig (innerhalb eines Maschinenbefehls) angesprochen werden können um zwei verschiedenen Operationen auf verschiedenen Daten ausführen zu können (siehe Abschnitt 1.3.2). Dies nennt sich auch **MIMD** (Multiple Instructions, Multiple Data). Jedoch bieten die vom Compiler als Input akzeptierten Sprachen (LLVM IR und SPIR-V) keine MIMD-Instruktionen, die auch von den meisten Computerarchitekturen nicht unterstützt werden. Ein Optimierungsschritt, der somit mitunter die Ausführungszeit eines kompilierten OpenCL-Kernels sehr stark verkleinern kann, ist das Kombinieren von Instruktionen. Den theoretisch maximalen Ertrag würde das Kombinieren von Instruktionen erreichen, wenn bei jedem Maschinenbefehl beide ALUs produktive Befehle ausführen. Dies ist jedoch in der Praxis aus verschiedenen Gründen kaum erreichbar, da eine Vielzahl an Kriterien beachtet werden muss, um in einem Maschinenbefehl beide ALUs verwenden zu können. Ebenso ist der Befehlssatz der Multiplikations-ALU mit acht Instruktionen [Bro13, Tabelle 13] sehr begrenzt und kann somit nur für eine begrenzte Anzahl an Operationen verwendet werden.

Der Instruction Combiner, der seinen Quellcode in der Datei `optimization/Combiner.cpp` hat, iteriert über alle Instruktionen, die eine der beiden ALUs verwenden (also alle arithmetischen und logischen Berechnungen sowie einfach `mov` zum Kopieren von Werten) und kontrolliert, ob die nächste Instruktion auch eine der ALUs verwendet. Ist dies der Fall, werden die nötigen Kriterien für die beiden gefundenen Instruktionen untersucht:

1. Werden verschiedene ALUs verwendet?
2. Werden keine Peripherieregister gelesen oder geschrieben?
3. Sind die beiden Instruktionen unabhängig voneinander?
4. Besteht kein Konflikt zwischen den Ausgaben der beiden Instruktionen?

5. Ist die zweite Instruktion unabhängig von Status-Flags, die eventuell von der ersten gesetzt werden?
6. Setzt maximal eine der Instruktionen die Status-Flags?
7. Wird maximal ein konstanter Wert als Eingabe verwendet?
8. Werden maximal zwei verschiedene Eingabewerte verwendet?

(1) Als Erstes muss überprüft werden, ob die Operationen der beiden Instruktionen auf unterschiedlichen ALUs ausgeführt werden können. Da die Additions-ALU mit ihren 24 Befehlen (im Gegensatz zu den acht Befehlen der Multiplikations-ALU) die meisten Operationen ausführt, scheitert das Kombinieren oft bereits an diesem Kriterium [Bro13, Tabellen 12 und 13]. Die einzige Operation, die wahlweise auf beiden ALUs ausgeführt werden kann ist `mov`, das Kopieren eines Wertes in ein anderes Register. Hierfür wird auf der Additions-ALU der Befehl `or` verwendet (z. B. `out = or in, in`), da gilt: $A \vee A = A$. Auf der Multiplikations-ALU wird der Befehl `v8min` verwendet, der byteweise das Minimum der beiden Operanden berechnet. Auch hier wird für beide Operanden derselbe Wert verwendet (z. B. `out = v8min in, in`), was zu einem einfachen Kopieren führt. Das zweite Kriterium (2) ist nicht zwingend erforderlich, erleichtert jedoch die Registerzuweisung in Abschnitt 2.3.2 und erspart das Überprüfen auf spezielle Kombinationen (z. B. das gleichzeitige Beschreiben von VPM Konfigurationsregistern). Da die Operationen auf den beiden ALUs zeitgleich ausgeführt werden, darf die zweite Instruktion nicht auf das Ergebnis (3) oder die möglicherweise von der ersten Instruktion gesetzten Status-Flags (5) zugreifen. Ebenso dürfen die Instruktionen nicht in die gleiche Variable schreiben (4). Hier gibt es jedoch die Ausnahme, dass wenn beide Instruktionen eine bedingte Ausführung haben und sich Bedingungen der Ausführung der Instruktionen gegenteilig sind, dann dürfen die Instruktionen kombiniert werden, auch wenn sie in die gleiche Variable schreiben, da so immer genau eine der Instruktionen ausgeführt wird. Ein Beispiel hierfür, das bei jedem relationalen Operator eingefügt wird, um das Ergebnis korrekt zu setzen, sind die Instruktionen `out = bool true (if some flag is set)` und `out = bool false (if same flag is not set)`. Der Maschinencode der VideoCore IV GPU ist so spezifiziert, dass eine Instruktion nicht festlegen kann, welche der beiden ALUs die Status-Flags aktualisiert. Genauer besagt die offizielle Dokumentation, dass „die [Status-]Flags werden von der Additions-ALU aktualisiert, außer die Additions-ALU führt ein `nop` aus [...] in welchem Falle die Flags von der Multiplikations-ALU aktualisiert werden" [Bro13, Tabelle 1]. Das bedeutet, dass kontrolliert werden muss (6), ob einerseits nur eine der Instruktionen die Status-Flags setzt und andererseits diese Instruktion auch auf der Additions-ALU ausgeführt wird. Wie bereits in Abschnitt 1.3.2 beschrieben, wird bei Berechnungen mit direkt verwendeten Werten (Konstanten) dieser anstelle des Operanden aus der Registerdatei B geladen. Somit

kann nur ein konstanter Wert pro Instruktion verwendet werden, weshalb auch bei kombinierten Instruktionen dafür gesorgt werden muss, dass falls beide Instruktionen einen konstanten Wert verwendet, dieser den gleichen Wert hat (7). Da zwei binäre Operationen kombiniert werden, kann das Ergebnis bis zu vier Operanden haben. Jedoch kann ein Maschinenbefehl nur von maximal zwei physikalischen Registern (plus vier Akkumulatoren) lesen (siehe [Bro13, QPU Instruction Encoding]). Da jedoch bei diesem Optimierungsschritt noch nicht bekannt ist, ob die verwendeten lokalen Variablen in der Registerzuweisung (siehe Abschnitt 2.3.2) auf Akkumulatoren oder physikalische Register abgebildet werden, werden zur Sicherheit nur Instruktionen kombiniert, bei denen das Ergebnis maximal zwei verschiedene Eingabewerte (feste Register, konstante Werte oder lokale Variablen) benötigt (8). Bei der Analyse dieser Optimierung, angewendet auf die zur Entwicklung und zum Testen des VC4CC-Compilers verwendeten OpenCL-Kernel, hat sich eine durchschnittliche Verkleinerung der Anzahl der Instruktionen um 6% ergeben. Im Vergleich zu den durchschnittlich weniger als 1% der Kompilierzeit, die in diesem Optimierungsschritt verbracht wird, ist diese Optimierung rentabel.

2.3 Backend

Die Aufgabe des Backends des Compilers, das sich in der Quelldatei `asm/CodeGenerator.cpp` befindet, ist es, die Instruktionen aus der internen Repräsentation in Maschinencode der VideoCore IV-Architektur umzuwandeln. Bevor der Maschinencode erzeugt werden kann, muss dafür gesorgt werden, dass der Kernel auf die Parameter zugreifen kann (Abschnitt 2.3.1). An dieser Stelle werden auch die drei benötigten Delay-Instruktionen nach den Sprung-Instruktionen eingefügt (siehe Abschnitt 1.3.2), da sie für die genaue Berechnung der Codeposition zum Auflösen der Labels benötigt werden. Ebenso werden vorher noch die lokalen Variablen auf Register abgebildet (Abschnitt 2.3.2). Die eigentliche Codegenerierung ist eher einfach gehalten: Unter Hinzunahme der vorher auf Register und Codepositionen abgebildeten Variablen (inklusive der Parameter) und Labels wird jede Instruktion der internen Repräsentation auf genau ein Objekt eines weiteren Instruktionstypen abgebildet, der jedoch nur eine komfortablere Repräsentation der 64-Bit Maschinenbefehle darstellt. Die Liste dieser Objekte wird dann, wenn alle im Quellcode vorhandenen Kernel fertig kompiliert sind, auf dem Ausgabestrom ausgegeben (Abschnitt 2.3.4).

2.3.1 Start- und Stopsegmente

Im Compiler-Frontend werden die Signaturen sowie die Inhalte der OpenCL-Kernel eingelesen und im Middleend werden die Instruktionen innerhalb der Kernel (oder

in von den Kernel aufgerufenen Funktionen) in eine auf den Befehlssatz der VideoCore IV GPU (siehe Abschnitt 1.3.2) abbildbare Form gebracht. Die Signaturen (Parameter und Rückgabewerte) der Kernelfunktionen werden bis zu diesem Punkt nicht weiter betrachtet. Der Zugriff auf die Parameter wird einfach als Zugriff auf lokale Variablen des Parameter-Typen und mit dem Namen des jeweiligen Parameters behandelt. Der Rückgabewert des Kernels ist immer `void` (siehe [Gro12, Abschnitt 6.9]), weswegen keine weitere Behandlung für die Rückgabe von Werten vom Kernel in den Host-Code benötigt wird. Ein Kernel gibt hingegen Werte an die aufrufende Host-Anwendung zurück, indem er diese direkt in die durch die Parameter bestimmten Speicherbereiche schreibt. Um jedoch im Kernel auf die Parameterwerte zugreifen zu können, müssen diese zu Beginn der Ausführung des Kernels in die Register der QPU eingelesen werden. Dies geschieht im **Startsegment**, das vor die Instruktionen des Kernels eingefügt wird. Dafür wird pro Kernel-Parameter eine Instruktion erstellt, die die Parameter in der richtigen Reihenfolge aus den Uniforms lesen und in die jeweiligen lokalen Variablen schreiben. Uniform Values (oder kurz: Uniforms) sind ein Konzept aus OpenGL und stellen Werte dar, die für alle Instanzen eines Shaders gleich sind. In der VideoCore IV-Architektur sind Uniforms „32-Bit Werte, die in Listen im Arbeitsspeicher gespeichert sind". Bei der Ausführung von Code auf den QPUs wird ein Zeiger zum Beginn der Uniform-Listen an den Treiber übergeben. Durch das Lesen eines bestimmten Peripherie-Registers kann ein Uniform-Wert GPU-seitig gelesen werden und der Uniform-Zeiger wird automatisch inkrementiert (siehe [Bro13, Kapitel 3]). Somit eignen sich Uniforms sehr gut, um Parameter an die QPUs zu übergeben, da GPU-seitig nur das Uniform-Register ausgelesen und in entsprechende lokale Variable kopiert werden muss. Neben den explizit in der Signatur der Kernel-Funktion definierten Parametern werden noch zusätzlich weitere Informationen über das ausgeführte Work-Item sowie dessen Work-Group (siehe Abschnitte 2.2.3 und 3.1.1) aus den Uniforms ausgelesen und in die jeweilige lokalen Variable kopiert. Das Startsegment, das an den Maschinencode für den Beispielkernel zur Berechnung der Fibonaccizahlen aus Anhang A.1 angehängt wird, kann in den ersten 16 Zeilen des Kernelcodes in Anhang A.6 gesehen werden (Zeilennummern 2 bis 17).

Das **Stopsegment** hingegen wird an das Ende des Codeblocks eines Kernels angehängt. Hier wird dem Scheduler der VideoCore IV GPU mitgeteilt, dass dieser Hardwarethread jetzt endet. Dies geschieht über ein Signal, ein besonderes Feld innerhalb eines ALU-Befehls, das bestimmte Aktionen in der Grafikhardware auslösen kann (siehe [Bro13, Kapitel 3]). Wie auch bei Sprüngen (siehe Abschnitt 1.3.2) müssen nach dem Senden des Signals drei `nop`-Instruktionen eingefügt werden, um die Pipeline zu leeren. Innerhalb dieser Instruktionen fängt der Scheduler bereits an, die Vorbereitungen für die Ausführung des nächsten Programms (OpenCL-Kernel, OpenGL-Shader oder anderer benutzerdefinierter Code) zu starten, falls weitere Programme eingereiht sind. Bevor der Thread beendet wird, wird noch ein spezi-

elles Peripherieregister geschrieben, das einen Interrupt auslöst. Dieser teilt dem Treiber mit, dass ein Programm die Ausführung auf einer QPU beendet hat. Die hostseitige Handhabung wird in Abschnitt 4.2.4 beschrieben. Ähnlich wie auch das Startsegment kann auch das Stopsegment in Anhang A.7 in den Zeilen 8 bis 11 gut erkannt werden.

2.3.2 Registerallokation

Die meisten Compiler (wie auch der VC4CC-Compiler und LLVM) arbeiten intern mit einer Zwischensprache (z. B. LLVM IR), die neben Funktionen, die die eigentliche Zielarchitektur nicht unterstützt meist auch eine „unbegrenzte" Anzahl an lokalen Variablen unterstützt. Die von der Zielplattform nicht unterstützten Funktionen und Operationen müssen im Laufe des Compilers auf unterstützte Operationen umgewandelt werden (siehe Abschnitt 2.2) und die lokalen Variablen müssen auf Speicherbereiche der Zielhardware (z. B. Register oder Hauptspeicher) abgebildet werden. Dies geschieht in der Registerallokation (oder auch Registerzuordnung). Der Grundsatz der Registerzuordnung ist es, alle benötigten Variablen auf die durch die Hardware begrenzte Anzahl an Registern abzubilden, sodass es keine Konflikte gibt. D. h. dass zu keinem Zeitpunkt mehrere Variablen auf das gleiche Register abgebildet werden dürfen, da diese sich sonst gegenseitig überschreiben. Gelingt es einer Registerzuordnung nicht, alle Variablen auf die Register des Zielprozessors abzubilden, ohne dieses Kriterium zu verletzen, kann auf Register Spilling zurückgegriffen werden. Register-Spilling („Register ausschütten") bedeutet, dass Variablen (zumindest für einen Teil ihres Gültigkeitsbereiches) auf den Hauptspeicher ausgelagert werden, wodurch sie keine Register blockieren. Jedoch müssen dafür Instruktionen für das verhältnismäßig langsame Schreiben und Lesen in den Hauptspeicher eingefügt werden, was die Gesamtperformance des Programms verringert. Daher versuchen die meisten Algorithmen für die Registerallokation, das Spilling weitestgehend zu vermeiden [Per08, Abschnitt 2.1.4].

Für die Registerzuordnung gibt es verschiedene Standardverfahren, wovon zwei hier kurz vorgestellt werden: Die Registerzuordnung mithilfe von Graph Coloring („Einfärben von Graphen") und Linear Scan („Lineare Suche"). Bei der Verwendung der linearen Suche werden die Basisblöcke eines Programms (siehe Abschnitt 2.2.6) in eine lineare Abfolge gebracht, sodass danach die Gültigkeitsbereiche der Variablen (vom ersten Schreiben bis zum letzten Lesen) auf zusammenhängende Intervalle abgebildet werden können. Diese Intervalle werden „eingefärbt", d. h. sie werden so den verfügbaren Registern zugeordnet, dass keine sich überlappende Intervalle die gleiche Farbe haben (dem gleichen Register zugeordnet sind). Die lineare Suche ist meist schneller als die Verwendung von Graphen, erzeugt jedoch nicht optimalen Code, da Register zum Teil durch ein Gültigkeitsintervall einer Variable blockiert werden, obwohl diese Variable in einem Bereich des Intervalls

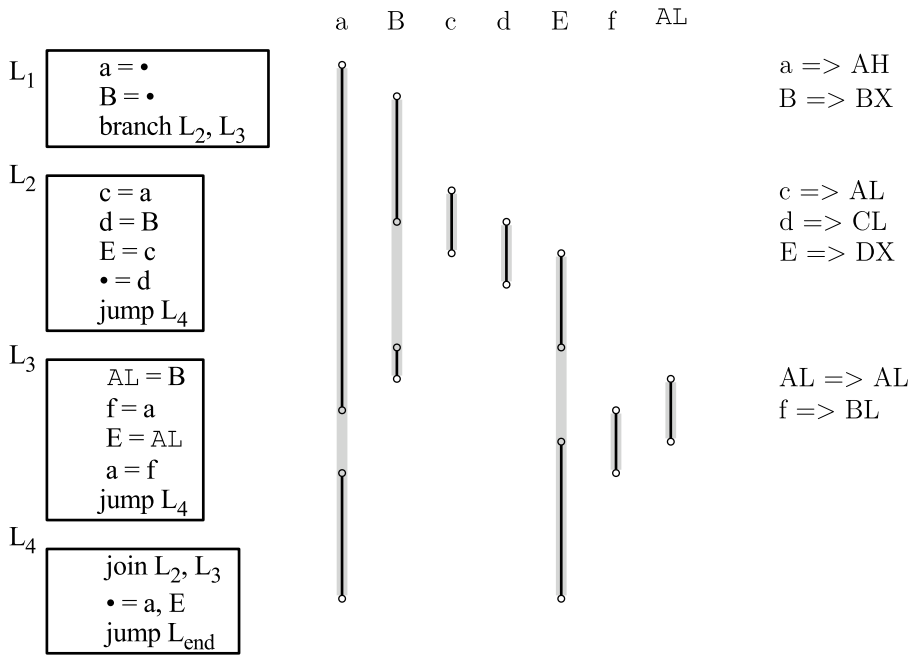

Abbildung 2.2: Lineare Suche für die Registerzuweisung [Per08, Abbildung 2.10]

nicht verwendet wird. Abbildung 2.2 aus [Per08] zeigt ein solches Beispiel: Da die Gültigkeitsintervalle der Variablen `B` und `E` Lücken aufweisen, könnten diese auf das gleiche Register abgebildet werden, was jedoch der Algorithmus nicht unterstützt. Das gleiche gilt für die Variablen `a` und `f`. Graph Coloring erzeugt in dieser Hinsicht besseren Code, da Variablen sich bei der Registerzuweisung nur blockieren, wenn sie gleichzeitig gültig sind, wodurch die Register der Zielarchitektur besser ausgenutzt werden können und seltener Variablen auf den Hauptspeicher ausgelagert werden müssen. Hierfür werden alle Variablen als Knoten eines ungerichteten Graphen verwendet, dessen Kanten die gegenseitige Blockierung von möglichen Registern darstellen, d. h. zwei Knoten, zwischen denen eine Kante existiert, dürfen nicht auf das gleiche Register abgebildet werden. Ist der Graph erstellt, wird jedem Knoten eine Farbe (jeder Variable ein Register) zugeordnet, unter der Berücksichtigung, dass keine benachbarten Knoten die gleiche Farbe haben dürfen (dem gleichen Register zugeordnet sein dürfen). Ist dies nicht ohne Konflikte möglich, müssen auch hier Variablen auf den Arbeitsspeicher ausgelagert werden. [Per08, Abschnitte 2.2.1 und 2.2.2]

Aufgrund der „irregulären Architektur" der VideoCore IV GPU können die bereits genannten geläufigen Verfahren für die Registerzuordnung nicht ohne Anpassung verwendet werden. Die Bezeichnung „Irreguläre Architektur" bezieht sich hier auf die Eigenschaft, dass es zwei physikalische Registerbänke gibt und Instruktionen nicht auf zwei Operanden der gleichen Registerbank zugreifen können [Bro13, QPU

Instruction Encoding]. Weiterhin kommt der Sonderfall hinzu, dass nur die zusätzlichen Akkumulatoren in zwei aufeinander folgenden Instruktionen geschrieben und wieder gelesen werden können (siehe Abschnitt 1.3.2). Dieses Problem wird auch als Precoloring bezeichnet, da in bestimmten Situationen Variablen bereits im Voraus bestimmten Registern (oder hier Registerbänken) zugeordnet werden müssen (siehe auch [Per08, Abschnitt 2.1.2]). Der in dieser Arbeit verwendete Algorithmus für die Registerzuordnung basiert auf dem Graph Coloring mit Precoloring, hat jedoch die weitere Einschränkung, dass keine Variablen auf den Hauptspeicher ausgelagert werden (sog. „Spill-free" Registerallokation). Diese Einschränkung wurde hinzugefügt, da der Zugriff auf den Hauptspeicher vergleichsweise langsam ist, vor allem, da die mehreren QPUs nicht parallel auf den Arbeitsspeicher zugreifen können (siehe Abschnitt 5.3.3). Ebenso müsste für die Auslagerung von Variablen zur Laufzeit ein bestimmter Arbeitsspeicherbereich verfügbar gemacht werden, was auf von der Laufzeitbibliothek auf der Host-CPU veranlasst werden müsste. Ein der in dieser Arbeit verwendeten Implementierung sehr ähnlicher Algorithmus ist der Runeson/Nyström Graph Coloring Algorithmus für irreguläre Architekturen, der auch in der OpenGL-Implementierung für die VideoCore IV verwendet wird [Anh15].

Im ersten Schritt der Registerallokation, die in der Quelldatei `asm/GraphColoring.cpp` zu finden ist, wird für jede Variable ein Knoten im ungerichteten Graphen erstellt, für die die initial blockierten Registerbänke (A, B oder Akkumulatoren) gesetzt werden. Eine Variable kann z. B. initiale blockierte Registerbänke besitzen, wenn sie als Operand zusammen mit einem konstanten Wert verwendet wird (dann darf sie nicht auf die Registerbank B abgebildet werden) oder in einer Instruktion geschrieben und in der nächsten gelesen wird (Read-after-Write), in welchem Fall sie auf ein Akkumulator abgebildet werden muss. Dies entspricht dem Precoloring. Ebenso werden jedem Knoten die mit der enthaltenen Variablen assoziierten Instruktionen (also Instruktionen, in denen die Variable geschrieben oder gelesen wird) zugeordnet, um eventuelle Konflikte in der Registerzuordnung effizienter beheben zu können. Anschließend werden die Kanten erstellt. Da eine Instruktion nur maximal jeweils ein Register von den Registerbänken A und B lesen kann (siehe auch Abschnitt 2.2.7), müssen zwei verschiedene Arten von Kanten erstellt werden. „Normale" Kanten geben an, dass die Gültigkeitsbereiche zweier Variablen sich überschneiden und somit die beiden Variablen nicht auf das gleiche Register abgebildet werden dürfen. Die zweite Art von Kanten wird verwendet, wenn zwei Variablen zusammen als Operanden einer Instruktion verwendet werden und besagt, dass falls einer der Operanden auf eine physikalische Registerbank abgebildet wird, die gesamte Registerbank für die andere Variable gesperrt ist. Abbildung 2.3 zeigt einen Graphen, der für das Codebeispiel aus Abbildung 2.2 generiert wird. Hierbei sind die Knoten mit dem Namen der jeweiligen lokalen Variable bezeichnet. Alle Knoten für lokale Variablen, deren Gültigkeitsbereiche sich überschneiden, sind

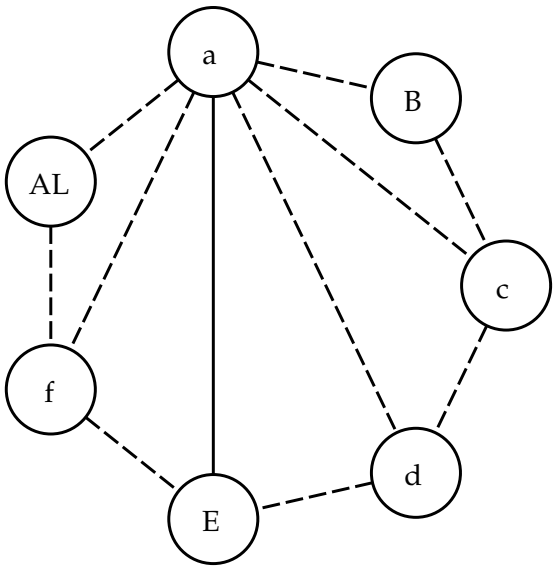

Abbildung 2.3: Graph für die Registerzuodnung des Codebeispiels aus [Per08, Abbildung 2.10]

mit einer gestrichelten Linie verbunden, während die Knoten von Variablen, die gemeinsam in Operationen verwendet werden mit einer durchgestrichenen Linie verbunden sind. Um auch für diese Art von Knoten ein Beispiel zu haben, wird davon ausgegangen, dass die Instruktion • `= a, E` aus Abbildung 2.2 beide lokale Variablen `a` und `E` gleichzeitig verwendet.

Für das Einfärben des Graphen werden die Knoten gleichzeitig mit der Zuordnung der Kanten in zwei Mengen aufgeteilt: Die „geschlossene" Menge beinhaltet alle Knoten, die nur noch eine verfügbare Registerbank besitzen (entweder durch das initiale Precoloring oder durch die Zuordnung der benachbarten Knoten), während alle anderen Knoten (für welche die zugeordnete Registerbank noch nicht fest steht) der „offenen" Menge hinzugefügt werden. Die eigentliche Einfärbung weist jedem Knoten in der geschlossenen Menge ein noch freies Register auf der einzigen noch verfügbaren Registerbank zu. Ebenso wird die zugewiesene Registerbank bei allen Nachbarn blockiert, die zusammen als Operanden verwendet werden, falls es sich um eine physikalische Registerbank handelt und das zugewiesene Register wird für alle anderen Nachbarn blockiert. Hierfür verwaltet jeder Knoten sowohl die noch für die zugeordnete Variable möglichen Registerbänke als auch die noch nicht durch Nachbarknoten blockierten einzelnen Register. Kann, durch die Zuweisung eines Nachbarn, ein Knoten nur noch auf eine einzelne Registerbank zugeordnet werden, wird er zu der geschlossenen Menge hinzugefügt. Anschließend wird der aktuell bearbeitete Knoten aus der geschlossenen Menge entfernt. Ist die geschlossene Menge leer, d. h. alle Knoten sind entweder einem bestimmten Register zugeordnet

oder haben noch mehrere verfügbare Registerbänke (für die es auch noch freie verfügbare Register gibt), wird ein Knoten von der offenen Menge entfernt, einer der noch verfügbaren Registerbänke zugewiesen und in die geschlossene Menge eingefügt. Daraufhin wird wieder die rekursiv die gesamte geschlossene Menge abgearbeitet, bis diese wieder leer ist. Dieser Vorgang wird solange wiederholt, bis auch die offene Menge leer ist.

Bei dem implementierten Algorithmus können verschiedene Konflikte in der Registerzuordnung auftreten, die sich jedoch alle gleich andeuten: Ein Konflikt ist aufgetreten, wenn ein Knoten des Graphen entweder keine verfügbaren Registerbänke oder keine verfügbaren einzelnen Register mehr hat. Diese Kriterien werden in der Abarbeitung der geschlossenen Menge getestet, die fehlerhaften Knoten einer Fehlermenge hinzugefügt und für die weitere Bearbeitung der restlichen Knoten erst einmal ignoriert. Daraus ergibt sich, dass die Registerzuordnung erfolgreich verlaufen ist, wenn die Fehlermenge leer ist. Ist dies nicht der Fall, d. h. am Ende der Registerzuordnung gibt es Elemente in der Menge fehlerhafter Knoten, muss in einem zweiten Schritt eine Fehlerbehebung durchgeführt werden. Hierfür wird für jeden fehlerhaften Knoten der Grund des Fehlers analysiert und dementsprechend behandelt:

Im einfachsten Fall, wenn eine Variable aufgrund eines Read-after-Writes einem Akkumulator zugeordnet werden muss, es jedoch keine freien Akkumulatoren mehr gibt, wird zwischen alle Read-after-Writes dieser Variable eine Delay-Instruktion (`nop`) eingefügt um das Read-After-Write zu unterbrechen, sodass die Variable auch einer der physikalischen Registerbänke zugeordnet werden kann.

Im zweiten Fall kann eine Variable initial einer der beiden physikalischen Registerbänke zugeordnet werden, auf der es auch noch freie Register gibt. Jedoch ist die gesamte Registerbank blockiert, weil die Variable mit anderen Variablen zusammen als Operanden der gleichen Instruktion verwendet wird, die bereits auf die gleiche Registerbank festgesetzt sind (siehe auch Abschnitt 2.2.7). Damit die Variable auf eines der noch freien Register zugeordnet werden kann, muss jede Instruktion, in der diese Variable als Operand verwendet wird, überprüft werden. Nimmt die Instruktion mehrere Operanden, von denen einer die physikalische Registerbank blockiert, wird vor der Instruktion eine Kopie der fehlerhaften Variable in eine neue temporäre Variable eingefügt, die dann als Operand der eigentlichen Instruktion verwendet wird. Dies führt zwar wieder ein Read-after-Write ein, das aber in weiteren Behebungen behandelt wird.

Der dritte Fehlerfall tritt auf, wenn eine Variable für keine der initial verfügbaren Registerbänke mehr freie Register hat, auf die sie abgebildet werden könnte. Kann eine Variable auf gar kein Register mehr zugeordnet werden, d. h. alle Akkumulatoren sowie die jeweils 32 physikalische Register der beiden Registerbänke sind von anderen Variablen belegt, deren Gültigkeitsbereiche sich mit der betroffenen

Variable überschneiden, wird die Fehlerbehandlung abgebrochen. An dieser Stelle müsste die Variable auf den Hauptspeicher ausgelagert werden, was jedoch nicht unterstützt wird. Stattdessen kann ein solches Programm derzeit nicht kompiliert werden. In den bisher mehr als 240 Testprogrammen tritt ein solcher Fall nur dreimal auf. Gibt es jedoch noch „freie" Register, für die die gesamte Registerbank durch andere Variablen blockiert, das einzelne Register jedoch nicht belegt ist, wird die gleiche Behebung wie im 2. Fall durchgeführt. Für alle Instruktionen, bei denen diese Registerbank durch eine andere als Operand verwendete Variable blockiert wird, wird die betroffene Variable in eine temporäre Variable kopiert, die dann als Operand der eigentlichen blockierenden Instruktion verwendet wird.

Da die Fehlerbehandlungen zum Teil weitere Konflikte einführen können, muss eventuell die Registerzuordnung neu ausgeführt werden. Um diesen Aufwand zu vermeiden, wird erst getestet, ob die Konfliktbehandlung überhaupt neue Konflikte eingeführt hat. Die Behebung des ersten Fehlerfalls führt z. B. keine weiteren Konflikte ein, da die eingefügte Delay-Instruktion die Kanten des Graphen nicht verändert und somit alle anderen Zuordnungen gültig bleiben. Da bei den Fehlerfällen zwei und drei jeweils neue Variablen eingefügt werden, muss der Graph neu eingefärbt werden, es sei denn, für die neuen Variablen gibt es passende freie Register. Somit wird, falls nötig, die Registerzuordnung und die anschließende Fehlerbehebung so oft ausgeführt, bis einerseits es keine Konflikte in der Registerzuordnung mehr gibt oder andererseits die Grenze der maximalen Schritte überschritten ist. Diese Grenze existiert, um eventuelle unendliche wiederholte Registerzuordnungen zu verhindern, für Fehler, die nicht richtig behandelt werden.

Mit dem aktuellen Algorithmus kann es bei komplexeren Kerneln vorkommen, dass keine gültige Registerzuordnung gefunden werden kann und somit das Programm nicht erfolgreich kompiliert werden kann. Um auch komplexere Programme zu unterstützten, kann der verwendete Algorithmus erweitert werden, indem Register Spilling angewendet wird. Anstatt lokale Variablen auf den langsamen Hauptspeicher auszulagern, kann (ähnlich der Optimierung der Speicherzugriffe in Abschnitt 2.2.5) der verfügbare VPM-Speicher verwendet werden, um dort Werte zwischenzuspeichern. Hierfür muss bei der Registerzuordnung bestimmt werden, welche Variablen in welchen Abschnitten des Programmcodes auf welchen Bereich der VPM ausgelagert werden soll. Dabei sollen die ausgelagerten Variablen so gewählt werden, dass deren Anzahl möglichst gering ist. Wie bei jedem Zugriff auf die VPM muss auch die Auslagerung von Variablen mit dem Hardware-Mutex abgesichert werden. Ebenso muss darauf geachtet werden, dass jede Variable pro QPU auf der das Programm ausgeführt wird, einen eigenen VPM-Bereich zugewiesen wird und auch, dass der für Speicherzugriffe verwendete VPM-Bereich keine ausgelagerten Variablen überschreibt. Aufgrund des hohen Aufwands, vor allem bei der Aufteilung der VPM, sowie der Tatsache, dass das Problem der fehlgeschlagenen

Registerzuordnung bei den verfügbaren Testprogrammen nur sehr selten auftritt, ist das Register Spilling in die VPM derzeit noch nicht implementiert.

2.3.3 Validierung

Nachdem die Labels nach Codepositionen aufgelöst und alle lokalen Variablen erfolgreich freien Registern zugewiesen sind, kann der erzeugte Maschinencode ausgegeben werden. Jedoch wird vor der Generierung des kompletten Binärcodes (siehe Abschnitt 2.3.4), falls so konfiguriert, der erstellte Maschinencode validiert. Dies geschieht mit einem externen Programm namens **vc4asm**, einem Assembler und Disassembler von Maschinencode für die VideoCore IV-Architektur von und zu einer domänenspezifischen Assembler-Sprache [Mü14]. Die Validierung wird hauptsächlich zur Überprüfung der richtigen Funktionsweise des Compilers durchgeführt und dient dazu, Fehler im erzeugten Code bereits bei der Kompilierung zu erkennen. Werden Fehler im Maschinencode nicht erkannt, schlägt zur Laufzeit die Programmausführung fehl, jedoch ohne das aufrufende Programm über die Art oder Stelle des Fehlers zu benachrichtigen, da die VideoCore IV GPU keine Mechanismen zur Fehlerbehandlung bietet. Im schlimmsten Fall friert die GPU und möglicherweise das ganze System ein und muss neu gestartet werden. Daher ist es wichtig, fehlerhaften Code erst gar nicht ausführen zu können. Die Hauptfunktionen von vc4asm (Assemblieren und Disassemblieren) werden zwar nicht verwendet, jedoch bietet das Programm eine sehr gute Validierung auf Missachtung der Besonderheiten der VideoCore IV GPU. Unter anderem werden folgende Regeln überprüft [Mü14] [Bro13, Abschnitt Summary of Instruction Restrictions]:

- Physikalische Register dürfen nicht in einer Instruktion geschrieben und in der nächsten bereits wieder gelesen werden.
- Zwischen zwei Branch-Instruktionen müssen mindestens drei Instruktionen Abstand liegen.
- Beide ALUs können nur in das gleiche Register schreiben, wenn sie inverse Bedingungen besitzen (z. B. Additions-Operation wird nur ausgeführt, wenn das Null-Flag gesetzt ist und Multiplikations-Operation wird nur ausgeführt, wenn das Null-Flag nicht gesetzt ist).
- Zwischen dem Schreiben in bestimmte Peripherie-Konfigurationen (z. B. für Uniforms, VPM, siehe Abschnitt 1.3.1) und dem Auslesen dieser Komponenten muss eine gewisse Anzahl an Instruktionen Abstand sein, bis die neue Konfiguration übernommen wird.
- Nach dem Thread-Ende Signal darf nicht mehr auf die Peripherie-Register geschrieben werden, da diese eventuell bereits für das nächste Programm vorbereitet werden.

2.3.4 Output

Der letzte Schritt eines jeden Compilers ist die Generierung des Codes der Zielsprache. Im Fall des VC4CC-Compilers kann Maschinencode für die VideoCore IV-Architektur in verschiedenen Formaten erzeugt werden. Je nach Konfiguration wird der Code im Hexadezimal-, Binärformat oder Assembler ausgegeben. Die Ausgabe im Hexadezimalformat wird v. a. in Tests verwendet, in denen das Ergebnis direkt in das Testprogramm eingebunden wird, um das Laden aus einer Datei zu überspringen. Die Anhänge A.5 bis A.7 zeigen das Ergebnis der Kompilierung des Code-Beispiels aus Anhang A.1 in Hexadezimal-Format. Hierbei wird zur besseren Lesbarkeit an jede Instruktion der jeweilige Assembler-Code angehängt. Der verwendete Assemblercode ist in einer selbst-erstellten Sprache geschrieben, die an die Assemblersprache aus dem vc4asm-Assembler (siehe Abschnitt 2.3.3) angelehnt ist. Die Darstellung in Binärcode wird von der VC4CL-Laufzeitbibliothek verwendet, um den kompilierten Kernel-Code auf die QPUs zu übertragen.

Unabhängig von dem verwendeten Format besteht der generierte Code aus drei Abschnitten: Der erste Abschnitt ist der Header. Dieser beginnt mit einer Magic Number, einer festgelegten Bitfolge zur Erkennung von Dateiformaten oder -inhalten. Die Magic Number für kompilierte Programme ist `0xDEADBEAF` und dient zur Erkennung, ob es sich um ein mit dem VC4CC-Compiler kompilierten Code handelt, sowie zur Bestimmung der Endianess (little-endian oder big-endian) des Codes. Gefolgt wird die Magic Number von den Informationen der einzelnen Kernel. Für jeden Kernel wird ein Block erstellt, der mit dem Offset der Kernel-Instruktionen vom Beginn des erzeugten Codes sowie die Anzahl der Instruktionen in dem Kernel beginnt. Des Weiteren wird der Name des Kernels gespeichert, der von der Implementierung der OpenCL-Laufzeitbibliothek (siehe Kapitel 4) verwendet wird, um den Kernelcode zu identifizieren. Ebenso werden für alle Parameter deren Namen gespeichert, sowie die Größe in Bytes und weiter, ob es sich um einen Zeiger handelt, ob der Parameter ein Eingangs- oder Ausgangsparameter ist und ob der Parameterwert konstant ist. Diese Informationen werden zum einen von der Laufzeitbibliothek benötigt, um das Setzen von Parameterwerten auf Fehler zu überprüfen und zum anderen bietet die Laufzeitbibliothek Funktionen an, um diese Informationen (z. B. den Parameternamen) auslesen zu können. Die Kernel-Metadaten werden bei von der Laufzeitbibliothek nach dem Kompilieren des Programms extrahiert (siehe Abschnitt 4.2.2). Der Header wird mit 64 0-Bits abgeschlossen. Darauf folgen die optionalen globalen Daten, die beim Start eines Kernels in einen speziellen Speicherbereich bitweise kopiert werden. Dieser Block wird ebenfalls mit einer 64-Bit Null abgeschlossen (siehe Anhang A.5, Zeile 17). Die Größe der globalen Daten wird nicht explizit gespeichert, kann jedoch aus dem Offset des ersten Kernels bestimmt werden. Der dritte Abschnitt besteht aus den Instruktionen der einzelnen Kernel. Hierbei werden die Instruktionen aller Kernel lückenlos aneinandergereiht. Der Be-

ginn und die Größe der einzelnen Kernel wird aus den Metadaten extrahiert. Bei der Ausführung eines Kernels wird der entsprechende Bereich in den Speicherbereich der QPUs kopiert (siehe Abschnitt 4.2.4).

In den Anhängen A.5 bis A.7 kann man gut die in Abschnitt 2.3.1 beschriebene Start- und Stopsegmente erkennen. In den Zeilen 2 bis 17 von Anhang A.6 werden die Parameter (inklusive der Informationen über das ausgeführte Work-Item und dessen Work-Group) aus dem speziellen Uniform-Register `unif` kopiert. Da für dieses Beispiel (die Berechnung der Fibonaccizahlen aus Anhang A.1) keine Informationen über das Work-Item und die Work-Group verwendet wird, werden diese Parameter verworfen, indem sie in das spezielle Register – verschoben werden. Nur die tatsächlich verwendeten expliziten Parameter der Kernel-Funktion (Zeilen 15 bis 17) werden in physikalische Register geschrieben. Ebenso kann man in den Zeilen 8 bis 11 des Anhangs A.7 gut das Stopsegment erkennen, in dem der Host-Interrupt `irq` geschrieben wird, gefolgt vom Auslösen des `thrend`-Signals zur Signalisierung des Endes des Threads sowie `nop`-Instruktionen zum Leeren der Pipeline.

3. OpenCL-Standardbibliothek

In diesem Kapitel wird die Implementierung der GPU-seitigen OpenCL C-Standardbibliothek sowie einigen weiteren OpenCL C-Features, die nicht direkt zur Standardbibliothek gehören, beschrieben.

3.1 Standard

Die OpenCL C-Standardbibliothek ist eine „ausgiebige Zusammenstellung eingebauter Funktionen für skalare und Vektoroperationen" [Gro12, Abschnitt 6.12]. Die Standardbibliothek ähnelt der C-Standardbibliothek in vielerlei Hinsicht: Sie muss von allen Implementierungen vollständig unterstützt werden, sowie einige Funktionen (vor allem mathematische Routinen) sind direkt aus der C-Standardbibliothek übernommen. Jedoch ist die OpenCL C-Standardbibliothek für SIMD-Prozessoren optimiert, was sich einerseits in den Vektoroperationen und andererseits in einer Vielzahl an speziellen Standardfunktionen zeigt. So bietet die OpenCL C-Standardbibliothek, wo es sinnvoll ist, für die skalare Variante einer Standardfunktion auch noch vektorisierte Varianten, die die gleiche Operation, jedoch mit Vektoren mit 2, 3, 4, 8 oder 16 Elementen durchführen. [Gro12, Abschnitt 6.12]

3.1.1 Aufbau

Die OpenCL 1.2-Spezifikation teilt im Abschnitt 6.12 die Funktionen der Standardbibliothek in mehrere Kategorien ein. Eine Auswahl dieser Kategorien ist im Folgenden beschrieben, alle weiter spezifizierten Funktionen können in der OpenCL 1.2-Spezifikation eingesehen werden.

Work-Item Funktionen dienen der Abfrage von Informationen über die aktuell ausgeführte Instanz eines Kernels. Alle Instanzen eines Kernels bekommen bei der Ausführung die gleichen Parameter übergeben. Jedoch arbeiten alle Instanzen meist auf verschiedenen Abschnitten der übergebenen Speicherbereiche, um eben die beinhalteten Operationen auf einer Vielzahl an Daten gleichzeitig ausführen zu können. Um den zugeordneten Speicherbereich bestimmen zu können, bietet die OpenCL-Standardbibliothek eine Liste von Funktionen an, die Informationen wie die aktuelle lokale oder globale ID, sowie die Offsets, die Anzahl der Work-Groups und die Größe der aktuellen Work-Group (d. h. die Anzahl der Work-Items

in der Work-Group) abfragen können. Die lokalen und globalen IDs können als Indizes in dem durch die Kernelausführung bestimmten Indexraum verstanden werden, wobei die lokale ID den Index in der jeweiligen Dimension innerhalb der aktuellen Work-Group und die globale ID den Index im gesamten Indexraum der Ausführung darstellt. So besitzt z. B. in einem dreidimensionalen Indexraum das fünfte Work-Item in der dritten Work-Group mit der Work-Group-Größe `(8, 2, 1)` die lokalen Indices `(4, 0, 0)` und die globalen Indices `(20, 4, 2)`. Die lokalen IDs berechnen sich aus dem Index der jeweiligen Kernelausführung, z. B. `(0, 0, 0)` für die erste und `(7, 0, 0)` für die achte Ausführung. Die globalen IDs ergeben sich aus den lokalen IDs, den globalen Offsets und den Indizes der derzeit ausgeführten Work-Group und werden über die Formel `global offset + Work-Group-Index * Work-Group-Größe + lokale ID` komponentenweise berechnet. Die globalen Offsets können bei der Ausführung des Kernels bestimmt werden und werden auf die globalen Indizes aufaddiert. Die Work-Item Funktionen sind in Abschnitt 6.12.1 des OpenCL 1.2-Standards beschrieben.

Die **mathematischen Routinen** nehmen einen großen Teil der Standardbibliothek ein und ähneln stark den mathematischen Funktionen in der C-Standardbibliothek. Um die auf den meisten OpenCL-Geräten vorhandene SIMD-Eigenschaft ausnutzen zu können, sind sämtliche mathematischen Funktionen für skalare Typen und Vektortypen definiert. Für jede mathematische Funktion ist ebenso ein maximaler Fehler definiert, den eine Implementierung einer solchen Funktion aufweisen darf (siehe Abschnitt 3.1.2). Ebenso werden einige Funktionen (unter anderem `exp2`, `log2`, `recip` und `sqrt`) in zwei weitere Varianten angeboten. Zum einen gibt es die Variante `half_xxx` (z. B. `half_exp2`), die die vorgeschriebene Genauigkeit des Ergebnisses deutlich lockert, um den Implementierungen schnellere, dafür ungenauere Verfahren für die Berechnung zu ermöglichen. Entgegen ihres Namens rechnet eine `half_xxx`-Funktion nicht auf dem Datentyp `half`, sondern auf Basis von `float`. Ebenso gibt es die Variante `native_xxx` (z. B. `native_log2`), die die Genauigkeit komplett den Implementierungen überlässt und somit erlaubt, beliebig ungenau (und dafür sehr viel schnellere) Berechnungen, z. B. durch spezielle Hardwareinstruktionen zu verwenden. Die Liste aller Funktionen und deren Verhalten ist in Abschnitt 6.12.2 des OpenCL 1.2-Standards spezifiziert. [Gro12, Abschnitt 6.12.2].

Eine weitere Kategorie sind die **Ganzzahlfunktionen**. Diese dienen weniger der Berechnung mathematischer Operation, dafür werden Funktionen wie `rotate` (für das Rotieren von Bits in einer Zahl), `upsample` (kombiniert zwei Zahlen eines kleineren Typen zu einer Zahl eines größeren Typen, indem das erste Argument als obere und das zweite Argument als untere Hälfte genommen wird) sowie `popcount` (zum Zählen der gesetzten Bits) definiert. Ebenso werden spezielle Funktionen spezifiziert, die auf gewisser Hardware bessere Performance als ihre "normalen" Varianten bieten können. Dazu zählt unter anderem auch `mul24`, die zwei 24-Bit breite Zahlen

multipliziert und auf der VideoCore IV GPU direkt in eine Hardwareinstruktion umgewandelt werden kann (siehe Abschnitt 1.3.2). [Gro12, Abschnitt 6.12.3]

Wie bereits in Abschnitt 3.1 genannt, bietet die OpenCL C-Standardbibliothek auch speziell für die Berechnung mit (mathematischen) Vektoren bestimmte Funktionen an. Darunter fallen die Kategorien **Geometrische Funktionen** und **Vektor Lade- und Speicherfunktionen** in den Abschnitten 6.12.5 und 6.12.7 des OpenCL 1.2-Standards. Die geometrischen Funktionen bieten nützliche Funktionen wie das Kreuzprodukt und Skalarprodukt zweier Vektoren, sowie die Berechnung der euklidischen Länge eines Vektors und eine Funktion zur Normalisierung eines Vektors. Die Lade- und Speicherfunktionen für Vektoren bieten die Möglichkeit, Vektoren von Daten aus Zeigern von skalaren Typen zu laden oder in skalare Zeiger zu speichern.

Des Weiteren werden in den Abschnitten 6.12.8 bis 6.12.11 Funktionen für den (synchronen) Speicherzugriff aus Kernel heraus definiert. Funktionen wie `barrier` und `mem_fence` dienen als Synchronisationspunkte. `barrier` garantiert, dass alle Instanzen eines Kernels an dieser Instruktion darauf warten, dass alle anderen Work-Items innerhalb derselben Work-Group ebenso den Code bis zu der Barriere ausgeführt haben, bevor sie die weitere Ausführung fortsetzen. Also wird garantiert, dass der Code vor der Barriere von allen Rechenkernen fertig ausgeführt ist, bevor der Code nach der Barriere ausgeführt werden kann. `mem_fence` hingegen stellt sicher, dass alle Änderungen an den Speicherbereichen vor der Instruktion abgeschlossen sind, bevor die Speicherzugriffe nach der Instruktion ausgeführt wird und verbietet somit u. a. dem Compiler, über diesen Befehl hinweg Speicherzugriffe umzusortieren. Abschnitt 6.12.10 des OpenCL-Standards definiert Funktionen für das Laden von gewissen Speicherbereichen. Zum einen asynchron, während der Kernel weiter ausgeführt wird und zum anderen Funktionen für das Holen der Daten in den Cache (`prefetch`), um darauf später schneller zugreifen zu können. Der Abschnitt 6.12.11 definiert die im Abschnitt 3.2.2 dieser Arbeit genauer beschriebenen Funktionen für atomare Berechnung auf Speicherbereichen. [Gro12, Abschnitte 6.12.8 bis 6.12.11]

Die letzte größere Kategorie von Standardfunktionen sind die Funktionen zum **Lesen und Schreiben von Bildern**, die im Abschnitt 6.12.14 des Standarddokuments spezifiziert sind. Der Standard bietet eine Vielzahl an Funktionen, um einzelne Pixel in Bildern verschiedener Formate (1D, 2D, 3D und Arrays von 1D und 2D Bilder) zu Lesen und zu Schreiben. Hierfür sind die Lese- und Schreibfunktion vielfach überladen, um eine Reihe von Farbkanälen (darunter Graustufen, Rot-Grün-Blau, Rot-Grün-Blau-Transparenz) und Datentypen (8-, 16- und 32-Bit Ganzzahlen, Fließkommazahlen, sowie spezielle Typen, wie auf $[0..1]$ genormte 8-Bit Farbwerte) unterstützen zu können. Die vollständige Liste der unterstützten Formate kann im OpenCL 1.2-Standard im Abschnitt 5.3.1.1 nachgesehen werden. Zum Lesen von

Pixelwerten können Funktionen mit oder ohne Angabe eines Samplers gewählt werden. Ein Sampler ist ein konstantes „Objekt", das gewisse Eigenschaften für das Lesen von Pixeln angibt, z. B. die Interpolationsmethode, wenn zwischen zwei Pixelwerten gelesen wird oder das Verhalten, wenn Koordinaten außerhalb des Bildes abgefragt werden. Des Weiteren werden noch Funktionen zum Abfragen von Bildeigenschaften (wie den Maßen, den Datentyp und die Farbkanäle) bereitstellt. Dies ist unter anderem nötig, da Bilder in OpenCL C ein sog. „opaque type" (ein „undurchsichtiger Typ") sind. Das bedeutet, dass der Standard keine interne Struktur des Bildtypen vorschreibt und es auch keine Möglichkeit gibt, diese zur Programmierzeit einzusehen. Dies ermöglicht es Implementierern der OpenCL-Standardbibliothek, eine eigene, für die jeweilige Hardware effiziente Datenstruktur zum Speichern der Bilddaten zu bestimmen. [Gro12, Abschnitt 6.12.4]

Die OpenCL C-Programmiersprache unterstützt zwar das Konvertieren via Casting von skalaren Typen, z. B. `int i; short s = (short)i`, jedoch nicht von Vektortypen. Hierfür werden eine Reihe von expliziten **Konvertierungsfunktionen** angeboten. Diese Funktionen besitzen die Signatur `convert_<dstTyp>(<srcTyp>)` und erlauben die Konvertierung zwischen allen nativen Datentypen mit der gleichen Vektorlänge, d. h. zur Konvertierung nach `int4` gibt es die Funktion `convert_int4()` mit Parametern von den Typen `char`, `uchar4`, `short4`, `ushort4`, `uint4` und `float4`. Für sämtliche Konvertierungen zwischen ganzen Zahlen und Fließkommazahlen existieren weitere Varianten der Konvertierungsfunktion, bei denen der Rundungsmodus explizit angegeben werden kann. Ebenso werden Varianten der Konvertierungsfunktion zwischen verschiedenen Ganzzahltypen definiert, die eine „gesättigte" Konvertierung erlauben, also bei Über- oder Unterlauf der Werte, diese auf den höchsten oder niedrigsten im Zieltyp repräsentierbaren Wert abgebildet werden, anstatt die überlaufenden Bits abzuschneiden. [Gro12, Abschnitt 6.2.3]

3.1.2 Numeric Compliance

Um auf verschiedenen Plattformen für die gleiche Berechnung die gleichen Ergebnisse zu bekommen und auch um den OpenCL-Programmierern zu garantieren, das mathematische Funktionen richtig berechnet werden, schreibt der OpenCL-Standard für die mathematischen Operationen und Funktionen der OpenCL C-Sprache und Standardbibliothek eine maximale Ungenauigkeit des Ergebnisses vor. Im Kapitel 7 des OpenCL 1.2-Standards [Gro12] wird das erwartete numerische Verhalten spezifiziert. Da für Ganzzahloperationen eine exakte Berechnung erwartet wird, beziehen sich die numerischen Vorgaben im OpenCL 1.2-Standard nur auf Fließkommaberechnungen. OpenCL schreibt von den in IEEE 754 (der Spezifikation für die `float` und `double` Fließkommaformate) definierten Rundungsmodi nur einen vor, nämlich das Runden zur nächsten geraden Zahl. Das `EMBEDDED_PROFILE`

lockert diese Anforderung, indem es auch das Runden zur Null hin (Abschneiden der nicht repräsentierbaren Stellen) erlaubt, welches auch von den Instruktionen der VideoCore IV nativ ausgeführt wird (z. B. beim Umwandeln einer Fließkommazahl in eine Ganzzahl mit der Instruktion `ftoi`). Ebenso erzwingt das `FULL_PROFILE` die Unterstützung von Unendlich und Not-a-Number, was auch vom `EMBEDDED_PROFILE` gelockert wird. Weiterhin definiert der OpenCL 1.2-Standard für jede mathematische Funktion der Standardbibliothek sowie für die Fließkommaoperationen den maximalen erlaubten Fehler (als Abweichung vom „exakten" Wert) in ULP. Ein ULP (Unit in the Last Place) ist der „Unterschied zweier benachbarten Fließkommazahlen" [Gro12, Abschnitt 7.4]. Für Ganzzahltypen wäre ein ULP immer eins, da unabhängig der Größenordnung alle ganzen Zahlen dargestellt werden können. Dies ist jedoch für Fließkommazahlen nicht der Fall, da diese im Format $S * M \times 2^E$ gespeichert sind, mit dem Vorzeichen S (1 oder -1), der Mantisse M und dem Exponent E. Für den `float` Datentypen ist die Mantisse M 23 Bit und der Exponent E 8 Bit lang. Daraus ergibt sich ein ULP von $2^{-23} \times 2^E$, das von dem Exponenten E abhängig ist ($ulp(2^0) \approx 1.19 \times 10^{-7}$ und $ulp(2^{20}) = 0.125$). Die genaue Definition der maximalen erlaubten Fehler der definierten mathematischen Funktionen kann in den Tabellen 7.1 und 10.1 des OpenCL 1.2-Standards für das komplette und das eingebettete Profil eingesehen werden und unterscheiden sich nur minimal. Anzumerken ist, dass für die nativen Varianten mancher mathematischen Funktionen (siehe Abschnitt 3.1.1) kein maximaler Fehler angegeben ist („Implementation-defined"), die Genauigkeit des Ergebnisses also frei in der Hand des Implementierers liegt. Ebenso besitzen die Varianten mit halber Genauigkeit einen erlaubten maximalen Fehler von 8192 ULP, d. h. sie müssen nur bis zur zehnten Binärstelle genau sein. Des Weiteren wird das Verhalten bestimmter mathematischen Funktionen in Grenzfällen (Parameter oder Ergebnis sind keine gültigen Zahlen oder Unendlich) vorgegeben. Die genauen Vorschriften hierfür stehen in Abschnitt 7.5.1 des OpenCL-Standards. [Gro12, Kapitel 7, 10]

3.2 Implementierung

Die OpenCL-Standardbibliothek wird im Projekt **VC4CLStdLib** implementiert. Sie besteht aus einer Reihe von OpenCL C-Headern, welche die Funktionen der Standardbibliothek nach den bereits in Abschnitt 3.1.1 erwähnten Kategorien aufteilen. Eine weitere Header-Datei `VC4CLStdLib.h` bindet alle der Bibliotheksheader ein und bildet somit Zugriff auf die gesamte Implementierung der Standardbibliothek (siehe Abschnitt 2.1.1). Die Implementierung der Standardfunktionen geschieht in direkt in den Header-Dateien, d. h. die Funktionen werden dort deklariert und definiert. Der große Vorteil der direkten Einbindung der Funktionsrümpfe in den Quellcode ist, dass diese somit zum einen vom Precompiler (z. B. LLVM) auf Syntax-

und Semantikfehler überprüft werden und zum anderen auch bereits optimiert werden können, da dem Precompiler die Funktionsrümpfe bekannt sind. Ausgenommen sind hier nur die Deklarationen der intrinsischen Funktionen (siehe Abschnitte 2.2.3 und 3.2.3). Um den Programmieraufwand einfach zu halten werden viele Makros verwendet. Da, wie bereits in Abschnitt 3.1 erwähnt, in der OpenCL C-Standardbibliothek viele der Funktionen für skalare und Vektortypen definiert sind, müssen die Implementierungen dieser Funktionen auch für skalare Typen und Vektortypen geschrieben werden. Um, vor allem bei einfachen Funktionen, diese Arbeit zu umgehen, werden die Vektorversionen der Funktionen automatisch durch Makros generiert. Listing 3.1 zeigt ein Beispiel, wie 36 Funktionen aus einer Zeile Code generiert werden. Die 36 Funktionen ergeben sich aus sechs Typen (`uchar`, `char`, `ushort`, `short`, `uint` und `int`) mit jeweils sechs Vektorgrößen (1, 2, 3, 4, 8 und 16 Elemente). Das verwendete Macro `SIMPLE_INTEGER_3` ist wie alle weiteren Macros zum generieren für Funktionen in der Datei `_overloads.h` definiert.

```
1  SIMPLE_INTEGER_3(clamp, val, minval, maxval,
2    min(max(val, minval), maxval)
3  )
4  //generates:
5  uchar clamp(uchar val, uchar minval, uchar maxval) {
6    return min(max(val, minval), maxval);
7  }
8  uchar2 clamp(uchar2 val, uchar2 minval, uchar2 maxval) {
9    return min(max(val, minval), maxval);
10 }
11 uchar3 clamp(uchar3 val, uchar3 minval, uchar3 maxval) {
12   return min(max(val, minval), maxval);
13 }
14 uchar4 clamp(uchar4 val, uchar4 minval, uchar4 maxval) {
15   return min(max(val, minval), maxval);
16 }
17 uchar8 clamp(uchar8 val, uchar8 minval, uchar8 maxval) {
18   return min(max(val, minval), maxval);
19 }
20 uchar16 clamp(uchar16 val, uchar16 minval, uchar16 maxval) {
21   return min(max(val, minval), maxval);
22 }
23 // [...] the same for char ... char16, ushort ... ushort16,
     short ... short16, uint ... uint16 and int ... int16
```

Listing 3.1: Beispiel der Generierung von Funktionen durch Makros

In den folgenden Abschnitten wird für einige nennenswerte Kategorien aus Abschnitt 3.1.1 die Implementierung mit eventuellen Besonderheiten beschrieben. Die

Implementierung der nicht erwähnten Funktionen ist meist sehr einfach gehalten, indem die Berechnung in einigen wenigen OpenCL C-Instruktionen geschieht oder direkt an eine der definierten intrinsischen Funktionen weitergeleitet wird.

Wie bereits in den Abschnitten 2.2.3 und 2.3.1 beschrieben, müssen die Informationen über die Work-Item bei der Ausführung von OpenCL-Kernels per Parameter an diese übergeben werden, da sie erst beim Starten des Kernels berechnet werden. Deshalb werden diese Funktionen durch intrinsische Funktionen implementiert (siehe Abschnitt 2.2.3), die die speziellen „versteckten" Parameter für die jeweilige Information (z. B. lokale/globale ID oder Offset) auslesen und zurückgeben.

Die Instruktion `barrier` wird in einen OpenCL-Kernel eingefügt, um den Programmfluss zwischen den verschiedenen Instanzen zu Synchronisieren. Genauer: es muss sichergestellt werden, dass jede Ausführung solang blockiert, bis alle anderen Ausführungen die `barrier`-Instruktion erreicht haben. Hierfür werden zwölf der 16 verfügbaren Hardwaresemaphore verwendet. Dazu inkrementiert jedes, außer dem ersten Work-Item, das erste Semaphore, wartet darauf (durch Dekrementieren), dass das jeweils eigene Semaphore inkrementiert wird und inkrementiert danach das nächstfolgende Semaphore. Somit gibt jedes Work-Item das darauf folgende frei, nachdem es selber freigegeben wurde. Das erste Work-Item hingegen dekrementiert das erste Semaphore sooft, wie es andere Work-Items gibt, die in dieser Work-Group ausgeführt werden und gibt dann das zweite Semaphore frei. Diese Implementierung garantiert, dass das erste Work-Item erst freigegeben wird (und damit erst dann das zweite Work-Item freigibt), wenn alle anderen Work-Items die `barrier`-Instruktion erreicht haben.

Für das asynchrone Kopieren von Speicherbereichen, das im OpenCL 1.2-Standard im Abschnitt 6.12.10 definiert ist, wird ein Trick verwendet: Zum einen wird nicht asynchron, sondern synchron kopiert, d. h. die QPUs blockieren während des Kopiervorgangs. Ebenso wird nicht, wie im Standard beschrieben „das asynchrone Kopieren [...] von allen Work-Items einer Work-Group" [Gro12, Abschnitt 6.12.10] ausgeführt, sondern nur vom ersten Work-Item der Work-Group. Diese Idee wurde von der hostseitigen OpenCL-Implementierung pocl [pd10] übernommen und hat den Vorteil, das der zu kopierende Speicherbereich nicht auf die QPUs aufgeteilt werden muss. Außerdem müssten die QPUs sowieso für die Speicherzugriffe synchronisiert werden (siehe Abschnitt 1.3.1), da keine parallelen Speicherzugriffe stattfinden können. Mit der Lösung, dass nur das 1. Work-Item die Kopie durchführt, können die anderen weiter rechnen.

3.2.1 Mathematische Routinen

Die zu implementierenden mathematischen Funktionen der Standardbibliothek lassen sich in zwei Kategorien einteilen: Die einen Funktionen lassen sich exakt

berechnen, d. h. unter der Voraussetzung, dass alle in der Implementierung verwendeten Funktionen und Operationen exakt sind, wird für alle Parameter das korrekte Ergebnis berechnet. Darunter zählen z. B. `ceil` zum Aufrunden einer Fließkommazahl auf die nächste ganzen Zahl, `copysign` zum Übertragen des Vorzeichen auf eine andere Fließkommazahl und `fabs` zum Berechnen des Betrags einer Fließkommazahl. Diese Funktionen sind meist sehr einfach gelöst und werden deshalb nicht weiter betrachtet. Die andere Kategorie von mathematischen Funktionen lassen sich gar nicht, oder zumindest nicht mit der gegebenen Hardware, exakt berechnen. Darunter fallen z. B. die trigonometrischen Funktionen `sin`, `cos` und `tan`, sowie die Berechnungen des Logarithmus und der Exponentialfunktion. Der Grundgedanke bei der Implementierung solcher Funktionen ist es, einen (meist numerischen) Algorithmus zu finden, der sich auf der VideoCore IV-Hardware effizient umsetzen lässt und für alle erlaubten Parameterwerte ein Ergebnis liefert, das exakt genug ist (d. h. dessen Fehler kleiner als die spezifizierte Grenze ist, siehe Abschnitt 3.1.2). Wobei hierbei die Effizienz der Implementierung eine kleinere Rolle spielt als die Genauigkeit der Ergebnisse, da diese vom OpenCL-Standard vorgegeben wird. Bei der Suche nach effizienten Algorithmen hilft es, dass die meisten der mathematischen Funktionen der OpenCL-Standardbibliothek in den Standardbibliotheken vieler weiteren Programmiersprachen (und für eine Vielzahl an Plattformen) bereits implementiert sind und es so bereits vorhandene und auch auf Genauigkeit überprüfte Implementierungen für die meisten Funktionen gibt. Jedoch muss bei der Auswahl des Algorithmus unter anderem zusätzlich beachtet werden, dass manche Implementierungen auf Basis von Tabellen von Konstanten basieren, in denen Teilergebnisse nachgeschlagen werden, anstatt diese zu berechnen. Tabellenbasierte Algorithmen sind jedoch für eine Implementierung auf der VideoCore IV GPU ungeeignet, da diese auf den Hauptspeicher ausgelagert werden müssten und der Zugriff auf diese Werte die Performance erheblich beeinträchtigt. Ebenso bietet der VideoCore IV-Befehlssatz keinen Maschinenbefehl für z. B. Ganzzahl- oder Fließkommadivision, wodurch Implementierungen, die auf diese Operationen basieren, erheblich langsamer werden (siehe Abschnitt 2.2.4). Weiterhin sind Implementierungen ungeeignet, die auf bedingten Sprüngen (`if`-Anweisung) oder dynamischen Schleifen (`while`-Anweisungen mit den Operanden als Parametern) basieren. Die OpenCL C-Standardbibliothek verlangt für jede mathematische Funktion skalare und vektorielle Implementierungen. Da jedoch Bedingungen von `if` oder `while`-Anweisungen immer skalar sein müssen, kann somit nicht pro Element bestimmt werden, ob ein Sprung getätigt wird oder nicht. `if`-Anweisungen lassen sich jedoch zum Teil durch den trinären Operator `?:` ersetzen, da dieser die Bedingung elementweise testet. Jedoch hat der trinäre Operator den Nachteil, dass immer beide möglichen Werte berechnet werden müssen und am Ende für jedes Vektorelement entschieden wird, welcher der Werte verwendet wird.

Die Tabelle 3.1 listet eine Auswahl der implementierten mathematischen Funktion,

Funktion	Algorithmus	Genauigkeit	max. Fehler
atan	Padé-Approximation mit Argumentreduktion	$1.2 * 10^{-7}$	$5.9 * 10^{-7}$
cos	Taylor-Serie mit Argumentreduktion	$2 * 10^{-7}$	$4.7 * 10^{-7}$
cosh	Taylor-Serie mit Argumentreduktion	$1.5 * 10^{-7}$	$4.7 * 10^{-7}$
erf	numerische Approximation	$1.2 * 10^{-7}$	$1.9 * 10^{-6}$
exp	Padé-Approximation mit Argumentreduktion	$1.2 * 10^{-7}$	$4.7 * 10^{-7}$
lgamma	numerische Approximation	$1.2 * 10^{-7}$	nicht definiert
log	Taylor-Serie mit Argumentreduktion	$4.1 * 10^{-7}$	$4.7 * 10^{-7}$
rsqrt	Newton-Verfahren mit angenähertem Initialwert	$1.2 * 10^{-7}$	$4.7 * 10^{-7}$
sin	Taylor-Serie mit Argumentreduktion	$1.2 * 10^{-7}$	$4.7 * 10^{-7}$
sinh	Taylor-Serie	$2 * 10^{-7}$	$4.7 * 10^{-7}$
sqrt	Taylor-Serie mit initialem Schätzwert	$1.2 * 10^{-7}$	$4.7 * 10^{-7}$

Tabelle 3.1: Auswahl der Implementierung mathematischer Funktionen

die Art der Implementierung sowie den maximalen Fehler der Implementierung und den maximalen erlaubten Fehler des OpenCL 1.2-Standards. Die Fehler (sowohl der Implementierung als auch die Maximalwerte im Standard) sind als relative Fehler angegeben. Wie bereits in Abschnitt 3.1.2 beschrieben, lässt sich der im OpenCL 1.2-Standard in den Kapiteln 7 und 10 in ULP angegebene maximaler erlaubter Fehler in den relativen Fehler umwandeln, indem $ulp(x)/x = (2^{-23} \times x)/x = 2^{-23} \approx 1.19 \times 10^{-7}$ berechnet wird. Die Angaben der maximalen Fehler der jeweiligen Implementierungen stammen in den meisten Fällen aus der Quelle, in der auch der Algorithmus beschrieben ist. In den anderen Fällen (z. B. `cos` und `sin`) wurde der maximale Fehler numerisch berechnet. Die maximal erlaubten Fehler der Funktionen `lgamma` und `lgamma_r` zum Berechnen des Logarithmus der Gamma-Funktion sind im OpenCL 1.2-Standard nicht definiert [Gro12, Tabelle 7.1, Fußnote 69].

Bei den in Tabelle 3.1 aufgeführten Algorithmen, für die „Argumentreduktion" angegeben ist, wird der Wertebereich der Parameter in einen kleineren Bereich umgewandelt, für den sich die Funktion (einfacher) berechnen lässt. So kann z. B. der Zweierlogarithmus einer Fließkommazahl effizienter berechnet werden, indem $log2(x) = log2(M \times 2^E) = E + log2(M)$, wobei der Exponent `E` über Bitoperationen extrahiert werden kann und sich die Mantisse `M` nur noch im Wertebereich $1 \leq M < 2$ befindet, für das effiziente Algorithmen zur Annäherung des Logarithmus

existieren. Ebenso lassen z. B. sich die trigonometrischen Funktionen Sinus und Cosinus vereinfachen, indem die Äquivalenzen $sin(x) = sin(x \pm 2\pi)$ und $cos(x) = cos(x \pm 2\pi)$ verwendet werden, um den Parameter in den Wertebereich $-\pi \leq x \leq \pi$ zu bringen, für den die Taylor-Serie bereits nach acht Iterationen ein genügend genaues Ergebnis liefert. Der „initiale Schätzwert", z. B. bei der Berechnung der Quadratwurzel wird durch die SFU-Recheneinheit berechnet (siehe Abschnitt 1.3.1).

Die Tabelle 3.1 listet nur mathematische Funktionen auf, deren Implementierung mithilfe eines umfangreicheren Algorithmus umgesetzt ist. Viele der weiteren mathematischen Funktionen sind entweder sehr einfach gelöst (z. B. die Funktionen zum Runden: `ceil`, `floor`, `round`, `rint`, `trunc`) oder werden durch Verwendung der Algorithmen in Tabelle 3.1 implementiert. Ähnlich der Argumentreduktion werden hier mathematische Beziehungen zwischen den Funktionen ausgenutzt. So werden z. B. `erfc(x)` auf `1 - erf(x)`, `exp10(x)` auf `exp(x * log(10))` und `acos(x)` auf $\pi/2$ `- asin(x)` abgebildet. Die Implementierung aller in der OpenCL C-Standardbibliothek festgelegten mathematischen Funktionen kann in der Header-Datei `_math.h` des VC4CLStdLib-Projektes eingesehen werden. Hier sind auch für die meisten Funktionen der verwendete Algorithmus, die Quelle des Algorithmus sowie der maximale Fehler dokumentiert. Die Algorithmen zur Berechnung der in Tabelle 3.1 aufgelisteten Funktionen stammen aus zum Teil von verschiedenen anderen Softwareimplementierungen (wie pocl [pd10] oder glibc, der C-Standardbibliothek des GCC-Compilers). Andere Algorithmen basieren auf mathematischen Werken wie Numerical Recipes [PFTV92] oder Matters Computational [Arn10]. Ein Beispiel, wie mathematische Funktionen implementiert sind, kann in Anhang A.8 anhand der Implementierung des natürlichen Logarithmus eingesehen werden. Hierbei definiert `COMPLEX_1` wieder die Funktionen für skalare und vektorielle Fließkommatypen und das Macro `result_t` wird auf den jeweils verwendeten Typen gesetzt, z. B. `float` für die skalare Variante oder `float16` für Vektoren mit 16 Elementen. Somit muss auch für etwas komplexere Funktionen nur einmal Code geschrieben werden, der Funktionen für die verschiedenen Vektorvarianten generiert.

Wie bereits in Abschnitt 3.1.1 beschrieben, definiert die OpenCL C-Standardbibliothek für manche häufig verwendeten mathematischen Funktionen Versionen mit eingeschränkter oder nicht fest definierter Genauigkeit. Die Varianten mit halber Genauigkeit (z. B. `half_exp`) als auch die Varianten mit undefinierter, geräteabhängiger Genauigkeit (z. B. `native_sin`) werden gleichermaßen implementiert. Für Funktionen, für die es eine Instruktion in der SFU (siehe Abschnitt 1.3.1) gibt, namentlich die Berechnung des multiplikativen Inverses, des Zweierlogarithmus und der Exponentialfunktion auf Basis von zwei werden über die SFU berechnet. Andere mathematischen Funktionen werden mithilfe der jeweiligen Version mit voller Genauigkeit umgesetzt.

3.2.2 Atomare Operationen

Der OpenCL 1.2-Standard spezifiziert eine Reihe von atomaren Funktionen [Gro12, Abschnitt 6.12.11]. Diese Funktionen führen mathematische Berechnungen auf Speicherbereichen des lokalen oder globalen Speichers durch und sind nur für 32-Bit Ganzzahlen definiert. Zu den atomaren Operationen gehören die Addition/Subtraktion, Inkrementierung/Dekrementierung, Minimums-/Maximumsberechnung, das bitweise Und, Oder und Exklusives Oder sowie Funktionen zum Austauschen von Werten und das vorherige Vergleichen (compare-and-swap). Der OpenCL 1.2-Standard bietet keine Definition der „Atomarität" dieser Funktionen. Jedoch besagt die Spezifikation der offiziellen Erweiterung für 64-Bit atomare Funktionen (`cl_khr_int64_base_atomics`), dass „diese Transaktionen für das [OpenCL] Gerät, auf dem diese atomaren Funktionen ausgeführt werden, atomar sind. Es gibt keine Garantie, der Atomarität, wenn atomare Operationen auf demselben Speicherbereich von verschiedenen [OpenCL] Geräten aus ausgeführt werden" ([Gro15, Abschnitt 9.3]). Dies bedeutet für die Implementierung auf dem Raspberry Pi, dass die Atomarität nur für Zugriffe über die VideoCore IV GPU und nicht auch von hostseitigen Zugriffen auf den verwendeten Speicherbereich garantiert werden muss.

Implementiert werden die atomaren Funktionen (wie bereits in Abschnitt 2.2.3 angedeutet) mithilfe des globalen Hardware-Mutex. Anstatt, wie bei „normalen" Speicherzugriffen, das Auslesen und Beschreiben des Speicherbereichs separat über das Hardware-Mutex zu sichern, wird der ganze Funktionsaufruf mit dem Mutex synchronisiert. Da für die VideoCore IV GPU nur ein Mutex zur Verfügung steht, wird so automatisch die Atomarität innerhalb der Zugriffe aus allen QPUs heraus gewährleistet. Listing 3.2 zeigt ein Beispiel einer atomaren Funktion zum Inkrementieren einer 32-Bit Ganzzahl. Die Makros `INLINE` und `OVERLOADABLE` werden zu Compilerdirektiven expandiert, die für die Funktionsweise nicht weiter relevant sind. Da „normalen" Speicherzugriffen automatisch mit dem Hardware-Mutex abgesichert werden, werden für die atomaren Funktionen die intrinsischen Funktionen `vc4cl_dma_read` und `vc4cl_dma_write` verwendet, die einen Speicherzugriff explizit ohne Synchronisation durchführen. Dies ist nötig, da der gesamte Funktionsrumpf bereits über die intrinsischen Funktionen `vc4cl_mutex_lock` und `vccl_mutex_unlock` gesichert wird.

3.2.3 Intrinsics

Wie bereits in Abschnitt 2.2.3 beschrieben, besitzt die VC4CL-Standardimplementierung eine Vielzahl an intrinsischen Funktionen. Damit diese intrinsischen Funktionen aus den Implementierungen der Funktionen der OpenCL-Standardbibliothek heraus aufgerufen werden können und der Code auch vom CLang-Compiler kompiliert werden kann (siehe Abschnitt 3.2), müssen diese Funktionen in den OpenCL

```
1 INLINE int atomic_inc(volatile __global int * ptr)
    OVERLOADABLE
2 {
3   vc4cl_mutex_lock();
4   int old = vc4cl_dma_read(ptr);
5   vc4cl_dma_write(ptr, old + 1);
6   vc4cl_mutex_unlock();
7   return old;
8 }
```

Listing 3.2: Implementierung der Standardfunktion zum atomaren inkrementieren einer Ganzzahl

C-Headern deklariert werden. Dies geschieht in der Header-Datei `_intrinsics.h`. Die verschiedenen Arten von intrinsischen Funktionen und deren Umsetzung innerhalb des Compilers sind in Abschnitt 2.2.3 beschrieben.

Wie bereits in Abschnitt 2.2.3 erwähnt, werden intrinsische Funktionen meistens verwendet, um spezielle Maschinenbefehle für die höhere Programmiersprache verfügbar zu machen oder Funktionen für die jeweilige Zielarchitektur zu optimieren (z. B. mathematische Routinen). Da der VC4CC-Compiler die Stärken und Schwächen sowie die Effizienz der Zielarchitektur kennt, würde es sich anbieten, möglichst viele Funktionen als intrinsische Funktionen zu implementieren, um diese dann im Compiler effizient umsetzen zu können. Jedoch haben intrinsische Funktionen den Nachteil, dass ein anderer Compiler (wie in dieser Arbeit der LLVM-Compiler für die Vorkompilierung, siehe Abschnitt 2.1.1) die Semantik dieser Funktionen nicht kennt und diese daher auch nicht optimieren kann. So ist es zum Beispiel in der Implementierung der OpenCL-Standardbibliothek für die VideoCore IV GPU sinnvoll, eine intrinsische Funktion für das Maximum zweier Fließkommazahlen `vc4cl_fmax` anzubieten, da einerseits es einen solchen Maschinenbefehl in der VideoCore IV-Architektur gibt, auf den die Funktion abgebildet werden kann, und andererseits alternative Implementierungen (z. B. `x > y ? x : y`) immer mehrere Taktzyklen benötigen. Jedoch kann der LLVM-Compiler einen Aufruf `vc4cl_fmax(0.5f, 1.0f)` nicht optimieren, während ein Aufruf mit der alternativen Implementierung `0.5f > 1.0f ? 0.5f : 1.0f` vom Compiler optimiert werden kann, da alle Werte und auch die Semantik der Operatoren bekannt sind. Diese Undurchsichtigkeit und damit Unoptimierbarkeit sollte beim Schreiben von intrinsischen Funktionen beachtet werden. Daher gibt es in der Implementierung der OpenCL-Standardbibliothek nur solche intrinsischen Funktionen, bei denen der Compiler sowieso die Semantik nicht kennt (z. B. Zugriff auf Bilder sowie Peripherie oder zur Laufzeit gesetzte Werte), die intrinsische Implementierung auf einen einzelnen Maschinenbefehl abgebildet wird (z. B. Minimums- und Maximumsfunktionen) oder eine Implementierung in reinem OpenCL C-Code nicht möglich oder erlaubt

ist (z. B. Typ-Konvertierung von `char8` auf `int8`) sowie für Operationen, die es in der OpenCL C-Sprache nicht gibt (wie Count-Leading-Zeroes und die 24-Bit Multiplikation).

3.2.4 Nicht unterstützte Funktionen

Wie bereits in Abschnitt 1.2.4 genannt, unterstützt diese Implementation aus verschiedenen Gründen nur das eingebettete Profil der OpenCL 1.2-Spezifikation, das eine abgespeckte Version des Standards darstellt. Grund dafür ist unter anderem, das bestimmte Datentypen und Operation nicht unterstützt werden müssen. Darunter fallen die Datentypen `double` und `long` für 64-Bit Fließkomma- und Ganzzahlarithmetik. Diese werden nicht unterstützt, da die Hardware der VideoCore IV GPU keine 64-Bit großen Register oder Instruktionen bereitstellt, die mit 64-Bit Operanden arbeiten können (siehe Abschnitt 1.3.2). Weiterhin ist das Verhalten für Fließkommaoperationen an den Grenzwerten und im Fehlerfall nicht definiert. D. h. eine Implementierung eines eingebetteten Profils, muss positives und negatives Unendlich, sowie `NaN` nicht richtig handhaben können. Dies erspart einige Instruktionen, die anderweitig nötig wären, um das Grenzwertverhalten sowie das Verhalten von `NaN` als Eingabe richtig zu implementieren. Das korrekte Verhalten kann eventuell nachträglich implementiert werden (siehe auch Kapitel 6).

OpenCL 1.2 spezifiziert eine `printf`-Funktion, die, ähnlich zur C-Variante, ein Formatierungstext und optionale Parameter entgegennimmt, diese in den Formatierungstext einfügt und auf die Standardausgabe ausgibt. Da jedoch die `printf`-Funktion in OpenCL auf einem OpenCL-Gerät (z. B. einer GPU) ausgeführt wird, die keinen Zugriff auf die Standardausgabe hat, gibt es die Einschränkungen, dass eine `printf`-Implementierung im OpenCL-Standard die Ausgabe erst nach Beendigung der Kernelausführung schreiben muss. Um die `printf`-Funktion zu implementieren würde es sich eignen, beim Start eines Kernels einen globalen Buffer im Speicherbereich der GPU anzulegen (siehe Abschnitt 4.2.1), dessen Adresse und Größe ähnlich dem Work-Item-Eigenschaften als „versteckte" Parameter zu übergeben (siehe Abschnitt 2.3.1), GPU-seitig in diesen Buffer zu schreiben (ähnlich der C-Funktion `sprintf`) und diesen Buffer nach der Ausführung auszulesen und auf der Standardausgabe auszugeben. Die Schwierigkeit bei der Implementierung ist jedoch nicht, den formatierten Text von der GPU auf die Standardausgabe zu bekommen, sondern die zusätzlichen Parameter korrekt in den Text einzufügen. Hierfür müssten Hilfsfunktionen geschrieben werden, die z. B. ganze oder Fließkommazahlen aller unterstützter Typen auf Texte fester (oder dynamischer) Länge abbilden können. Da in bisher keiner der über 240 zusammengetragenen Kernel zum Testen des VC4CC-Compilers die `printf`-Funktion verwendet wird, wird dieser Aufwand erst einmal zurückgestellt.

Ebenso nicht unterstützt wird die Bildbearbeitung jeglicher Art. Aufgrund der ausgiebigen Funktionen der OpenCL C-Standardbibliothek für den Zugriff auf Bilder, ist eine vollständige Implementierung aller Funktionen der Bildbearbeitung sehr komplex [Gro12, Abschnitt 6.12.14]. Vor allem die Funktionen zum Lesen der einzelnen Bildpunkte sind sehr aufwendig. Zum einen durch die Option der Interpolation von Werten zwischen gespeicherten Bildpunkten und zum anderen wegen der Eigenschaft, dass beim Lesen und Schreiben eines Bildes die Implementierung eine Konvertierung zwischen allen unterstützten Datentypen der Bildpunkte bereitstellen muss. Daher wird die optionale Komponente der Bildbearbeitung derzeit nicht unterstützt. Es sind bereits manche Funktionen für die Bildbearbeitung experimentell umgesetzt, wobei diese bisher ungetestet und nicht vollständig sind und standardmäßig deaktiviert sind.

3.3 Implementierte Erweiterungen

OpenCL ist per Spezifikation erweiterbar und es gibt auch eine Reihe offiziell von Khronos definierter Erweiterungen, wie die Unterstützung von 16-Bit Fließkommazahlen und das Austauschen von Objekten zwischen OpenCL und OpenGL (siehe auch Abschnitt 1.2.4 und die Spezifikation der offiziellen Erweiterungen [Gro15]). Implementierte Erweiterungen, die keine Auswirkungen auf die Ausführung von OpenCL-Kernels haben, da sie keine weiteren Macros, Funktionen oder Typen für die OpenCL C-Programmiersprache und Standardbibliothek definieren, werden im Abschnitt 4.3 behandelt. Darunter zählen z. B. der Khronos ICD Loader und auch das Erstellen von Programmen aus SPIR-V-Zwischencode. Andere Erweiterungen definieren neue Macros oder Funktionen, die auf den OpenCL-Gerät zur Verfügung stehen und werden daher im Kapitel zur OpenCL-Standardbibliothek behandelt. Der erste Satz von implementierten Erweiterungen sind die atomaren Operationen auf dem Hauptspeicher. Die folgenden Erweiterungen müssen von allen Geräten unterstützt werden, die OpenCL 1.2 unterstützen [Gro12, Tabelle 4.3]:

- `cl_khr_global_int32_base_atomics`
- `cl_khr_global_int32_extended_atomics`
- `cl_khr_local_int32_base_atomics`
- `cl_khr_local_int32_extended_atomics`

Genauer gesagt, wurden diese Erweiterungen für OpenCL 1.0 definiert und sind mit der OpenCL 1.1-Spezifikation teil der Standardbibliothek geworden [Gro12, Anhang F.1]. Die leicht anders benannten Funktionen aus den Erweiterungen werden zur Abwärtskompatibilität unterstützt und sind in der VC4CL-Implementierung nur

eine Weiterleitung auf die atomaren Funktionen der Standardbibliothek (siehe Abschnitt 3.2.2). Ebenso ist die Erweiterung `cl_khr_byte_addressable_store`, die das Byte-weise adressieren von Speicherbereichen erlaubt, mit OpenCL 1.1 in den Standard übernommen worden und muss somit auch unterstützt werden.

Des Weiteren werden noch optionale Erweiterungen verschiedener Hersteller unterstützt: NVIDIA spezifiziert die sehr kleine Erweiterung `cl_nv_pragma_unroll` [WAG17], die ein Compiler-Pragma `#pragma unroll` mit einem optionalen Faktor definiert, das vor Schleifen positioniert werden kann, um dem Compiler den Hinweis zu geben, diese Schleifen aufzurollen, d. h. in sequentielle Instruktionen umzuformen. Der optionale Faktor gibt hierbei an, wie oft aufgerollt werden soll. Ist er nicht angegeben, kann der Compiler versuchen, die gesamte Schleife in eine lineare Form zu bringen. Das Aufrollen von Schleifen („loop unrolling" oder auch „loop unwinding") wird in Compilern verwendet, um vor allem bei Schleifen, bei denen die Anzahl der Iterationen fest steht (z.B: bei `for`-Schleifen mit konstantem Startwert, Endwert und Schritt), die anderweitig benötigten Sprünge zurück an den Anfang des Schleifenrumpfs zu entfernen, um Sprünge und somit Instruktionen zu sparen. Das Pragma `#pragma unroll` wird vom LLVM-Compiler bereits seit längerem unterstützt, unter anderem auch für alle C-Sprachen (C, C++ und auch OpenCL), weswegen für die Implementierung dieser Erweiterung kein extra Aufwand nötig ist [LLV17a, Abschnitt Statement Attributes]. Ebenso wird eine weitere eher kleine Erweiterung von ARM unterstützt: `cl_arm_get_core_id` [EC13] erweitert die OpenCL-Standardbibliothek um eine Funktion, die eine eindeutige ID für die aktuelle OpenCL Compute Unit (eine Unterteilung eines OpenCL-Geräts) zurückgibt, um so z. B. auf innerhalb einer Work-Group gemeinsamen temporären Speicherbereiche zugreifen zu können. Da in der VC4CL-Implementierung die VideoCore IV GPU genau eine Compute Unit besitzt, ist die Funktion `uint arm_get_core_id(void)` so implementiert, das sie immer Null als Ergebnis liefert.

4. OpenCL-Runtime

Die OpenCL-Runtime ist die hostseitige Laufzeitbibliothek, die von Anwendungen und Bibliotheken angesprochen werden muss, um OpenCL verwenden zu können. Der OpenCL-Standard spezifiziert die Signaturen sämtlicher Funktionen der Laufzeitbibliothek sowie das erwartete Verhalten im Erfolgs- oder Fehlerfall. Die Typen und Funktionen der Bibliothek sind in der Programmiersprache C definiert. Die OpenCL-Laufzeitbibliothek ist in den Kapiteln 4 und 5 des OpenCL 1.2-Standards definiert.

4.1 Standard

Der OpenCL 1.2-Standard teilt die Laufzeitbibliothek in zwei Kapitel auf. Der erste Teil der Laufzeitbibliothek, der in der Spezifikation in Kapitel 4 „The OpenCL Platform Layer" definiert ist, beschreibt „plattformspezifische Features, die es Anwendungen erlauben, OpenCL-Geräte und deren Konfigurationen abzufragen [...]" [Gro12, Kapitel 4]. Damit eine Anwendung mit OpenCL arbeiten kann (d. h. OpenCL-Kernel auf dafür vorgesehenen Geräten auszuführen kann), muss diese erst einmal Zugriff auf die verfügbaren OpenCL-Implementierungen bekommen. Wie bereits in Abschnitt 1.2.1 beschrieben, wird eine OpenCL-Implementierung durch eine **OpenCL-Plattform** repräsentiert. Die Plattform beschreibt, welche „Instanz" der (unter Umständen mehreren) verfügbaren OpenCL-Implementierungen verwendet wird. So kann z. B. auf einem Raspberry Pi eine VC4CL OpenCL-Implementierung neben einer pocl-Implementierung ([pd10]) vorhanden sein. Da alle OpenCL-Implementierungen die gleiche API besitzen, muss in diesem Fall (dass es zur Laufzeit mehrere verfügbaren Plattformen gibt), eine besondere Erweiterung (Khronos ICD Loader, siehe Abschnitt 4.3.1) verwendet werden, die diese Plattformen dynamisch lädt. Die OpenCL-Plattform gibt an, welche Implementierung verwendet wird und welche Version von OpenCL (z. B. OpenCL 1.2), welches Profil (volles Profil oder eingebettetes Profil) und Erweiterungen diese unterstützt (siehe Abschnitt 1.2.4), sowie den Namen und die Version der verwendeten Implementierung. Eine OpenCL-Plattform kann ein oder mehrere **OpenCL-Geräte** besitzen. Ein OpenCL-Gerät ist eine Repräsentation einer Hardware, die OpenCL-Kernel ausführen kann. Wie auch für die übergeordnete Plattform können für ein Gerät allgemeine Informationen abgefragt werden. Darunter fallen der Name, unterstützte Erweiterungen (OpenCL erlaubt es den Geräten einer Plattform, unterschiedliche Erweiterungen zu unter-

stützen), der Typ (z. B. CPU, GPU oder FPGA) und der Hersteller der Hardware. Ebenso können eine Vielzahl für die Ausführung (und vor allem die Optimierung) von OpenCL-Kernel für das jeweilige Gerät wichtige Werte abgefragt werden, wie die Anzahl der Prozessoren, die Größe des vorhandenen Gerätespeichers, die native Vektorbreite und viele mehr. Die gesamte Liste der verfügbaren Abfragen kann im OpenCL 1.2-Standard in der Tabelle 4.3 eingesehen werden. Neben den OpenCL-Geräten besitzt eine OpenCL-Plattform auch noch mehrere **OpenCL-Kontexte**, die jedoch vom Anwender selbst erstellt werden müssen. Ein OpenCL-Kontext dient der Verwaltung von weiteren Objekten, wie Befehlswarteschlangen, Buffer und Programme und kann ein oder mehreren Geräten zugeordnet sein (siehe Abbildung 1.1). [Gro12, Kapitel 4]

Das zweite Kapitel „The OpenCL Runtime" beschreibt die weiteren Objekte und Funktionen, die benötigt werden, um Kernel zu kompilieren, deren Parameter zu setzen und zu starten [Gro12, Kapitel 5]. Ebenso wie alle Objekte dieses Kapitels einem OpenCL-Kontext zugewiesen sind (siehe Abbildung 1.1), werden alle Befehle für ein OpenCL-Gerät einer **Befehlswarteschlange** zugeordnet. Diese sorgt für die Synchronisierung der Befehle, die durch **Event**-Objekte dargestellt werden und verwaltet deren Ausführung auf dem OpenCL-Gerät. Dies ist nötig, da OpenCL es Anwendungen erlaubt, über mehrere Host-Threads aus auf die OpenCL-Implementierung zuzugreifen. **Buffer** stellen Speicherbereiche dar, die zwischen dem Hostspeicher und dem Gerätespeicher kopiert werden können und werden benötigt, um dem Kernel Speicherbereiche für die Parameter bereit zu stellen. Die OpenCL-API bietet eine Vielzahl an Operationen, die auf Buffern ausgeführt werden können. Darunter fallen das Kopieren von und in einen Speicherbereich der Hostanwendung, das Kopieren in einen anderen Buffer sowie das Befüllen mit festen Werten. Um hostseitig die **Bilder** verwalten zu können, die von den Kernelinstanzen ausgelesen oder beschrieben werden können (siehe auch Abschnitt 3.1.1), bietet die Laufzeitbibliothek auch hierfür Funktionen, um Bilder zu erstellen, zu beschreiben und auszulesen. Ebenso ist eine Funktion zum Abfragen der unterstützten Bildformate definiert. Wie auch für alle anderen OpenCL-Objekte (z. B. Plattform, Kontext, Programm) können über weitere Funktionen Informationen über Bilder (wie die Maße, das Farbschema, usw.) und Buffer (wie die Größe, eventuelle damit verbundenen Host-Speicherbereiche und die Lese- und Schreibflags) abgefragt werden. Ein **OpenCL-Programm** repräsentiert den Quellcode und/oder den kompilierten Binärcode eines oder mehreren Kernels. OpenCL unterstützt die Erstellung von Programm-Objekten aus OpenCL C-Quellcode (siehe Abschnitt 1.2.2), aus einer beliebigen von der OpenCL-Implementierung unterstützten Zwischensprache (z. B. SPIR-V, siehe Abschnitte 2.1.3 und 4.3) und aus gerätespezifischen Binärcode heraus. Programme, die aus Quellcode oder einer Zwischensprache erstellt wurden, können über spezielle Funktionen kompiliert und verlinkt werden. Für die Kompilierung unterstützt OpenCL zusätzlicher Parameter, die in den Abschnitten 5.6.4 und 5.6.5

der OpenCL 1.2-Spezifikation eingesehen werden können. Um OpenCL-Kernel ausführen zu können, muss aus einem Programm-Objekt ein **OpenCL-Kernel**-Objekt extrahiert werden, dem daraufhin die Parameter für die Ausführung gesetzt werden können. Wie auch für Programme bietet die Laufzeitbibliothek für Kernel eine Funktion, um Informationen (wie den Namen, den Binärcode, die zur Kompilierzeit gesetzten Attribute und die Parameter) abzufragen. Ist ein Kernel-Objekt erstellt, wird es einer Befehlswarteschlange zugeordnet, die globalen und lokalen Dimensionen für die Work-Items und Work-Groups (siehe Abschnitt 3.1.1) gesetzt und dann für die Ausführung eingereiht. Weiterhin werden noch Funktionen geboten, um Informationen über die eingereihten Befehle (wie die Ausführung eines Kernels oder das Kopieren eines Buffers in den Gerätespeicher) abzufragen und um auf bestimmte Befehle zu warten. [Gro12, Kapitel 5]

Die OpenCL-Laufzeitbibliothek besitzt eine C-Schnittstelle und somit keine automatische Speicherverwaltung. Daher bietet sie eine manuelle Speicherverwaltung, indem für jeden Typen (Plattform, Kernel, Buffer, usw.) ein Paar von Funktionen bereitgestellt werden, um einen Referenzzähler für ein Objekt zu erhöhen oder zu verringern. Besitzt ein Objekt keine Referenzen mehr (also der Referenzzähler ist null), kann es von der OpenCL-Implementierung entfernt werden. Jedoch muss ein Anwendungsentwickler explizit diese Funktionen aufrufen, um Objekte am Leben zu erhalten oder freizugeben.

4.2 Umsetzung

Die OpenCL-Laufzeitbibliothek ist im Projekt **VC4CL** implementiert. Aufgrund der einfacheren Programmierung und der Verfügbarkeit von Objekt-orientierten Konzepten ist VC4CL in C++ geschrieben. Somit werden die in der OpenCL-Laufzeit spezifizierten Objekte durch „echte" C++-Objekte repräsentiert (z. B. der Klassen `Buffer` oder `Kernel`), anstatt der im Standard definierten Zeiger. Dafür muss jedoch eine Schnittstelle implementiert werden, die die Funktionen der Laufzeitbibliothek auf Methodenaufrufe der internen Objekte, sowie zwischen den verschiedenen Repräsentationen der OpenCL-Objekte umwandelt. Dafür sind grundsätzlich alle Funktionen der OpenCL-Laufzeitbibliothek gleich aufgebaut: Als Erstes wird überprüft, ob die übergebenen OpenCL-Objekte gültig sind (d. h. ob sich hinter dem Zeiger ein gültiges VC4CL-Objekt befindet, siehe Abschnitt 4.3.1 für Details). Danach wird die jeweilige Instanzmethode des Objektes aufgerufen. In den folgenden Abschnitten wird ausschnittsweise die Umsetzung der Implementierung dargestellt, indem für eine Auswahl an Features der OpenCL-Laufzeitbibliothek aufgezeigt wird, wie diese implementiert sind.

4.2.1 Buffer

Der OpenCL 1.2-Standard definiert OpenCL-Bufferobjekte als eindimensionale Speicherbereiche, die initial auf einem beliebigen Speicher liegen (z. B. Hauptspeicher oder Grafikspeicher) und bei Verwendung auf den Speicher des verwendenden OpenCL-Geräts migriert werden. Die Migration kann unter anderem durch das Kopieren des Buffers oder durch das Abbilden in den Adressbereich des jeweiligen Prozessors geschehen. Weiterhin können Buffer über zusätzliche Laufzeitfunktionen gelesen und beschrieben werden, indem jeweils ein Speicherbereich mitsamt Größe angegeben wird, in den der Bufferinhalt hineinkopiert oder dessen Inhalt in den Buffer geschrieben wird. Diese Funktionen sind nötig, da abhängig von der Hardware der OpenCL-Implementierung ein Buffer, der auf dem Grafikspeicher liegt, nicht vom Hauptspeicher aus direkt zugänglich ist und nur die OpenCL-Implementierung auf die Daten des Buffers zugreifen kann. Weiterhin lassen sich Buffer einmal unterteilen (d. h. ein Teilbuffer kann nicht nochmal unterteilt werden), wobei die Teilbuffer eine rein konzeptionelle Aufgabe besitzen, um z. B. auf mehrere Bufferbereiche gleichzeitig Lesen und Schreiben zu können. Das Datenlayout und die Operationen auf einen Buffer werden durch Teilbuffer nicht verändert, jedoch können Teilbuffer als Parameter der Laufzeitfunktionen anstelle „echten" Buffern verwendet werden, um eine Operation nur auf dem festgesetzten Teil (Startoffset und Länge) auszuführen. [Gro12, Abschnitt 5.2]

Die Umsetzung der OpenCL-Bufferobjekte in der VC4CL OpenCL-Implementierung ist verhältnismäßig einfach aufgebaut. Die VideoCore IV GPU besitzt keinen eigenen Hardwarespeicher (siehe Abschnitt 1.3.3). Ebenso wird beim Reservieren eines Speicherblocks im Grafik-Speicherbereich des Hauptspeichers, dieser automatisch in den virtuellen Speicher der Host-Anwendung abgebildet. Dadurch fällt zum einen das Migrieren des Buffers zwischen Hauptspeicher und Grafikspeicher weg, zum anderen können Lese- und Schreibzugriffe aus der Host-Anwendung heraus durch die Standard C-Funktion `memcpy` implementiert werden. Zur Allokation eines Speicherbereichs im Grafikspeicher wird über die Mailbox-Schnittstelle (siehe Abschnitt 4.2.4) ein Handle beantragt, das einen Speicherbereich mit der gewünschten Größe und dem gewünschten Alignment repräsentiert. Das Handle wird durch einen weiteren Mailbox-Befehl auf eine Speicheradresse im virtuellen Speicherbereich der VideoCore IV GPU festgelegt. Diese Adresse wird von der GPU verwendet, um auf den Speicherbereich zugreifen zu können. Durch ein weiteres Memory Mapping (Abbilden von Speicherbereichen in einen anderen virtuellen Adressbereich) wird die Adresse in eine Speicheradresse des Adressbereichs der Host-Anwendung abgebildet. Diese wird hostseitig zum Beschreiben und Auslesen des Buffers verwendet, deutet jedoch auf den gleichen Speicherblock. Bei der Freigabe des Buffers werden die drei Schritte in umgekehrter Reihenfolge ausgeführt: das Mapping in den Host-Adressbereich wird aufgehoben, die Speicheradresse im GPU-Adressbereich

wird aufgelöst und am Schluss das Handle wieder freigegeben. Da auf der GPU angelegte Speicherbereiche – im Gegensatz zu im Hostspeicher reservierter Speicher – bei Programmende nicht automatisch wieder freigegeben werden – die GPU bekommt nicht mit, wenn ein Programm beendet wird – muss bei Buffern besonders dafür gesorgt werden, dass diese ordnungsgemäß wieder freigegeben werden. Dafür wird das C++-Programmieridiom RAII (Resource Acquisition is Initialization) verwendet, das dafür gedacht ist, Ressourcen in Objekte zu Kapseln, die in ihrem Destruktor die jeweilige Ressource automatisch und garantiert wieder freigeben. Die RAII-Klasse zum Verwalten der Adressen und des Handles für Speicherbereiche im Grafikspeicher in der VC4CL-Implementierung heißt `DeviceBuffer` und wird mit der Konstruktion eines Buffers angelegt und mit dessen Freigabe auch freigegeben. Teilbuffer teilen sich den Speicherbereich mit ihrem Vater-Buffer und somit auch das `DeviceBuffer`-Objekt.

4.2.2 Kompilierung

Eine der Hauptziele von OpenCL ist es, eine plattformunabhängige Schnittstelle bereit zu stellen, die gleich angesprochen werden kann, unabhängig von den Besonderheiten der darunter liegenden Architektur (z. B. durch das automatische Migrieren von Buffern, siehe Abschnitt 4.2.1). Deshalb muss auch der Code zur Ausführung von OpenCL-Kerneln in einer plattformunabhängiger Weise verbreitet werden können. Hierfür bestimmt die OpenCL-Spezifikation, dass OpenCL-Implementierungen einen Compiler beinhalten müssen, der als Eingangssprache mindestens OpenCL C-Quellcode unterstützt, und diesen in hardwarespezifische Maschineninstruktionen umwandelt. Dieser Compiler ist für die VC4CL-Implementierung in Kapitel 2 beschrieben. Wie bereits in Abschnitt 4.1 erläutert, können OpenCL-Programme mit Quellcode einer von der Implementierung unterstützten Quellsprache (z. B. OpenCL C, SPIR-V) oder direkt mit dem Binärcode von bereits für die jeweilige Hardware kompilierten Kerneln erzeugt werden. Wird eine OpenCL-Programm-Objekt mit Quellcode erstellt, muss dieses über die Laufzeitfunktionen `clBuildProgram` oder `clCompileProgram` und `clLinkProgram` kompiliert und verlinkt werden. `clBuildProgram` ruft die Funktionen `clCompileProgram` und `clLinkProgram` hintereinander auf und ist deshalb nur eine Vereinfachung der Kompilierung von OpenCL-Programmen mit nur einer Quelldatei. Ein Aufruf von `clBuildProgram` ruft den VC4CC-Compiler aus Kapitel 2 auf, übergibt die zusätzlichen Optionen an den jeweiligen Präcompiler (siehe Abschnitt 2.1.1) und speichert das Ergebnis sowie den Kompilierungslog in das Programm-Objekt. Die VC4CL-Implementierung unterstützt jedoch derzeit noch keine Verlinkung mehrerer Quellcodedateien. Stattdessen wird der Verlinkungsschritt dazu verwendet, um aus dem vom VC4CC-Compiler erzeugten Binärcode die Metadaten der Kernel (z. B. Offset vom Start des Binärcodes, Länge in Wörtern, Anzahl und Art der Parameter)

zu extrahieren. Anhand dieser Metadaten werden die im Programm die enthaltenen Kernel gefunden und gegenseitig abgetrennt, sowie der Block der globalen Daten definiert. Die Erstellung des Metadaten-Blocks wird in Abschnitt 2.3.4 beschrieben. Werden beim Aufruf von `clLinkProgram` mehrere Quellprogramme angegeben, wird ein Fehlerstatus zurückgegeben, dass das Verlinken nicht möglich ist.

4.2.3 Command Queue

In OpenCL werden die meisten Befehle nicht direkt beim Funktionsaufruf ausgeführt, sondern in eine Befehlswarteschlange eingereiht, die diese Befehle abarbeitet. Ausgenommen hiervon – also Funktionen, die direkt ausgeführt werden – sind die Funktionen der Laufzeitbibliothek zum Erstellen und Freigeben von OpenCL-Objekten und zum Abfragen derer Eigenschaften. Die in eine Befehlswarteschlange eingereihten Befehle werden durch Event-Objekte repräsentiert, mit denen z. B. der aktuelle Status der Ausführung verfolgt werden kann. Der OpenCL 1.2-Standard erlaubt Befehlswarteschlangen, sowohl asynchron, als auch – wenn die Anwendung einen speziellen Parameter beim Erstellen der Warteschlange angibt – in einer anderen Reihenfolge zu arbeiten [Gro12, Abschnitt 5.1]. In der VC4CL-Implementierung werden alle in Befehlswarteschlangen eingereihte Befehle in einem separaten Thread ausgeführt. Grund dafür ist unter anderen, dass die Ausführung eines Kernels auf der VideoCore IV GPU blockiert, bis diese abgeschlossen ist (siehe Abschnitt 4.2.4), was bereits dazu führt, dass Kernel nicht asynchron gestartet werden können. Andererseits erlaubt die OpenCL 1.2-Spezifikation die Verwendung der OpenCL-Laufzeitbibliotheken aus mehreren Host-Threads heraus [Gro12, Anhang A.2]. Somit werden, unabhängig von der verwendeten OpenCL-Befehlswarteschlange, alle Befehle in eine globale Warteschlange eingereiht. Ein separater Thread wartet auf neue Einträge in der Warteschlange und arbeitet diese dann – je nach Typ – ab. Der Code zur Abarbeitung der Befehle befindet sich in der Datei `src/queue_handler.cpp` des VC4CL-Projekts. Der Einfachheit halber werden die Befehle in der Reihenfolge abgearbeitet, in der sie zur Warteschlange hinzugefügt wurden. Dadurch vereinfacht sich auch die Handhabung von Abhängigkeiten zwischen Befehlen. OpenCL erlaubt es in den meisten Funktionen der Laufzeitbibliothek, bei denen ein Befehl an eine Befehlswarteschlange angereiht wird, eine optionale Liste von Events anzugeben, von denen der neu erstellte Befehl abhängig ist, d. h. die fertiggestellt werden müssen, bevor der neue Befehl ausgeführt wird. So kann eine Anwendung z. B. die Ausführung eines Kernels in Auftrag geben, die von der vorher gestarteten Initialisierung der vom Kernel verwendeten Buffer abhängig ist. Da in dieser Implementierung alle Befehle in der Reihenfolge der Erstellung ausgeführt werden, müssen die Abhängigkeiten zwischen Befehlen und anderen, vorher erstellten, Befehlen nicht weiter beachtet werden.

4.2.4 Kommunikation mit GPU

Wie bereits erwähnt, findet die Kommunikation mit der VideoCore IV-Hardware über eine Mailbox-Schnittstelle statt. Eine Mailbox-Schnittstelle bietet Anwendungen nachrichtenbasierten Zugriff auf eine bestimmte Hardware, d. h. Befehle an diese Hardware werden in einem fest definierten Format über einen Systemaufruf (einen syscall via `ioctl`) an ein Kernel-Modul, also einen Teil des Linux Kernels gesendet, das diese Befehle dann dekodiert und in Befehle für die verwendete Hardware umwandelt. Somit kann eine Anwendung ohne besondere Rechte die darunterliegende Hardware steuern, wobei das Kernel-Modul (der Treiber) den Zugriff auf die Hardware kontrollieren kann. Das Raspbian Betriebssystem, die speziell für die Raspberry Pi-Modelle erstellte Debian-Distribution, implementiert eine solche Mailbox-Schnittstelle, um sämtliche Funktionen der VideoCore IV GPU zu steuern und abzufragen. Über diese Schnittstelle wird unter anderem Grafikspeicher reserviert und wieder freigegeben (siehe Abschnitt 4.2.1), die Ausführung von OpenCL-Kernels gestartet und Hardwareeigenschaften wie der vorhandene Host- und Grafikspeicher oder die aktuelle Taktrate der GPU abgefragt. Da bei Raspberry Pi-Modellen die Firmware der VideoCore IV GPU (genauer: die VPU, siehe Abschnitt 1.3) nicht nur für Grafikberechnungen, sondern auch für die Steuerung weiterer Peripherie und auch den Bootvorgang (z. B. das Laden des Bootloaders und das Hochfahren der Host-CPU) zuständig ist, können über diese Mailbox-Schnittstelle weitere Einstellungen wie die Taktgeber oder die Power-States der verschiedenen Komponenten (z. B. CPU, GPU, UART, PWM und SDRAM) gesetzt und ausgelesen werden [Fou17a].

Eine Anfrage über die Mailbox-Schnittstelle blockiert solange, bis das Ergebnis zurückgegeben wird, d. h. das Starten eines Kernels blockiert den aufrufenden Thread, bis die Ausführung beendet ist und ein Status-Wert zurückgegeben werden kann. Intern wartet das Linux Kernel-Modul solange blockierend, bis alle QPUs, auf denen ein Programm ausgeführt wird, den jeweiligen Hardware-Interrupt gefeuert haben (siehe Abschnitt 2.3.1). Dies ist auch ein Grund, warum alle Zugriffe auf die Mailbox in einem eigenen Thread laufen, um die Anwendungssthreads nicht zu behindern (siehe Abschnitt 4.2.3). Intern greift das Kernel-Modul auf der anderen Seite eines Mailbox-Aufrufs auf bestimmte Peripherie-Register zu, um die Grafikhardware zu steuern. Z.B. wird zum Ausführen eines Programms auf einer QPU die Startadresse der Uniforms (Parameter) in ein Register und die Startadresse des Codeblocks in ein weiteres Register geschrieben, woraufhin die VPU dieses Programm in eine interne Warteschlange einreiht [Bro13, Tabellen 65 und 66].

Da eine Anwendung, die die VC4CL OpenCL-Implementierung nutzen will, sowieso mit root-Rechten gestartet werden muss (siehe Abschnitt 1.3.3), bietet es sich an, direkt aus er VC4CL-Implementierung heraus auf diese Peripherieregister zuzugreifen, um sich so den zusätzlichen Overhead eines Systemcalls und der kernelseitigen

Bearbeitung zu sparen. Jedoch hat der Aufruf über die Mailbox-Schnittstelle den Vorteil, dass das Kernel-Modul alle Aufrufe systemweit synchronisiert, also es keine Race Conditions beim Auslesen oder Schreiben der Werte geben kann, nicht einmal, wenn eine andere Anwendung, z. B. eine OpenGL-Anwendung, die die VideoCore IV GPU für grafische Berechnungen verwendet, gleichzeitig Code auf der GPU ausführen will. Um eine möglichst gute Performance zu erreichen und trotzdem die gemeinsame Verwendung der VC4CL-Implementierung mit anderen Bibliotheken, die auf die GPU zugreifen, zu ermöglichen, kann ein OpenCL-Kernel sowohl über einen sicheren Mailbox-Aufruf als auch über die schnellere Variante gestartet werden, die direkt in die Peripherieregister schreibt. Welche der beiden Methoden zum Ausführen eines Kernels verwendet wird, wird beim Kompilieren der VC4CL-Laufzeitbibliothek festgelegt. Der tatsächliche Performanceunterschied der beiden Methoden kann in Abschnitt 5.3.2 eingesehen werden. Die Peripherieregister werden zusätzlich für andere rein informative Abfragen genutzt, z. B. für das Auslesen der Anzahl an QPUs, der Größe der VPM, der Hardwareversion und weiterer statistischer Werte. [Bro13, Abschnitt 10].

Für die Ausführung eines OpenCL-Kernels wird ein Speicherbereich im Grafikspeicher angelegt und der Maschinencode der Kernelfunktion und die globalen Daten hineinkopiert. Daraufhin werden die „versteckten" Parameter berechnet, d. h. die lokalen und globalen IDs und Größen werden aus der Anzahl der Work-Groups und Work-Items innerhalb einer Work-Group berechnet und die Startadresse der globalen Daten wird gesetzt. Die Berechnung der „versteckten" Parameter geschieht für jede Ausführung neu, da diese sich verändern, wohingegen die expliziten Parameter (der Kernel-Funktion) konstant bleiben. Gestartet wird über die gewählte Methode (Systemaufruf über die Mailbox-Schnittstelle oder über das Schreiben der Peripherieregister) immer eine ganze Work-Group, d. h. es werden so viele QPUs angesprochen, wie benötigt werden, um alle Work-Items einer Work-Group gleichzeitig auszuführen. Dies limitiert auch die maximale unterstützte Größe einer Work-Group auf die Anzahl der verbauten QPUs (zwölf bei allen Raspberry Pi-Modellen, siehe Abschnitt 1.3.2). Weitere Work-Groups werden hintereinander ausgeführt, jeweils wieder mit dem gleichen Code- und Datenblöcken und abgeänderten „versteckten" Parametern.

4.2.5 Nicht unterstützte Funktionen

Auch in der hostseitigen Implementierung der OpenCL-Laufzeitbibliotheken gibt es Funktionen des OpenCL 1.2-Standards, die derzeit (noch) nicht unterstützt werden. Darunter zählt die Unterteilung von OpenCL-Geräten. Der OpenCL 1.2-Standard spezifiziert, dass OpenCL-Geräte in mehrere Kind-Geräte unterteilt werden können [Gro12, Abschnitt 4.3]. Diese Unterteilung kann bei leistungsstärkeren Prozessoren sinnvoll sein, bei denen auch die Hardware weiter unterteilt ist, z. B. in Blöcke von

Kernen, die sich einen Cache teilen. Da sich nicht alle OpenCL-Geräte sinnvoll unterteilen lassen, ist dieses Feature auch optional, d. h. eine Implementierung darf alle definierten Unterteilungsmodi von Geräten („in X gleich große Kind-Geräte", „in Kind-Geräte, die jeweils X Elemente haben" oder „in Kind-Geräte, die sich eine Cache-Ebene teilen") als nicht unterstützt ablehnen. Die VideoCore IV-Architektur ließe sich zwar z. B. in die verschiedenen Schichten (siehe Abschnitt 1.3) unterteilen, was jedoch aufgrund der sehr geringen Anzahl von zwölf Prozessoren und der Eigenschaft, dass sich selbst die Schichten den Speicherzugriff teilen, keinen Nutzen erbringt.

Wie bereits in Abschnitt 4.2.2 erläutert, wird noch kein Verlinken mehrerer Quelldateien unterstützt. Jedoch ist derzeit (Stand: 05.10.2017) für die SPIR-V Tools-Bibliothek [Gro17c], die im SPIR-V-Frontend verwendet wird (siehe Abschnitt 2.1.3), ein Linker in Entwicklung, der nach Fertigstellung möglicherweise verwendet werden kann, um zumindest über das SPIR-V-Frontend kompilierte Programme verlinken zu können. Falls das Verlinken unterstützt werden soll, entweder durch die SPIR-V Tools-Bibliothek, oder durch einen eigenen Linker, muss das Verhalten der Laufzeitfunktionen für das Kompilieren und das Verlinken geändert werden. Da der VC4CC-Compiler in jedem Fall Maschinencode generiert, kann das Ergebnis nicht mehr verlinkt werden. Stattdessen könnte z. B. die Vorkompilierung von OpenCL in LLVM IR oder SPIR-V im „Kompilierungsschritt" und die weitere Kompilierung (und Verlinkung) des Zwischencodes mithilfe des VC4CC-Compilers im „Verlinkungsschritt" ausgeführt werden.

Wie auch aufseiten der GPU werden die hostseitigen Funktionen zum Erstellen, Auslesen, Beschreiben und Löschen von Bildern [Gro12, Abschnitt 5.3] derzeit nicht unterstützt. Auch hier sind für manche der Funktionalitäten der optionalen Komponente der Bildbearbeitung bereits experimentelle Implementierungen vorhanden. Diese sind aber auch ungetestet und müssen über eine Compileroption explizit eingeschaltet werden.

4.3 Implementierte Erweiterungen

Für den OpenCL 1.2-Standard gibt es auch Erweiterungen, die hostseitig neue Funktionen bereitstellen. Wie auch bei den kernelseitigen Erweiterungen werden eine Auswahl der in der OpenCL-Registry [Gro17a] verfügbaren Host-Erweiterungen implementiert, die entweder sich mit der VideoCore IV-Hardware leicht umsetzen lassen oder die einen größeren Nutzen bieten. Eine eher kleinere Erweiterung ist `cl_altera_device_temperature` von Altera, einer Tochter von Intel, die die Funktion `clGetDeviceInfo` der Laufzeitbibliothek zum Holen verschiedener Informationen, wie dem Gerätenamen, den unterstützten Erweiterungen und Datentypen, um eine Abfrage erweitert, die die aktuelle Gerätetemperatur in Grad

Celsius zurückgibt [KN14]. Wie bereits in Abschnitt 4.2.4 beschrieben, kann über die Mailbox die Temperatur des Grafikchips ausgelesen werden. Dieser Wert wird in Grad Celsius umgerechnet und bei Aufruf der `clGetDeviceInfo` Funktion mit dem neu eingeführten Parameter ausgegeben.

Eine umfangreichere Erweiterung ist die Unterstützung von sog. Shared Virtual Memory (SVM). Shared Virtual Memory, eine Terminologie, die im OpenCL 2.0-Standard eingeführt wird, bedeutet, das sich der Host und OpenCL-Geräte den (virtuellen) Speicher teilen, d. h. das beide Prozessoren auf den Speicher zugreifen können, ohne das dieser vom Hauptspeicher in den Gerätespeicher (oder umgekehrt) kopiert werden muss. Dies ist vor allem bei Grafikprozessoren von Vorteil, die sich den Speicher sowieso mit der CPU teilen, wie den Raspberry Pi-Modellen. Da jedoch somit CPU und GPU gleichzeitig auf den gleichen Speicherbereich zugreifen können, muss die Anwendung dafür sorgen, dass schreibender Zugriff synchronisiert wird, indem z. B. der Host nur auf einen SVM-Speicherbereich schreibt, wenn gerade kein Kernel ausgeführt wird, der diesen Speicherbereich verwendet. Aus anderen bereits in Abschnitt 1.2.4 erwähnten Gründen kann der OpenCL 2.0-Standard für die VideoCore IV-Architektur nicht implementiert werden. Deshalb wird die Erweiterung `cl_arm_shared_virtual_memory` von ARM implementiert, die die Funktionen zur Unterstützung von SVM für ältere OpenCL-Versionen verfügbar macht [EP16]. Die Implementierung der Erweiterung ist an die Implementierung der Standard-Buffer angelehnt: In beiden Fällen wird bei der Erstellung eines (SVM-) Buffers ein genügend großer Speicherbereich aus dem GPU Speicher reserviert (siehe Abschnitte 4.2.1 und 4.2.4) und in einem `DeviceBuffer`-Objekt verwaltet, um den Speicherbereich auch wieder freigeben zu können. Jedoch kann bei der Erstellung von SVM-Speicherbereichen direkt ein Zeiger zurückgegeben werden, der sich im Adressbereich des Hosts befindet, aber auf den über die Grafikhardware reservierten Speicher zeigt. Der OpenCL 2.0-Standard (und somit auch die ARM Erweiterung) definiert verschiedene Stufen für die Unterstützung von SVM. So kann SVM einerseits soweit unterstützt werden, dass die Host-CPU auf Speicher, der mit `clSVMAlloc`, also der geräteseitigen Allokation, zugreifen kann. Andererseits kann auch, falls es die OpenCL-Implementierung unterstützt, der geräteseitiger Kernelcode auf Speicher zugreifen, der durch die „normalen" Speicherallokationsmethoden des Hosts reserviert wurde (z. B. `malloc`). Da über die Mailbox reservierter Speicher automatisch im Speicherbereich der Host-CPU liegt, jedoch die GPU nur auf die GPU-seitig angelegten Speicherbereiche zugreifen kann (andere Speicherzugriffe werden von der VPM maskiert, also ignoriert [Bro13, Abschnitt QPU Reading and Writing of VPM]), wird nur die erste Variante unterstützt.

Weiterhin wird die Erweiterung `cl_khr_il_program` implementiert, die es erlaubt, die Zwischensprache SPIR-V als Quellcode zum Erstellen von OpenCL-Programmen zu verwenden. `cl_khr_il_program` ist offiziell zwar nur für die

OpenCL-Version 2.0 als Erweiterung definiert [Gro16, Abschnitt 9.22], da sie nicht für ältere OpenCL-Versionen definiert ist und ab OpenCL 2.1 Teil des Standards ist. VC4CC-Compiler unterstützt diese Erweiterung trotzdem, da sowieso SPIR-V-Code als Eingabe akzeptiert wird (siehe Abschnitt 2.1.3). Die Erweiterung bietet eine neue Funktion `clCreateProgramWithILKHR`, die die gleiche Funktionalität wie die Standard-Funktion zum Erstellen von Programmen aus Quellcode `clCreateProgramWithSource` bietet, sowie einen zusätzlichen Parameter für `clGetDeviceInfo` zum Abfragen, welche SPIR-V-Version der Compiler unterstützt. Unterstützt wird derzeit (Stand: 05.10.2017) die neuste SPIR-V-Version, 1.2 [Gro17b].

4.3.1 Khronos ICD Loader

Die umfangreichste implementierte OpenCL-Erweiterung ist der Khronos ICD Loader (`cl_khr_icd`) [CGH+10][Gro15, Abschnitt 9.18]. ICD Loader steht hierbei für Installable Client Driver Loader und bietet eine Schnittstelle, um OpenCL-Implementierungen dynamisch zu laden. Generell können in den Programmiersprachen C/C++ keine zwei Implementierungen für die gleiche Schnittstelle/Funktion in einem Programm bestehen, oder genauer gesagt, eine der Implementierungen würde von der anderen überdeckt werden. OpenCL ist jedoch so konzeptioniert (unter anderem durch die Plattform als Repräsentation einer Implementierung), dass eine Anwendung mit mehreren OpenCL-Implementierungen (z. B. für mehrere Grafikkarten, oder eine CPU- und eine GPU-Implementierung) arbeiten kann. Der ICD Loader bietet eine Lösung für dieses Problem, indem zur Laufzeit dynamisch entschieden wird, welche Implementierung für einen Funktionsaufruf verwendet wird. Dafür implementiert der ICD Loader die öffentliche Schnittstelle der OpenCL-Laufzeitbibliothek und leitet alle Anfragen an eine der verfügbaren Implementierungen weiter. Zu Beginn (bei der ersten Verwendung) lädt der ICD Loader auf eine plattformspezifische Weise alle verfügbaren OpenCL-Implementierungen (genauer gesagt: deren Bibliotheken, z. B. `*.so` oder `*.dll`-Dateien), um die Funktionsaufrufe delegieren zu können. Um jedoch vom ICD Loader angesprochen werden zu können, müssen OpenCL-Implementierungen einige Punkte beachten: Als Erstes dürfen die Funktionen nicht so heißen, wie im OpenCL 1.2-Standard vorgeschrieben, um nicht anstatt der ICD Loaders aufgerufen zu werden. Hierfür wird in der VC4CL-Implementierung, abhängig davon, ob diese für die Unterstützung des ICD Loaders oder für alleinstehende Verwendung kompiliert wird, der Präfix „VC4CL_" an alle Funktionen der OpenCL-Standardbibliothek angehängt. Weiterhin muss eine Funktion `clIcdGetPlatformIDsKHR` implementiert werden, die alle Plattformen dieser Implementierung zurückliefert. Diese Funktion darf keinen Präfix besitzen, muss also genauso heißen, wird aber von ICD Loader explizit innerhalb der für eine Implementierung geladene Bibliothek gesucht, wodurch sie eindeutig identifiziert

werden kann. Ebenso muss jedes OpenCL-Objekt ein zusätzliches Feld an erster Position (also ohne Offset zur Adresse des OpenCL-Objektes) besitzen, das einen Zeiger auf einen Dispatcher beinhaltet, der Zeiger auf alle OpenCL-Laufzeitfunktionen der jeweiligen Implementierung beinhaltet (siehe Listing 4.1). Der ICD Loader verwendet dann den Zeiger auf das Dispatch-Objekt sowie den fest definierten Index, um die Implementierung einer bestimmten Laufzeitfunktion aufzurufen (siehe Listing 4.2).

```
1 //defined by the OpenCL specification:
2 typedef struct _cl_context* cl_context;
3
4 //defined by the implementation:
5 struct _cl_context
6 {
7   struct _cl_icd_dispatch* dispatch = &vc4cl_dispatch;
8   /* [...] other members */
9 }
10
11 struct _cl_icd_dispatch vc4cl_dispatch = {
12   &VC4CL_clGetPlatformIDs,
13   &VC4CL_clGetPlatformInfo,
14   /* [...] all other pointers */
15 };
```

Listing 4.1: Aufbau der OpenCL Objekte für die Verwendung mit dem ICD Loader

```
1 cl_command_queue clCreateCommandQueue(cl_context context,
    cl_device_id device, cl_command_queue_properties
    properties, cl_int* errcode_ret)
2 {
3   return context->dispatch->clCreateCommandQueue(context,
    device, properties, errcode_ret);
4 }
```

Listing 4.2: Implementierung von `clCreateCommandQueue` im ICD Loader [CGH+10]

Entscheidend hierbei ist, dass der Zeiger auf das Dispatch-Objekt keinen Offset vom Zeiger auf das jeweilige OpenCL-Objekt besitzt, da der ICD Loader es an genau dieser Stelle erwartet. Dies ist jedoch für die C++-Objekte der VC4CL-Implementierung nicht gegeben, da diese virtuelle Methoden besitzen und somit am Anfang des Objektes die sich die `vtable` des Typs befindet. Um jedoch die Implementierung trotzdem mit dem ICD Loader kompatibel zu machen, wird ein Trick angewendet, der (genauso wie das Anhängen eines Präfixes an die Implementierungen der Laufzeitfunktionen) von der CPU-seitigen OpenCL-Implementierung pocl abgeschaut

wurde [pd10]. Listing 4.3 zeigt am Beispiel des Kontext-Typs, wie die Objekte aufgebaut sind: Alle internen Objekttypen erben von der Templateklasse `Object`, die ein Feld, vom jeweiligen OpenCL-Objekttyp definiert, der dem VC4CL-Objekttyp entspricht. Für `Context` ist das der Typ `_cl_context`. Dieser wiederum hat als erstes Feld den Zeiger auf das Dispatch-Objekt, der vom ICD Loader benötigt wird, sowie einen weiteren Zeiger, der im Konstruktor auf das interne C++-Objekt gerichtet wird. Dies ist nötig, damit die Implementierung sowohl vom OpenCL-Objekt (z. B. ein `cl_context`, der einen Zeiger auf `_cl_context` darstellt) auf das C++-Objekt (z. B. ein `Context`) als auch vom C++-Objekt auf das OpenCL-Objekt zugreifen kann. In Listing 4.4 wird ein stark vereinfachtes Beispiel der Umwandlung zwischen

```
1  struct _cl_context
2  {
3    struct _cl_icd_dispatch* dispatch = &vc4cl_dispatch;
4    void* object;
5    _cl_context(void* object) : object(object) { /* ... */ };
6  };
7
8  template< typename BaseType, ...>
9  class Object
10 {
11   /* [...] destructors, methods, ... */
12 protected:
13   BaseType base;
14   Object() : base(this) { };
15 }
16
17 class Context : public Object<_cl_context, ...>{ };
```

Listing 4.3: Aufbau eines C++ Objektes zur Unterstützung von ICD

OpenCL-Objekt und C++-Objekt anhand der bereits in Listing 4.2 gezeigten Laufzeitfunktion `clCreateCommandQueue` aufgezeigt. Hier wird über das `object` Zeiger-Feld des OpenCL-Objekts, das initial auf die Position des C++-Objektes gesetzt wird (siehe Listing 4.3) auf dieses zugegriffen und die im Standard für diese Funktion definierten Aktionen durchgeführt. Gibt die implementierte Funktion einen Zeiger auf ein OpenCL-Objekt zurück, so wird dieser erzeugt, indem der Zeiger auf das Feld `base` des C++-Objekts zurückgegeben wird. Diese Implementierung ist unter anderem möglich, da der OpenCL-Standard zwar die OpenCL-Objekttypen auf Zeiger von fest benannten Typen definiert (z. B. `cl_context` ist definiert als Zeiger auf `_cl_context`), den Aufbau dieser Typen jedoch der Implementierung überlässt. Ebenso ist es hier von Vorteil, dass eine OpenCL-Anwendung nie selbstständig den von OpenCL-Objekten belegten Speicher freigibt, sondern dies über die jeweiligen `clReleaseXXX` Funktionen veranlässt, die dann den Speicher auf

```
1 cl_command_queue clCreateCommandQueue(cl_context context,
    cl_device_id device, cl_command_queue_properties
    properties, cl_int* errcode_ret)
2 {
3   /* [...] check validity of all parameters according to the
      specification */
4   CommandQueue* queue = new CommandQueue(
5     (Context*) context->object,  /* [...] */);
6   return queue->base;
7 }
```

Listing 4.4: Implementierung von `clCreateCommandQueue` in VC4CL (Ausschnitt)

implementierungsspezifische Weise (z. B. die Freigabe des C++-Objektes anstatt des OpenCL-Objektes) freigeben kann.

Khronos bietet eine offizielle Implementierung des ICD Loaders für die gängigen Plattformen (Windows, Linux, Mac OS) an, die auf den meisten Linux-Distributionen in den offiziellen Paketquellen verfügbar ist, ebenso wie die OpenCL Header der Laufzeitbibliothek, die benötigt werden, um Bibliotheken mit OpenCL zu verlinken. So kann der ICD Loader z. B. auf der Raspbian Distribution mit dem Befehl `apt-get install ocl-icd` heruntergeladen und installiert werden.

5. Ergebnisse

In diesem Kapitel werden die Ziele dieser Arbeit (siehe Abschnitt 1.1) mit den Leistungen der Implementierungen verglichen, um zu sehen, wie gut diese Ziele umgesetzt werden. Ebenso wird auf noch ausstehende Probleme eingegangen, deren Lösungen entweder den Umfang dieser Arbeit übersteigen oder (vorerst) nicht umsetzbar sind (siehe Abschnitt 5.2.1).

5.1 OpenCL CTS

Die OpenCL Conformance Test Suite (CTS) ist eine Sammlung an Tests, die von der Khronos Group bereitgestellt wird. Damit OpenCL-Implementierungen als „OpenCL konform" gelten dürfen, müssen sie unter anderem diese Testsammlung bestehen. Khronos hat die OpenCL CTS am 16. Mai 2017 als Open-Source-Projekt für die Öffentlichkeit verfügbar gemacht, zusammen mit der Veröffentlichung der finalen Version von OpenCL 2.2 und der SPIR-V 1.2 Spezifikation [Gro17b]. Vor diesem Zeitpunkt konnte die OpenCL CTS nur über das OpenCL Adopters Program heruntergeladen werden, wofür eine Mitgliedschaft für OpenCL 1.2 mindestens 1.500 $ (USD) für akademische Institutionen oder 10.000 $ für andere Institutionen kostet (Stand: 06.10.2017) [Gro17e, Conformance > Adopters]. Durch die Veröffentlichung der OpenCL CTS können diese offiziellen Tests auch verwendet werden, um die Standardkonformität der VC4CL OpenCL-Implementierung zu überprüfen und zu verbessern.

Die Tests in der OpenCL CTS sind in Testprogramme unterteilt, die meist zusätzliche optionale Parameter nehmen, um z. B. nur einzelne Tests auszuführen. Ist für ein Testprogramm kein Parameter angegeben, werden alle in diesem Programm enthaltenen Tests durchgeführt. Ein Testprogramm stellt eine Kategorie von Tests dar. So können mit dem API-Testprogramm verschiedene allgemeine Funktionen der Host-API (Laufzeitbibliothek, siehe Abschnitt 4) getestet werden, wie das Abfragen der Plattform- und Geräteeigenschaften während das Compiler-Testprogramm die verschiedenen vorgeschriebenen Arten der Kompilierung und Parameter für den eingebauten Compiler und der Atomics-Test die korrekte Implementierung der atomaren Instruktionen auf dem OpenCL-Gerät überprüft (siehe Abschnitt 3.2.2). Ein zusätzliches Python-Skript kann alle vorhandenen Testfälle ausführen und für jeden Test in eine Logdatei dokumentieren, ob der Test erfolgreich ist und, falls nicht,

welcher Teil fehlgeschlagen ist. Ebenso werden Tests für optionale Features, wie die Unterstützung von Bildern oder 64-Bit Fließkommazahlen automatisch übersprungen, wenn die getesteten Features von der Implementierung nicht unterstützt werden.

Eine komplette Konformität ist in dieser Arbeit nicht angestrebt, da dies auch den Umfang erheblich sprengen würde. Jedoch kann in weiteren Entwicklungen (siehe Kapitel 6) ein größeres Augenmerk auf die Verbesserung der Konformität mit dem OpenCL-Standard geworfen werden. Einzelne Testprogramme aus der OpenCL CTS werden trotzdem verwendet, um z. B. einzelne Entwicklungen schnell testen zu können, falls die OpenCL CTS für genau dieses Feature einen Test bereitstellt. Ebenso werden Tests ausgeführt, um zu überprüfen, welches Verhalten der OpenCL-Standard z. B. bei bestimmten Fehler erwartet, falls dies nicht ausreichend im OpenCL-Standarddokument [Gro12] dokumentiert ist.

Die Kompatibilität der VC4CL-Implementierung mit der OpenCL CTS für OpenCL 1.2 wird stichprobenartig überprüft, um für einzelne Funktionen zu testen, ob die jeweilige Implementierung sich richtig verhält oder das richtige Ergebnis liefert. Ebenso wird die OpenCL CTS im Laufe der Entwicklung der VC4CL OpenCL-Implementierung verwendet, um zu überprüfen, ob Änderungen an der Implementierung sich immer noch richtig verhalten. Die Tabelle 5.1 zeigt den derzeitigen Stand der Kompatibilität mit der OpenCL CTS anhand der bestandenen Tests. Da die OpenCL CTS von einer Implementierung nicht unterstützte optionale Funktionalitäten (z. B. 64-Bit Arithmetik oder Unterstützung von Bildern) automatisch überspringt und als bestanden ausgibt, werden diese zu den bestandenen Testfällen hinzugezählt. Testprogramme, die komplett für den Test nicht implementierter Funktionen geschrieben sind, werden in der Tabelle 5.1 nicht aufgelistet.

Bei dem Testprogramm für die Kompilierungsoptionen und -direktiven ist anzumerken, dass vieler der Testfälle fehlschlagen, weil der VC4CC noch kein Verlinken mehrerer Quelldateien unterstützt (siehe Abschnitt 4.2.2). Die meisten anderen fehlgeschlagenen Testfälle basieren auf falschen Ergebnissen der VC4CL-Implementierung oder vereinzelt auch auf Abstürzen während der Testausführung.

Testprogramm	Testfälle	Bestanden	Anteil
Plattform- und Geräteinformationen	1	1	100%
Grundlegende Kernelfunktionalitäten	95	69	72%
Funktionen der Laufzeitbibliothek	72	60	83%
Kompilierungsoptionen und -direktiven	55	21	38%
Allgemeine OpenCL C Funktionen	17	8	47%
Geometrische OpenCL C Funktionen	8	1	12%
Relationale OpenCL C Funktionen	17	0	0
Threading-Dimensionen	12	0	0
Atomare OpenCL C Funktionen	13	4	30%
hostseitiges Profiling	31	31	100%
Event-Funktionen	28	24	85%
Speicherallokation	2	0	0
Bufferzugriff	93	73	78%
Ganzzahlarithmetik	94	2	2%

Tabelle 5.1: Bisher gesammelte Ergebnisse der Kompatibilität der VC4CL Implementierung mit der OpenCL CTS für OpenCL 1.2

5.2 Anwendungen

Ein Teil des Ziels dieser Arbeit ist es, die VC4CL-Implementierung soweit zum Laufen zu bringen, dass diese mit Bibliotheken, die OpenCL nutzen, verwendet werden kann (siehe Abschnitt 1.1). In diesem Abschnitt wird beschrieben, welchen grundsätzlichen Einschränkungen OpenCL-Bibliotheken unterliegen müssen, um mit der VC4CL-Implementierung nutzbar zu sein, sowie welche Bibliotheken aufgrund von anderen Problemen nicht lauffähig sind und welche Bibliotheken (eingeschränkt) mit VC4CL verwendet werden können.

Für den Funktionstest der VC4CL-Implementierung wurden OpenCL-basierte Bibliotheken nach folgendem Schema ausgewählt: Die Bibliotheken oder Programme dürfen maximal die OpenCL-Version 1.2 voraussetzen, um keine Funktionen zu erwarten, welche die VC4CL-Implementierung nicht bietet, müssen unter Linux lauffähig sein, aufgrund der verwendeten Linux-Distribution Raspbian, müssen Open Source sein, damit zum Testen und Debuggen der Quellcode sowie der Kernelcode eingesehen werden kann, und sie müssen mindestens ein ausführbares Programm besitzen (z. B. Testprogramme), mit deren Hilfe das Ergebnis von GPU-seitigen Berechnungen analysiert und verifiziert werden kann. Ebenso wurde versucht, Bibliotheken auszuwählen, die einen möglichst großen Anwendungsbereich besitzen, um den Nutzen einer OpenCL-Implementierung auf der VideoCore IV GPU besser aufzeigen zu können. Die Tabelle 5.2 listet alle Bibliotheken auf, deren Verwendbarkeit mit der VC4CL-Implementierung getestet wurde und die nicht bereits durch die

oben genannten Kriterien abgelehnt wurden. Ebenso wird gezeigt, ob und wie viele der in den jeweiligen Bibliotheken mitgelieferten Testfällen erfolgreich ausgeführt werden können sowie stichpunktartig die Gründe für die fehlgeschlagenen Testfälle.

5.2.1 Nicht unterstützte Anwendungen

Die Liste der (derzeit) von der VC4CL-Implementierung nicht unterstützten Anwendungen ist deutlich länger als die der unterstützten Anwendungen. Die Ursache dafür liegt nicht immer auf der Seite der VC4CL-Implementierung und lässt sich grob in eine zwei Kategorien aufteilen:

Bereits einige der getesteten Bibliotheken scheiden für die Verwendung mit der VC4CL OpenCL-Implementierung bereits grundsätzlich aus. Grund dafür ist, dass diese Bibliotheken nicht unterstützte Features oder Erweiterungen des OpenCL-Standards (wie Bilder oder 64-Bit Arithmetik) benötigen oder Voraussetzungen an die Hardware stellen, die die VideoCore IV der Raspberry Pi-Modelle nicht erfüllen kann. Zu diesen Voraussetzungen zählen unter anderem eine minimale Work-Group-Größe von mehr als zwölf Work-Items oder zu viel benötigter Grafikspeicher. Da die minimale Größe einer Work-Group (sowie auch der benötigte Gerätespeicher) meistens durch den Algorithmus oder dessen Implementierung bedingt ist, müsste eine Anpassung der Bibliotheken für die Verwendung mit dieser OpenCL-Implementierung aufseiten der Bibliotheks- und Anwendungsentwickler geschehen. In diese Kategorie fallen Bibliotheken wie die Physiksimulations-Engine Bullet, die Work-Groups mit mindestens 64 Work-Items benötigt und die Bibliothek für lineare Algebra ViennaCL, die eine Work-Group-Größe von 128 Work-Items fordert sowie CP2K, zum Simulieren von Moleküldynamik, welches die Unterstützung der 64-Bit Datentypen `double` und `long` voraussetzt. Generell hat sich bei dem Test einer Vielzahl an OpenCL-basierten Bibliotheken gezeigt, dass einige davon nicht mit einer (für Grafikprozessoren ungewöhnlich kleine) Work-Group-Größe von weniger als 128 umgehen können, da die Bibliotheken 128 oft als Mindestgröße festlegen, ohne das eigentliche Limit der OpenCL-Implementierung abzufragen, obwohl dies über die OpenCL-Laufzeitbibliothek möglich wäre. Weiterhin können manche Bibliotheken nicht mit Work-Item-Größen umgehen, die keine Zweierpotenz sind, was ebenso sehr unüblich, aber vom OpenCL-Standard erlaubt ist.

Die zweite Kategorie nicht unterstützter Bibliotheken ergibt sich durch noch nicht implementierte Features und fehlerhafte oder unvollständige Implementierung gewisser Features seitens der VC4CL Laufzeit- oder Standardbibliothek oder des VC4CC-Compilers. Für die meisten der Bibliotheken, die unter diese Kategorie fallen, scheitert die Unterstützung daran, das der VC4CC-Compiler die Kernel dieser Bibliotheken nicht vollständig kompilieren kann. Wie bereits in Abschnitt 2.3.2 beschrieben, unterstützt der derzeit verwendete Algorithmus für die Registerzu-

Bibliothek	Unterstützt	Grund	Quelle
BFGminer	Nein	Registerzuordnung	`http://github.com/luke-jr/bfgminer/`
boost compute	88/137	falsche Ergebnisse, benötigt `long`	`http://github.com/boostorg/compute`
Bullet Physics	Nein	Kompilierungsfehler, Work-Group-Größe	`http://github.com/bulletphysics/bullet3/`
CLBlast	Nein	Fehler in Testcode	`http://github.com/CNugteren/CLBlast`
clMathLibraries	Nein	Work-Group-Größe, Kompilierungsfehler	`http://github.com/clMathLibraries`
CLTune	Nein	Gerätespeicher	`http://github.com/CNugteren/CLTune`
CP2K	Nein	benötigt `long`	`http://www.cp2k.org/`
EasyCL	52/71	falsche Ergebnisse	`http://github.com/hughperkins/EasyCL`
gputools	Nein	benötigt Bilder	`http://github.com/maweigert/gputools`
hashcat	Nein	Work-Group-Größe	`http://hashcat.net/hashcat/`
John the Ripper	Nein	Registerzuordnung	`http://openwall.com/john/`
octopus	Nein	benötigt `double`	`http://octopus-code.org/wiki/Main_Page`
OpenCV 3.x	Nein	Kompilierungsfehler, Work-Group-Größe	`http://github.com/opencv/opencv`
rendergirl	Nein	Registerzuordnung	`http://github.com/henriquenj/rendergirl`
VexCL	5/37	Fehler in Testcode	`http://github.com/ddemidov/vexcl`
ViennaCL	0/19	Work-Group-Größe	`http://viennacl.sourceforge.net/`
Zeta	Nein	Work-Group-Größe	`http://github.com/smatovic/Zeta`

Tabelle 5.2: Für die Verwendung mit der VC4CL Implementierung getestete Bibliotheken

ordnung keine komplexeren Instruktionsstrukturen, da er keine Register in den Arbeitsspeicher auslagert und somit nur 68 lokale Werte gleichzeitig halten kann (64 aus den physikalischen Registern sowie vier Akkumulatoren, siehe Abschnitt 1.3.2). An dieser Einschränkung scheitert z. B. die Unterstützung der Passwortknacker-Bibliothek „John the Ripper", die überdurchschnittlich umfangreiche und komplexe OpenCL-Kernels verwendet. Ebenso tritt es vereinzelt auf, dass der VC4CC-Compiler gewisse Instruktionen nicht allgemeingültig umwandeln kann. D. h. es gibt vereinzelt Sonderfälle, die noch nicht unterstützt werden und somit nicht kompiliert werden können, wie das Umsortieren von SIMD-Vektoren, bei dem die Anordnung nicht zur Kompilierzeit feststeht. Ebenso beinhaltet der erzeugte Maschinencode für bestimmte Instruktionsfolgen noch Fehler, die erst bei der Ausführung ersichtlich werden, indem ein falsches Ergebnis berechnet wird.

5.2.2 Unterstützte Anwendungen

Wie man in Tabelle 5.2 sehen kann, wird derzeit noch keine der getesteten OpenCL-Bibliotheken vollständig unterstützt. Dies liegt wie bereits in Abschnitt 5.2.1 beschrieben, sowohl an der Unvollständigkeit der VC4CL-Implementierung als auch zum Teil an den Implementierungen der jeweiligen Bibliotheken, die nicht für eine so schwache Hardware angepasst sind. Jedoch werden einzelne Bibliotheken wie boost compute, welches GPU-beschleunigte Implementierungen für eine Vielzahl an Algorithmen bereit stellt, sowie EasyCL, eine Abstraktionsebene über die OpenCL-Laufzeitbibliothek, zumindest zu einem großen Teil (ca. 64%) unterstützt. Der Grad der Unterstützung richtet sich hierbei an den in Tabelle 5.2 aufgelisteten Anteil der unterstützten Testfälle der jeweiligen Bibliothek. Jedoch werden auch diese beiden Bibliotheken nur in einer modifizierten Version unterstützt, indem die Standardgröße einer Work-Group manuell auf einen Wert gesetzt wird, der von der VC4CL-Implementierung unterstützt wird (meistens auf 8 Work-Items). Im Gegensatz zu anderen Bibliotheken, wie ViennaCL oder der Bullet Physics Engine, ist es bei den Bibliotheken boost compute und EasyCL möglich, die standardmäßige Work-Group-Größe zu ändern, da diese keine algorithmische Relevanz hat, sondern die Bibliotheken mit einer beliebigen Work-Group-Größe arbeiten können.

Trotz der Tatsache, dass noch keine der getesteten Bibliothek komplett unterstützt wird, lässt sich die Funktionalität und Anwendbarkeit der VC4CL-Implementierung – zumindest in einem eingeschränkten Anwendungsbereich – trotzdem aufzeigen. Zum einen durch den in Tabelle 5.1 aufgelisteten Anteil an bestandenen Testfällen der OpenCL CTS (ca. 54% aller Testfälle werden unterstützt, ein Drittel hiervon aufgrund von Überspringen von Testfällen für nicht unterstützte Features). Zum anderen sind im Zuge dieser Arbeit Testprogramme entstanden, die von der VC4CL-Implementierung richtig kompiliert und ausgeführt werden. Hierunter zählt das bereits in Abschnitt 1.5 als Beispiel für die verschiedenen unterstützten Quellco-

deformate verwendete OpenCL-Programm zum Berechnen der zehn nächsten Fibonaccizahlen anhand zweier Ausgangswerte (siehe Anhang A.1). Ebenso ist zu Demonstrationszwecken das Projekt GameOfLife entstanden, welches Conway's Game Of Life sowohl über C++-Code auf der CPU als über einen OpenCL-Kernel auf einer GPU ausführen kann. Die Ausführung des GameOfLife-Projekts über die VC4CL-Implementierung berechnet das richtige Ergebnis, was die Richtigkeit zumindest des in diesem Kernel verwendeten Codes beweist. Jedoch ist die Performance im Vergleich zu der Ausführung auf der CPU eher schlecht. Die Gründe hierfür sind höchstwahrscheinlich der zusätzliche Overhead beim Starten der Kernel sowie der häufige Speicherzugriff aus dem Kernelcode heraus und werden in Abschnitt 5.3.3 genauer beschrieben.

5.3 Performance

Eine der entscheidenden Faktoren für die Verwendung von OpenCL (anstelle von hostseitiger Berechnung) ist die erwartete gesteigerte Performance für ausgewählte (meist mathematische) Berechnungen. Die in dieser Arbeit erstellte Implementierung dient zwar hauptsächlich als Proof-of-Concept (siehe Abschnitt 1.1), jedoch sollten damit durchgeführte Berechnungen wenigstens mit der Performance der Host-CPUs der Raspberry Pi-Modellen mithalten können. Daher werden in diesem Abschnitt sowohl die theoretische Maximalleistung der Implementierung als auch die Performance bei echten Benchmarks mit den jeweiligen Werten der Host-CPUs verglichen.

5.3.1 Theoretische Leistung

Abbildung 5.1 zeigt den Vergleich der theoretischen Leistung in GIOPS der CPUs der verschiedenen Raspberry Pi-Generationen und der VideoCore IV jeweils für einzelne Kerne sowie für den gesamten Prozessor. Hierbei gibt der untere ausgefüllte Balken die maximale Leistung ohne SIMD-Instruktionen und der obere schraffierte Balken die maximale Leistung unter Verwendung von SIMD-Instruktionen an. Die 1. Generation der Raspberry Pi-Modelle, die die Modelle A, A+, B und B+ sowie das Compute Module einschließt, besitzt einen Single-Core Prozessor mit einer Standardtaktrate von 700 MHz, was zu einer Maximalleistung von 700 MIOPS führt. Der Raspberry Pi Zero (in Abbildung 5.1 nicht aufgeführt) besitzt eine geringfügig bessere Leistung von 1 GIOPS, da er den gleichen Prozessorchip, jedoch mit einer Taktrate von 1GHz ausliefert. Raspberry Pi 2, als einziger Vertreter der Generation 2, besitzt einen 900 MHz Quad-Core Prozessor, der jedoch den ARM NEON-Befehlssatz [Wik17a, Abschnitt Advanced SIMD (NEON)] unterstützt, der bis zu 128-Bit SIMD-Befehle (vier parallele 32-Bit Ganzzahlberechnungen)

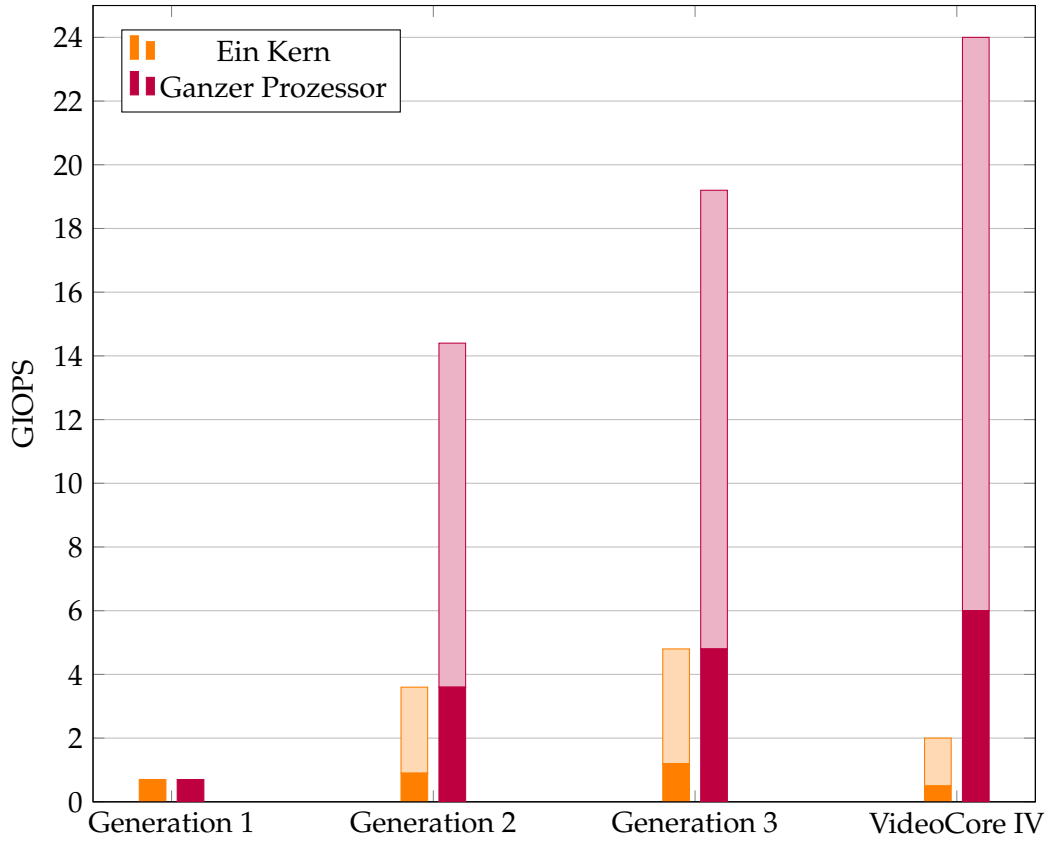

Abbildung 5.1: Vergleich der Prozessorleistungen der einzelnen Raspberry Pi-Modellen und der VideoCore IV GPU

bietet und somit eine Gesamtleistung von 14,4 GIOPS (900 MHz * vier Kerne * vierfachen SIMD) besitzt. Die Generation 3, bestehend aus dem Raspberry Pi 3 und dem Compute Module 3, wird mit einem 1,2 GHz Quad-Core, ebenfalls mit NEON-Unterstützung ausgeliefert und kommt so auf eine Maximalleistung von 19,2 GIOPS. Die theoretische maximale Rechenleistung der VideoCore IV GPU beträgt 24 GFLOPS/GIOPS. Diese ergeben sich aus einer Grundtaktrate von 250 MHz, einem vierfachen physikalischen Datenparallelismus, zwei asymetrischen ALUs und den zwölf in den Raspberry Pi-Modellen verbauten QPUs [Bro13, Kapitel 3]. Wenn man die theoretische Maximalleistung der VideoCore IV mit den Maximalleistungen der CPUs der verschiedenen Raspberry Pi-Modellen vergleicht, zeigt sich ein eindeutiges Bild, dass für gut parallelisierbare Aufgaben die VideoCore IV für alle Raspberry Pi-Modelle die beste theoretische Performance liefert. [Wik17f, Abschnitt Specifications]

Bei dem Vergleich in Abbildung 5.1 sind mehrere Punkte anzumerken: Es handelt sich um theoretische Maximalleistungen, die sich auf die Ausführung einfacher Ganzzahloperationen bezieht, die vom Befehlssatz des jeweiligen Prozessors unter-

stützt werden. Aufgrund des vereinfachten Befehlssatzes der QPUs schließt diese Angabe für die VideoCore IV GPU z. B. wichtige Operationen wie Ganzzahlmultiplikation oder Berechnung von Resten (Modulo) aus (siehe Abschnitt 2.2.4). Ebenso muss beachtet werden, dass die VideoCore IV GPU keinen äquivalente Funktionsumfang besitzt wie die CPUs im Vergleich, da nur Berechnungen ausgeführt werden können, jedoch kein Zugriff auf die Peripherie oder Datenträger möglich ist. Ein anderer wichtiger Faktor, der die tatsächliche Performance von dem theoretischen Maximum abweichen lässt, ist die Tatsache, dass für die theoretischen Werte davon ausgegangen wird, dass alle Instruktionen die volle SIMD-Breite ausnutzen können, was in realen Programmen nicht gegeben ist. Des Weiteren wird hier der nicht unbeachtliche Overhead für das Erstellen, Kompilieren und Starten eines OpenCL-Kernels vernachlässigt, sowie weitere Faktoren wie die Größe und Geschwindigkeit der Prozessorcaches und die verfügbare Speicherbandbreite. Im Abschnitt 5.3.2 wird die tatsächliche Performance der VideoCore IV GPU mit der Host-CPU eines Raspberry Pi-Modells verglichen.

5.3.2 Benchmarks

Der Japaner Yukimasa Sugizaki hat eine Reihe sehr interessanter Benchmarks und Timing-Analysen auf der VideoCore IV GPU durchgeführt und dokumentiert [Sug16]. Die Ergebnisse sind zum größten Teil auf Japanisch, lassen sich jedoch mithilfe von Online-Übersetzungswerkzeugen wie Google Translate in ein einigermaßen verständliches Englisch übersetzen. Unter anderem werden auf seinen Blog experimentell die Funktionsweise des Befehlscache, des Performance Counter [Sug16, Eintrag QI3] und die Verzögerungen bei Verwendung von Mutex [Sug16, Eintrag QI5,6] und Semaphoren [Sug16, Eintrag QI10] analysiert. Von größerem Interesse sind jedoch die Experimente zur Bestimmung der maximalen Speicherbandbreite beim Lesen und beim Schreiben in den Blogeinträgen QV52 und QV56. Hier wurden beim Lesen eine maximale Bandbreite von ca. 690 MB/s gemessen, die beim Lesen von 16 16-fachen 32-Bit SIMD-Vektoren (z. B. 16 aufeinander folgenden `int16` Werten) erreicht werden kann [Sug16, Eintrag QV52]. Beim Schreiben in den Arbeitsspeicher wurde ein Maximalwert von 1,12 GB/s beim Schreiben von 128 16-fachen 32-Bit SIMD-Vektoren und eine Bandbreite von ca. 300 MB/s beim Schreiben eines einzelnen 16-fachen SIMD-Vektors gemessen [Sug16, Eintrag QV53]. Diese Geschwindigkeiten beziehen sich jedoch nur auf die Übertragungsraten zwischen dem Hauptspeicher und der VPM und nicht der Übertragungsraten zwischen dem Hauptspeicher und den QPUs (siehe Abbildung 1.2). Die maximale theoretische Bandbreite zwischen einer QPU und der VPM beträgt bei einem Takt von 250 MHz, 4-fachen physikalischen Datenparallelismus (siehe Abschnitt 5.3.1) und 4-Byte Datentypen bis zu 4 GB/s, da der Zugriff auf die VPM keine weiteren Verzögerungen einführt. In der Praxis wird aber keiner der gemessenen Maximalwerte

erreicht werden, da OpenCL-Kernel meist nur wenige bis nur einzelne Werte am Stück aus zusammenhängenden Speicherbereichen lesen oder in diese schreiben, weswegen standardmäßig jeder einzelne in die VPM geschriebene Wert per DMA an den Arbeitsspeicher weitergegeben wird und auch jeder Wert einzeln aus dem Arbeitsspeicher in die VPM und dann in die Register der QPU gelesen wird (siehe Abschnitt 2.2.5 für eine Optimierung der Lade-/Schreibvorgänge). Sugizaki misst bei dem Lesen und Schreiben einzelner 16-fachen SIMD-Vektoren zwischen dem Hauptspeicher und der VPM Übertragungsraten von ca. 200 MB/s sowie 300 MB/s [Sug16, Einträge QV52 und QV53].

Für die Analyse der tatsächlichen Performance wird das OpenCL-Benchmarkprogramm clpeak verwendet. clpeak ist ein Kommandozeilenprogramm, zum Messen der „Höchstleistung von OpenCL-Geräten" [Kri17]. Das Benchmarkprogramm enthält vier Benchmarks zum Prüfen der maximalen Rechenleistung sowie zwei Benchmarks zum Ermitteln der verfügbaren Speicherbandbreite. Die Rechenbenchmarks werden für die Datentypen `int`, `float`, `half` (also 16-Bit Fließkommazahl) sowie `double` durchgeführt und führen jeweils eine Folge von Additions- und Multiplikationsinstruktionen für skalare Werte sowie Vektoren mit 2, 4, 8 und 16 SIMD-Elementen aus. Die Berechnungen sind so aufgebaut, dass jede Instruktion das Ergebnis der vorherigen benötigt, weswegen sich Optimierungen, wie das Kombinieren von Instruktionen (siehe Abschnitt 2.2.7) kaum anwenden lassen. Zum Testen der Speicherbandbreite gibt es ein Benchmark `transfer_bandwidth`, der nur hostseitig das Schreiben in und Lesen auf OpenCL-Buffern testet. Ein weiterer Benchmark `global_bandwidth` führt OpenCL-Kernel aus, die Daten von einem Eingabepuffer in einen Ausgabepuffer kopieren und testet somit die Geschwindigkeit, mit der das OpenCL-Gerät auf den eigenen Speicher zugreifen kann. [Kri17]

Da die VC4CL-Implementierung weder `double` noch `half` Datentypen unterstützt, werden diese Benchmarks nicht ausgeführt. Ebenso wird der Ganzzahlbenchmark übersprungen, da der hierfür generierte Code noch fehlerhaft ist und die verwendeten QPUs zum Einfrieren bringt. Alle anderen Benchmarks werden doppelt ausgeführt: Einmal werden die Kernel über die Mailbox-Schnittstelle und einmal über direktes Schreiben in die Peripherieregister gestartet (siehe Abschnitt 4.2.4), um den zusätzlichen Overhead der Mailbox-Aufrufe zu testen. In beiden Fällen wird auf allen zwölf QPUs gleichzeitig gerechnet. Die Ergebnisse werden mit den Ergebnissen der Benchmarks auf der CPU-seitigen OpenCL-Implementierung pocl [pd10] verglichen. Die Vergleichsergebnisse der clpeak-Benchmarks, ausgeführt mit pocl auf einem Raspberry Pi 2 (2. Generation) sind dem offiziellem clpeak github-Repository entnommen [Kri15].

Die Grafik 5.2 stellt die Ergebnisse der durchgeführten clpeak Benchmarks im Vergleich zueinander und zu den Ergebnissen der pocl-Implementierung dar. Auf der linken Skala auf der y-Achse ist die erreichte Rechenleistung in GFLOPS des Fließ-

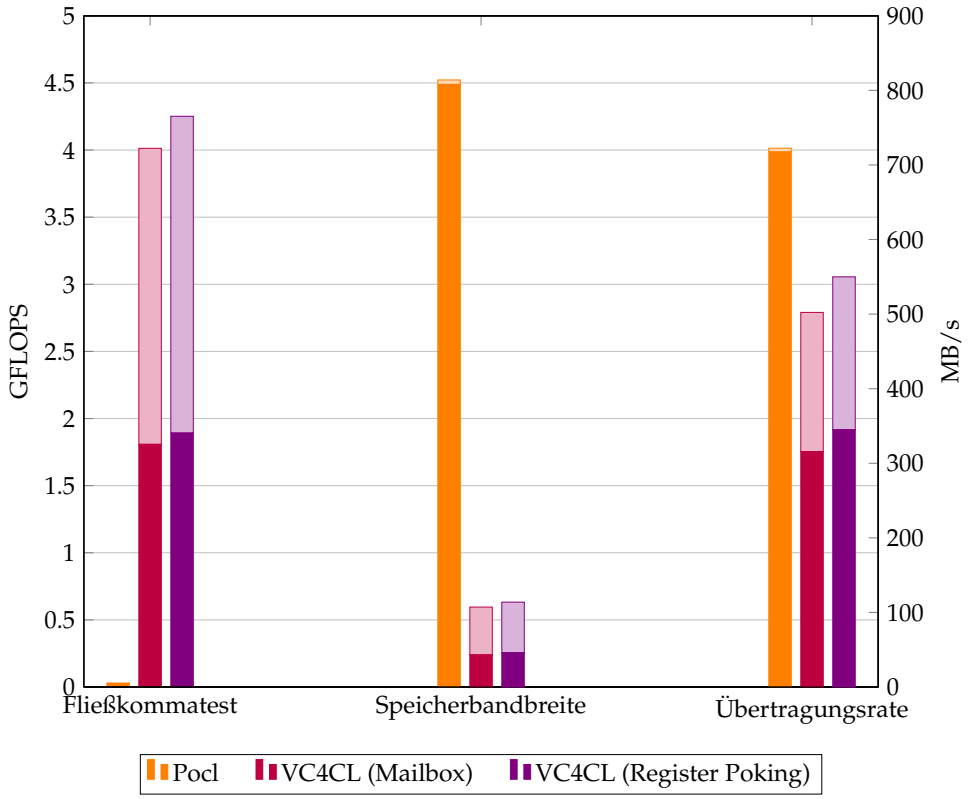

Abbildung 5.2: Vergleich der Ergebnisse der clpeak-Benchmarks

kommabenchmarks angegeben, während die rechte Skala die Speicherbandbreite der Benchmarks für das hostseitige Lesen und Schreiben von Buffern („Übertragungsrate") sowie die GPU-seitige Speicheranbindung („Speicherbandbreite") angibt. Die unteren ausgefüllten Balken geben jeweils das arithmetische Mittel der einzelnen Messwerte an (das Mittel der Ergebnisse für skalare Werte sowie 2-, 4-, 8- und 16-fachen SIMD-Vektoren) und die oberen schraffierten Balken zeigen das Maximum des jeweiligen Tests an (das z. B. beim Fließkommabenchmark für 16-fache SIMD-Vektoren erreicht wird). An den Ergebnissen kann man die Vermutungen aus Abschnitt 5.3.1 gut erkennen. Auf der einen Seite ist die pure Rechenleistung auf der GPU deutlich höher als bei gleichwertigen Berechnungen auf der CPU. Dies wird jedoch mit einer langsameren Speicheranbindung erkauft. Unter der Berücksichtigung, dass die Zeiten um die Funktion zum Starten eines Kernels herum gemessen werden und somit den Overhead zum Anlegen des Speichers für den Kernelcode sowie Kopieren des Kernelcodes und zum Ermitteln der Work-Item-Parameter beinhalten, sind die gewonnenen Ergebnisse sehr zufrieden stellend. Von den (ohne Kombination von Operationen auf der Additions- und Multiplikations-ALU) maximalen 12 GFLOPS (siehe Abschnitt 5.3.1) werden bis zu 4 GFLOPS, also ein Drittel des theoretischen Maximums erreicht. Ein ähnliches Bild ergibt sich bei der Speicherbandbreite: Hierfür hat Sugizaki für das Lesen und Schreiben einzelner

16-fachen SIMD-Vektoren eine Maximalleistung von ca. 200 MB/s beim Lesen und ca. 300 MB/s beim Schreiben [Sug16, Einträge QV52 und QV53] gemessen. Erreicht werden davon im Benchmark zur Speicherbandbreite bis zu 112 MB/s.

Ebenso zeigt die Grafik 5.2, wie erwartet (siehe Abschnitt 4.2.4), einen kleinen Performanceunterschied zwischen den Ausführungen der OpenCL-Kernel über den Mailbox-Systemaufruf und durch direktes Schreiben in die Peripherieregister. Den Performanceunterschied der Ausführungsmethoden kann man sich auch mit Profilingtools für die VideoCore IV-Hardware, wie dem im Projekt VC4CL mitgelieferten Kommandozeilenprogramm `v3d_profile`, aufzeigen lassen: Während eine Ausführung des clpeak-Fließkommatests direkt über das Schreiben der Register alle QPUs bis zu 100% (durchgehend über 95%) auslastet, erreicht die Ausführung über den Mailbox-Systemaufruf „nur" eine Auslastung von 93 bis 97%. Ebenso lässt sich sehr gut die Wartezeit bei Speicheranfragen in der DMA-Schnittstelle (siehe Abschnitt 2.2.5) erkennen, da bei beiden Ausführungsmethoden die Auslastung des Speicherzugriffstests `global_bandwidth` einen Maximalwert von ca. 56% erreicht. Die restliche Zeit wird durch das Warten auf Verfügbarkeit der Daten in der VPM (oder das Warten auf die Fertigstellung des Schreibvorgangs in den Arbeitsspeicher) verbraucht. Anhand der Höhe des Performanceunterschieds von ca. 5 bis 10 % und der Eigenschaft, dass der Mailbox-Aufruf im Gegensatz zum direkten Schreiben der Peripherieregister systemweit synchronisiert ist und somit gegen gleichzeitige Zugriffe auf die VideoCore IV abgesichert ist, kann ein Anwender entscheiden, welche der Methoden zum Ausführen von OpenCL-Kernel er nutzen will.

5.3.3 Bottlenecks

Die Ausführung von Code auf der VideoCore IV GPU ist zwar für einfache, nativ unterstützte, arithmetische Befehle sehr effizient (siehe Abbildung 5.2), jedoch kann die tatsächliche Performance manchmal sehr stark davon abweichen. Neben dem Overhead zum Starten und Beenden von Kernelcode lassen sich die meisten Performanceeinbußen auf eine Hand voll Punkte zurückführen:

Manche, auch häufig verwendeten, arithmetischen Operationen, sowie die Division (ganzer Zahlen und Fließkommazahlen) und die allgemeine Form der Multiplikation ganzer Zahlen werden vom Befehlssatz der QPUs nicht unterstützt und müssen daher in Software emuliert werden (siehe Abschnitt 2.2.4). Dadurch benötigen diese Operationen jedoch viel mehr Taktzyklen, als eine Berechnung in Hardware benötigen würde. So benötigt die vorzeichenlose Ganzzahlmultiplikation bereits elf, die vorzeichenbehaftete Variante 27 Instruktionen. Die Berechnung der Ganzzahldivision hingegen benötigt sogar mehr als 200 Taktzyklen. Ebenso muss die Fließkommadivision emuliert werden, was ca. 24 Instruktionen kostet. Der Geschwindigkeitsverlust durch Verwendung der nicht in Hardware verfügbaren

Operationen lässt sich nicht umgehen, da die arithmetischen Operationen meist nicht durch andere Operationen ersetzt werden können. Jedoch kann, wenn der gültige Wertebereich der Operanden eingeschränkt ist, dem Compiler geholfen werden, effizientere Implementierungen der Operationen zu generieren. So wird z. B. eine vorzeichenlose Ganzzahlmultiplikation zweier Zahlen vom Typ `short` oder `char` auf eine einzige ALU-Instruktion (`mul24`) abgebildet. Ebenso wird die im OpenCL-Standard definierte Funktion `mul24` in die gleiche Hardwareinstruktion umgewandelt. Ähnlich werden für eine vorzeichenlose Ganzzahldivision mit Operanden vom Typ `short` gleich ca. 100 Instruktionen weniger benötigt als bei der Division der vollen der 32-Bit. Für die Fließkommadivision bietet sich an, falls das Ergebnis nicht genau sein muss, die OpenCL-Standardfunktion `native_divide` zu verwenden, die vom VC4CC-Compiler in eine Berechnung des multiplikativen Invsersem mithilfe der SFU (siehe Abschnitt 1.3.1) und eine Multiplikation umgewandelt wird, was insgesamt fünf Taktzyklen benötigt.

Ein weiterer Flaschenhals ist der Speicherzugriff: Da sich alle QPUs eine VPM und damit den Arbeitsspeicherzugriff teilen (siehe Abschnitt 1.3.1) und die VPM jeweils nur einen Lese- und einen Schreibvorgang zeitgleich abwickeln kann, wird jeder Zugriff auf den Hauptspeicher über das Hardware-Mutex synchronisiert (siehe Abschnitt 2.2.3). Das bedeutet, dass bei einer parallelen Ausführung eines Kernels auf allen 12 verfügbaren QPUs mindestens beim ersten Speicherzugriff elf der QPUs warten müssen, bis der erste Prozessor seinen Speicherzugriff abgeschlossen hat. Die QPU, die als letztes an der Reihe ist, wird sogar elf Speicherzugriffe lang blockiert, was wiederum mehr als 100 Taktyzyklen Blockierzeit ergibt. Speicherzugriffe lassen sich nicht vermeiden, da OpenCL-Kernel keine Rückgabewerte haben können und auch die Parameter meist nur die Adressen der zu verwendenden Speicherbereiche sowie konstante Werte beinhalten. Jedoch kann versucht werden, die Speicherzugriffe so gering wie möglich zu halten, indem inital die benötigten Werte gelesen werden und am Ende wieder in den Hauptspeicher geschrieben werden, sämtliche Berechnungen jedoch auf lokalen Variablen stattfinden.

Ebenso entsteht bereits beim Starten von Kernel-Ausführungen ein nicht unerheblicher Overhead aufgrund des Systemaufrufs für die Kommunikation mit der GPU durch den Linux-Kernel (siehe Abschnitte 4.2.4 und 5.3.2). Dieser Overhead ist konstant, d. h. er ist bei jeder Ausführung eines OpenCL-Kernels ungefähr gleich, unabhängig von der Codelänge des ausgeführten Kernels. Dieser Overhead lässt sich verkleinern, indem die GPU direkt über die Peripherieregister gesteuert wird. Wie aber bereits in Abschnitt 4.2.4 erklärt, kann auf die Weise keine systemweite Synchronisation der Zugriffe auf die GPU erfolgen. Daher wird zusätzlich eine andere Optimierung angewendet, um den Einfluss dieses Overheads zu verringern: Der ausgeführte Code eines OpenCL-Kernels ist für alle Work-Items aller Work-Groups gleich. Allein die Parameter, genauer gesagt nur die „versteckten" Parameter für die

lokale ID und die ID der Gruppe, ändern sich zwischen den ausgeführten Instanzen eines Kernels. Daher bietet es sich an, mit einem Systemaufruf mehrere Instanzen hintereinander zu starten, die jeweils nur abgeänderte Parameter besitzen. Hierfür wird um den Maschinencode des Kernels eine Schleife hinzugefügt, die die komplette Ausführung wiederholt. Über einen weiteren „versteckten" Parameter wird bestimmt, wie oft der Kernel-Code ausgeführt werden soll. Da die Wiederholung des gesamten Maschinencodes auch das Laden der Parameter wiederholt, müssen beim Starten die Parameter aller Wiederholungen gesetzt werden. So werden z. B. mit einem Mailbox-Aufruf bis zu acht Work-Groups hintereinander ausgeführt, ohne das zwischendurch die CPU wieder tätig werden muss.

5.3.4 Compiler

Da OpenCL-Kernel meist, im Gegensatz zu den meisten hostseitigen Programmen (ausgenommen Programmiersprachen mit Just-In-Time Kompilierung), nicht im Voraus, sondern während der Laufzeit des Programms kompiliert werden, ist die Performance des Compilers auch von Bedeutung. Um die Geschwindigkeit der Kompilierung zu verbessern, wurde iterativ in folgenden Schritten vorgegangen: Als Erstes wurde über Profiling anhand der Kompilierung einer Vielzahl an OpenCL-Kerneln die Ausführungsdauer möglicher Kandidaten für eine Verbesserung gemessen. Daraufhin wurden die langsamsten Methoden genauer analysiert, wie sie ihre Funktionalität erfüllen und ob es eine effizientere Alternative gibt. Erfahrung hat gezeigt, dass eine der einfachsten und auch effizientesten Möglichkeiten zur Verbesserung der Performance eines Codes darin besteht, die verwendeten Datenstrukturen exakt auf den jeweiligen Anwendungsfall anzupassen. So eignen sich z. B. bei häufigem Einfügen und Löschen von Elementen verkettete Listen besser als die Speicherung in einem kontinuierlichen Speicherbereich (z. B. ein Array oder Vector), die jedoch wiederum bei seltener Modifikation der Struktur und häufiger Iteration über die Elemente effizienter ist. Ebenso kann die Geschwindigkeit einer Funktion zum Teil deutlich erhöht werden, indem Zwischenergebnisse gespeichert werden, anstatt mehrfach neu berechnet zu werden. Ein Beispiel hierfür ist die Verwendung eines „Precompiled Headers" für die Vorkompilierung (siehe Abschnitt 2.1.1). Die aufwendigste und auch fehleranfälligste Möglichkeit, die Geschwindigkeit einer Funktion zu verbessern ist es, den verwendeten Algorithmus zu wechseln. Hierbei kann jedoch auch der Performancegewinn mäßig ausfallen. Eine weitere Optimierung, welche die gesamte Ausführungszeit des Compilers um einiges verkürzt, ist das Parallelisieren von Kompilierungsschritten. So werden im VC4CC-Compiler die Optimierungen und auch die Registerzuweisung der lokalen Variablen jeder im Quellcode enthaltenen Kernelfunktion parallel ausgeführt. Dies ist möglich, da nach dem initialen Inlining aller Funktionen in die jeweilige Kernelfunktion (siehe Abschnitt 2.2.1) diese sich nicht mehr gegenseitig beeinflussen.

Funktion	Gesamtdauer	Anzahl Aufrufe	durchschnittliche Dauer
Precompile	439547 ms	238	1846 ms
Parser	17900 ms	203	88 ms
Optimizer	26493 ms	176	150 ms
CodeGenerator	192991 ms	303	636 ms

Tabelle 5.3: Profilingergebnisse des VC4C Compilers anhand von 239 OpenCL-Kernel (Ausschnitt)

Trotz des großen Aufwands, den Compiler performant zu programmieren, nimmt die Kompilierung von OpenCL-Programmen in Maschinencode für die VideoCore IV GPU immer noch sehr viel Zeit in Anspruch. Die Tabelle 5.3 zeigt einen Ausschnitt aus den Profilingergebnissen für die grundlegenden Schritte bei der Kompilierung eines OpenCL C-Quellcodes, Vorkompilierung (Abschnitt 2.1.1), Parsen (Abschnitt 2.1), Optimieren und Umwandeln (Abschnitt 2.2) und Codegenerierung (Abschnitt 2.3). Anhand der Ergebnisse des Profilers zeigt sich deutlich, dass die meiste Zeit in der Registerzuweisung (die anderen Teile der Codegenerierung haben vernachlässigbare Ausführungszeiten) sowie der Vorkompilierung verbracht wird.

5.4 Fazit

Der Abschnitt 5.2 zeigt, das derzeit noch keine der getesteten OpenCL-Bibliotheken von der VC4CL-Implementierung komplett unterstützt wird. Ebenso ist die Inkompatibilität sowohl auf eine unvollständige und noch fehleranfällige VC4CL-Implementierung als auch auf die Implementierungen der getesteten Bibliotheken zurückzuführen, die teilweise sehr einschränkende Annahmen treffen, was die Leistung der zugrundeliegenden Hardware angeht (siehe Abschnitt 5.2.1). Jedoch zeigt Abschnitt 5.2.2 auch auf, dass es grundsätzlich möglich ist, auf OpenCL basierende Anwendungen auf der VC4CL-Implementierung auszuführen, wenn auch mit teils deutlichen Einschränkungen. Die Tabelle 5.1 zeigt anhand der ausgeführten Testfälle der OpenCL Conformance Test Suite auf, wie weit welche Aspekte des OpenCL 1.2-Standards unterstützt werden und bereits verwendet werden können.

Die Ergebnisse des Performancevergleichs zwischen einer Ausführung von OpenCL-Benchmarks über die VC4CL-Implementierung auf der VideoCore IV GPU und der pocl-Implementierung auf der Host-CPU eines Raspberry Pis in Abschnitt 5.3.2 beweisen die im Zuge dieser Arbeit aufgestellte Vermutungen, dass die Ausführung von rechenintensiven Code auf der GPU einen hohen Performancegewinn hervorrufen kann, der jedoch mit einer langsameren Speicherbandbreite erkauft wird. Da die meisten OpenCL-Kernel anteilig mehr Berechnungen ausführen als sie auf den Speicher zugreifen, ist zumindest für bisher unterstützte Anwendungen eine deutliche Performancesteigerung gegenüber auf pocl ausgeführten OpenCL-Programmen zu

erwarten.

Diese Arbeit stellt einen guten Anfang für eine OpenCL-Implementierung dar, die es erlaubt, die verhältnismäßig starke Rechenleistung der VideoCore IV-Hardware (siehe Grafik 5.1) für nicht-grafische Berechnungen verwenden zu können und somit die Gesamtleistung der Raspberry Pi-Modelle für eine Vielzahl von Anwendungsfällen zu erhöhen. Ebenso wird aufgezeigt, dass trotz der Bedenken von an der Entwicklung der Raspberry Pi-Modelle oder deren Treiber beteiligten Entwickler (vgl. Kapitel 1) zumindest eine rudimentäre Unterstützung von OpenCL für die VideoCore IV GPU möglich ist.

6. Ausblick

Die Entwicklung der VC4CL OpenCL-Implementierung ist mit Abgabe dieser Arbeit keinesfalls abgeschlossen. Stattdessen wird versucht werden, die Unterstützung von OpenCL-basierter Software zu verbessern, bestehende Umsetzungen mancher Features zu verbessern sowie weitere Features und Erweiterungen des OpenCL-Standards zu unterstützen.

Allgemein können alle der in Kapitel 5 gezeigten Ziele noch verbessert werden. Daher ist es ein Ziel für die weitere Entwicklung, mehrere OpenCL-basierte Bibliotheken möglichst vollständig zu unterstützen sowie die Konformität mit der offiziellen OpenCL CTS Testsuite zu verbessern. Um dies zu erreichen, ist es auch notwendig, die Genauigkeit der Implementierungen der mathematischen Funktionen der OpenCL C-Standardbibliothek (siehe Abschnitt 3.2.1) zu erhöhen sowie ein korrektes Grenzwertverhalten (z. B. Unendlich, NaN) zu implementieren. Ebenso verbesserungswürdig ist die Performance, vor allem beim Compiler und manchen mathematischen Funktionen. Für die Verbesserung der Performance und auch der Genauigkeit der mathematischen Funktionen bietet es sich an, verschiedene Implementierungen testweise zu implementieren, die Genauigkeit der Ergebnisse zu berechnen (oder zu messen) sowie auf Basis des erzeugten Maschinencodes die performanteste Implementierung auszuwählen, die den maximalen erlaubten Fehler nicht überschreitet. Auf der anderen Seite sollten die Entwickler der getesteten Bibliotheken auf Inkompatibilitäten vonseiten der Bibliotheken hingewiesen werden sowie mögliche Lösungen vorgeschlagen werden.

Neben den möglichen Verbesserungen bestehender Features gibt es auch bereits eine Liste möglicher Erweiterung um neue Features. So könnte, wie bereits in den Abschnitten 2.2.5 und 2.3.2 beschrieben, die VPM als Cache für den Speicherzugriff sowie als Speicher für ausgelagerte Register verwendet werden, was einerseits die Performance bei häufigen Speicherzugriffen erhöht und andererseits mehr OpenCL-Kernel unterstützt, für die bisher keine passende Registerzuordnung gefunden werden kann. Ebenso könnten die Lookup-Tabellen für tabellenbasierte Implementierung mathematischer Algorithmen (siehe Abschnitt 3.2.1) in die VPM ausgelagert werden, wodurch deren Zugriffszeit deutlich erhöht werden kann und somit diese Implementierungen effizienter werden könnten. Weiterhin wurden erste Gedankenexperimente durchgeführt, die auf der VideoCore IV GPU verfügbare Hardware zum Laden von Texturen zu verwenden, um die Bearbeitung von Bildern zu unterstützen. Die Texture Lookup Unit (TMU, siehe Abbildung 1.2) unterstützt bereits

die meisten der vom OpenCL-Standard geforderten Formate und Interpolationsmethoden und würde somit zumindest das Laden von Pixelwerten um einiges vereinfachen. Eine weitere angedachte Erweiterung ist die Unterstützung des 16-Bit Fließkommatyps `half`. Auch hier bietet die VideoCore IV-Hardware bereits Möglichkeiten, 16-Bit Fließkommazahlen aus der VPM zu Lesen und nach 32-Bit Fließkommazahlen umzuwandeln, sowie `float`-Werte als `half`-Werte in die VPM zu Schreiben. Somit können sämtliche Berechnungen auf `half`-Werten als 32-Bit Fließkommaberechnungen durchgeführt werden und nur beim Lesen und Schreiben in die VPM muss eine Konvertierung durchgeführt werden.

Glossar

Basisblock

Block zusammenhängender Instruktionen, für die gilt, dass wenn die erste Instruktion dieses Blocks ausgeführt wird, alle weiteren Instruktionen des Blocks ebenso ausgeführt werden. D. h. Ein Basisblock beinhaltet (außer als erste Instruktion) keine Sprungziele und auch nur maximal einen Sprungbefehl (als letzte Instruktion)[Wik17b].

GFLOPS

10^9 Fließkommaoperationen pro Sekunde, eine Maßeinheit für Rechengeschwindigkeit. In dieser Arbeit werden 32-Bit-Fließkommazahlen (`float`) als Datentyp genommen.

GIOPS

10^9 Ganzzahloperationen pro Sekunde, eine weitere Maßeinheit für Rechengeschwindigkeit. In dieser Arbeit bezieht sich der Wert auf 32-Bit-Ganzzahlberechnungen.

GPGPU

General Purpose Computation on Graphic Processor Units, Verwendung von Grafikprozessoren für nicht-grafische Berechnungen.

Kernel

OpenCL-Kernel, Einstiegsfunktion für die Ausführung von in OpenCL C geschriebenen Code, vergleichbar mit der `main`-Funktion in C-Sprachen.

LLVM

Compiler-Architektur mit einer Vielzahl an Frontends (z. B. CLang für C, C++ und OpenCL) sowie Backends for verschiedene Architekturen.

LLVM IR

LLVM Intermediate Representation, Assembler-ähnliche Zwischensprache, SSA-basiert. Die verschiedenen LLVM-Frontends wandeln den verarbeiten Code in LLVM IR um, auf der die weiteren Schritte des Compilers (z. B. Optimierungen) durchgefüht wird.

Mailbox

Teil der Raspberry Pi-Firmware, leitet Anfragen an ein Modul des Linux-Kernels weiter, um so nachrichtenbasierten Zugriff auf die VideoCore IV aus dem Userspace heraus zu bieten.

QPU

Quad Processing Unit, 16-facher virtueller SIMD-Prozessor, frei programmierbar. Die QPUs stellen die Prozessorkerne der VideoCore IV GPU dar.

SFU

Special Functions Unit, Prozessor zum Annähern von mathematischen Funktionen wie die Quadratwurzel, das multiplikative Inverse der Quadratwurzel, den Zweierlogarithmus und die Exponentialfunktion auf der Basis 2. [Bro13, Kapitel 3].

SIMD

Single Instruction Multiple Data, Verarbeitung von mehreren Werten mit einer einzelnen Instruktion, d. h. die Hardware bietet Möglichkeiten, einen Befehl auf mehrere Datenwerte gleichzeitig auszuführen.

SPIR-V

Von der Khronos Group standardisierte Zwischensprache, die speziell für die Ausführung auf Grafikprozessoren entwickelt ist und als Zwischensprache für z. B. OpenGL-, Vulkan- und OpenCL-Anwendungen dient, ebensfalls SSA-basiert.

SSA

Static Single Assignment, eine Eigenschaft einer Intermediate Representation, die besagt, dass eine Variable genau ein einziges Mal einen Wert zugewiesen bekommt [Wik17g].

VideoCore IV

SoC Grafikprozessor der Raspberry Pi-Modelle, bestendend aus einer VPU (zur Steuerung der QPUs) und 12 QPUs (die eigentlichen Prozessorkerne).

VPM

Vertex Pipe Memory, die Speicherschnittstelle der VideoCore IV GPU, über die die Adressierung des Hautspeichers, sowie die Modi zum Lesen und Schreiben des Hauptspeichers konfiguriert werden. Alle QPUs teilen sich eine VPM, die zusätzlich einen kleine Cache besitzt [Bro13, Kapitel 7].

Work-Group

Sammlung von mehreren Work-Items, die parallel auf einer einzelnen Recheneinheit ausgeführt werden. Alle Work-Items einer Work-Group führen den gleichen Kernel mit den gleichen Parametern aus und teilen sich den lokalen Speicher sowie Speicherbarrieren [Gro12, Kapitel 2].

Work-Item

Einzelne Ausführung eines Kernels, kann durch einen oder mehreren Prozessoren ausgeführt werden, ist Teil einer Work-Group und hat eine eindeutige lokale und globale ID [Gro12, Kapitel 2].

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literaturverzeichnis

[Arn10] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag GmbH, 2010.

[CGH+10] Christopher Cameron, Benedict Gaster, Michael Houston, John Kessenich, Christopher Lamb, Laurent Morichetti, Aftab Munshi, und Ofer Rosenberg. Khronos ICD Loader. `https://www.khronos.org/registry/OpenCL/extensions/khr/cl_khr_icd.txt`, März 2010. Letzter Zugriff: 11.07.2017.

[EC13] Robert Elliott und Hui Chen. Arm_core_id. `https://www.khronos.org/registry/OpenCL/extensions/arm/cl_arm_get_core_id.txt`, April 2013. Letzter Zugriff: 11.07.2017.

[EP16] Robert Elliott und Mats Petersson. Arm_shared_virtual_memory. `https://www.khronos.org/registry/OpenCL/extensions/arm/cl_arm_shared_virtual_memory.txt`, März 2016. Letzter Zugriff: 11.07.2017.

[Gro12] Khronos OpenCL Working Group. The OpenCL Specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`, November 2012. Letzter Zugriff: 04.05.2017.

[Gro15] Khronos OpenCL Working Group. The OpenCL Extension Specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2-extensions.pdf`, September 2015. Letzter Zugriff: 05.05.2017.

[Gro16] Khronos OpenCL Working Group. The OpenCL Extension Specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-extensions.pdf`, März 2016. Letzter Zugriff: 11.07.2017.

[KN14] Michael Kinsner und David Neto. cl_altera_device_temperature. `https://www.khronos.org/registry/OpenCL/extensions/altera/cl_altera_device_temperature.txt`, Februar 2014. Letzter Zugriff: 11.07.2017.

[Per08] Fernando Magno Quintao Pereira. *Register Allocation by Puzzle Solving*. phdthesis, University of California, Los Angeles, `http://homepages.dcc.ufmg.br/~fernando/publications/papers/PhdDiss.pdf`, 2008. Letzter Zugriff: 06.07.2017.

[PFTV92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, und William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[Rod08] Thomas L. Rodeheffer. Software Integer Division. Technical report, Microsoft Research, Silicon Valley, August 2008. `https://www.microsoft.com/en-us/research/wp-content/uploads/2008/08/tr-2008-141.pdf`.

[WAG17] Jian-Zhong Wang, Bastiaan Aarts, und Vinod Grover. Loop unroll pragma extension. `https://www.khronos.org/registry/OpenCL/extensions/nv/cl_nv_pragma_unroll.txt`, 2017. Letzter Zugriff: 11.07.2017.

[Zha99] Michael Ruogu Zhang. Software Floating-Point Computation on Parallel Machines. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Mai 1999. `https://dspace.mit.edu/bitstream/handle/1721.1/80133/43609668-MIT.pdf`, Letzter Zugriff: 08.06.2017.

Online-Quellen

[Anh15] Eric Anholt. Open Source OpenGL on the Raspbery Pi. `http://www.anholt.net/papers/lca2015-rpi.pdf`, Januar 2015. Letzter Zugriff: 01.10.2017.

[Anh16] Eric Anholt. This week in vc4 (2016-10-11): QPU user shaders, Processing performance. `http://anholt.livejournal.com/48404.html`, Oktober 2016. Letzter Zugriff: 02.05.2017.

[Bro13] Broadcom. VideoCore® IV 3D Architecture Reference Guide. `https://docs.broadcom.com/docs-and-downloads/docs/support/videocore/VideoCoreIV-AG100-R.pdf`, September 2013. Letzter Zugriff: 07.05.2017.

[Fou17a] Raspberry Pi Foundation. Mailbox property interface. `https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface`, Februar 2017. Letzter Zugriff: 01.10.2017.

[Fou17b] Raspberry Pi Foundation. Raspberry pi cross compiler suite. https://github.com/raspberrypi/tools, Mai 2017. Letzter Zugriff: 19.05.2017.

[Gro15] Khronos Group. SPIR-V - A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels. https://www.khronos.org/registry/spir-v/papers/WhitePaper.html, März 2015. Letzter Zugriff: 02.05.2017.

[Gro17a] Khronos Group. Khronos OpenCL Registry. https://www.khronos.org/registry/OpenCL/, Mai 2017. Letzter Zugriff: 04.05.2017.

[Gro17b] Khronos Group. Khronos Releases OpenCL 2.2 With SPIR-V 1.2. https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2, Mai 2017. Letzter Zugriff: 15.06.2017.

[Gro17c] Khronos Group. SPIR-V Tools. https://github.com/KhronosGroup/SPIRV-Tools, Mai 2017. Letzter Zugriff: 14.05.2017.

[Gro17d] Khronos Group. SPIRV-LLVM. https://github.com/KhronosGroup/SPIRV-LLVM, April 2017. Letzter Zugriff: 02.05.2017.

[Gro17e] Khronos Group. The Khronos Group Inc. https://www.khronos.org/, 2017. Letzter Zugriff: 04.05.2017.

[Hal14a] Simon J. Hall. LLVM backend for QPU development. https://www.raspberrypi.org/forums/viewtopic.php?t=78919, Mai 2014. Letzter Zugriff: 02.05.2017.

[Hal14b] Simon J. Hall. LLVM QPU. https://github.com/simonjhall/llvm_qpu, Juni 2014. Letzter Zugriff: 02.05.2017.

[Her16] Herman H. Hermitage. Readme.md. https://github.com/hermanhermitage/videocoreiv/blob/master/README.md, Mai 2016. Letzter Zugriff: 07.05.2017, Revision 51f3531.

[Kri15] Bhat Krishnaraj. Raspberrypi2.log. https://github.com/krrishnarraj/clpeak/blob/master/results/Portable_Computing_Language/RaspberryPI2.log, Oktober 2015. Letzter Zugriff: 22.09.2017.

[Kri17] Bhat Krishnaraj. clpeak. `https://github.com/krrishnarraj/clpeak`, Juni 2017. Letzter Zugriff: 15.08.2017.

[LLV17a] LLVM. Attributes in Clang. `https://clang.llvm.org/docs/AttributeReference.html`, Juli 2017. Letzter Zugriff: 11.07.2017.

[LLV17b] LLVM. LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`, April 2017. Letzter Zugriff: 02.05.2017.

[LLV17c] LLVM. Precompiled Header and Modules Internals. `http://clang.llvm.org/docs/PCHInternals.html`, Juni 2017. Letzter Zugriff: 15.06.2017.

[Mes17] Mesa3D. The Mesa 3D Graphics Library. `https://www.mesa3d.org/intro.html`, Oktober 2017. Letzter Zugriff: 19.10.2017.

[MSD17] MSDN. Compiler Intrinsics. `https://msdn.microsoft.com/en-us/library/26td21ds`, Oktober 2017. Letzter Zugriff: 19.10.2017.

[Mü14] Marcel Müller. VC4ASM - macro assembler for Broadcom VideoCore IV aka Raspberry Pi GPU. `http://maazl.de/project/vc4asm/doc/index.html`, November 2014. letzter Zugriff: 01.05.2017.

[NVI17] NVIDIA. GeForce GTX 960 Specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960/specifications`, 2017. Letzter Zugriff: 04.05.2017.

[pd10] pocl developers. Portable Computing Language. `http://portablecl.org/`, 2010. Letzter Zugriff: 01.05.2017.

[Sug16] Yukimasa Sugizaki. VC4 blog. `http://imrc.noip.me/blog/vc4/`, März 2016. Letzter Zugriff: 20.07.2017.

[Upt12] Liz Upton. Eben gives a demo. `https://www.raspberrypi.org/blog/eben-gives-a-demo/`, Juni 2012. Letzter Zugriff: 19.08.2017.

[Upt14] Eben Upton. A birthday present from Broadcom. `https://www.raspberrypi.org/blog/a-birthday-present-from-broadcom/`, Februar 2014. Letzter Zugriff: 19.08.2017.

[Wik17a] Wikipedia. ARM Architecture. `https://en.wikipedia.org/wiki/ARM_architecture`, Mai 2017. Letzter Zugriff: 18.05.2017.

[Wik17b] Wikipedia. Basic block. `https://en.wikipedia.org/wiki/Basic_block`, April 2017. Letzter Zigruff: 17.06.2017.

[Wik17c] Wikipedia. Compiler. `https://en.wikipedia.org/wiki/Compiler`, April 2017. Letzter Zugriff: 12.05.2017.

[Wik17d] Wikipedia. Division algorithm. `https://en.wikipedia.org/wiki/Division_algorithm`, Juni 2017. Letzter Zugriff: 06.06.2017.

[Wik17e] Wikipedia. Free and open-source graphics device driver. `https://en.wikipedia.org/wiki/Free_and_open-source_graphics_device_driver`, April 2017. Letzter Zugriff: 02.05.2017.

[Wik17f] Wikipedia. Raspberry Pi. `https://en.wikipedia.org/wiki/Raspberry_Pi`, Mai 2017. Letzter Zugriff: 07.05.2017.

[Wik17g] Wikipedia. Static single assignment form. `https://en.wikipedia.org/wiki/Static_single_assignment_form`, Mai 2017. Letzter Zugriff: 21.05.2017.

A. Quellcode

```
1
2  /*
3   * Calculates the next 10 fibonacci numbers after start0,
      start1
4   * E.g. set start0 = start1 = 1 for the first 10 fibonacci
      numbers:
5   * 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
6   */
7  __kernel void fibonacci(const int start0, const int start1,
      __global int * out)
8  {
9    int fp = start0;
10   int fc = start1;
11   int tmp;
12
13   #pragma loop unroll
14   for(int i = 0; i < 10; i++)
15   {
16     tmp = fc + fp;
17     fp = fc;
18     fc = tmp;
19     out[i] = fc;
20   }
21 }
```

Listing A.1: OpenCL-Kernel zur Berechnung von Fibonaccizahlen

```
; ModuleID = '/home/daniel/workspace/VC4C/example/fibonacci.cl'
source_filename = "/home/daniel/workspace/VC4C/example/fibonacci.cl"
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"
target triple = "i386-unknown-linux-gnu"

; Function Attrs: norecurse nounwind
define void @fibonacci(i32, i32, i32* nocapture) local_unnamed_addr #9 !kernel_arg_addr_space !195 !kernel_arg_access_qual !196 !kernel_arg_type !197 !kernel_arg_base_type !197 !kernel_arg_type_qual !198 !kernel_arg_name !199 {
  %4 = add nsw i32 %0, %1
  store i32 %4, i32* %2, align 4, !tbaa !151
  %5 = add nsw i32 %4, %1
  %6 = getelementptr inbounds i32, i32* %2, i32 1
  store i32 %5, i32* %6, align 4, !tbaa !151
  %7 = add nsw i32 %4, %5
  %8 = getelementptr inbounds i32, i32* %2, i32 2
  store i32 %7, i32* %8, align 4, !tbaa !151
  %9 = add nsw i32 %5, %7
  %10 = getelementptr inbounds i32, i32* %2, i32 3
  store i32 %9, i32* %10, align 4, !tbaa !151
  %11 = add nsw i32 %7, %9
  %12 = getelementptr inbounds i32, i32* %2, i32 4
  store i32 %11, i32* %12, align 4, !tbaa !151
  %13 = add nsw i32 %9, %11
  %14 = getelementptr inbounds i32, i32* %2, i32 5
  store i32 %13, i32* %14, align 4, !tbaa !151
  %15 = add nsw i32 %11, %13
  %16 = getelementptr inbounds i32, i32* %2, i32 6
  store i32 %15, i32* %16, align 4, !tbaa !151
  %17 = add nsw i32 %13, %15
  %18 = getelementptr inbounds i32, i32* %2, i32 7
  store i32 %17, i32* %18, align 4, !tbaa !151
  %19 = add nsw i32 %15, %17
  %20 = getelementptr inbounds i32, i32* %2, i32 8
  store i32 %19, i32* %20, align 4, !tbaa !151
  %21 = add nsw i32 %17, %19
  %22 = getelementptr inbounds i32, i32* %2, i32 9
  store i32 %21, i32* %22, align 4, !tbaa !151
  ret void
}
```

Listing A.2: LLVM-IR Code des OpenCL-Kernels aus Listing A.1 (Ausschnitt)

```
119734787 65536 393230 39 0
2 Capability Addresses
2 Capability Kernel
5 ExtInstImport 1 "OpenCL.std"
3 MemoryModel 1 2
6 EntryPoint 6 6 "fibonacci"
3 Source 3 102000
4 Name 7 "start0"
4 Name 8 "start1"
3 Name 9 "out"
4 Decorate 9 FuncParamAttr 5
4 TypeInt 3 32 0
4 Constant 3 13 1
4 Constant 3 16 2
4 Constant 3 19 3
4 Constant 3 22 4
4 Constant 3 25 5
4 Constant 3 28 6
4 Constant 3 31 7
4 Constant 3 34 8
4 Constant 3 37 9
2 TypeVoid 2
4 TypePointer 4 5 3
6 TypeFunction 5 2 3 3 4

5 Function 2 6 0 5
3 FunctionParameter 3 7
3 FunctionParameter 3 8
3 FunctionParameter 4 9

2 Label 10
5 IAdd 3 11 7 8
5 Store 9 11 2 4
5 IAdd 3 12 11 8
5 InBoundsPtrAccessChain 4 14 9 13
5 Store 14 12 2 4
5 IAdd 3 15 11 12
5 InBoundsPtrAccessChain 4 17 9 16
5 Store 17 15 2 4
5 IAdd 3 18 12 15
5 InBoundsPtrAccessChain 4 20 9 19
5 Store 20 18 2 4
5 IAdd 3 21 15 18
5 InBoundsPtrAccessChain 4 23 9 22
5 Store 23 21 2 4
// [...] see next attachment
```

Listing A.3: SPIR-V textuelle Repräsentation des OpenCL-Kernels aus Listing A.1

```
// [...] see previous attachment
5 IAdd 3 24 18 21
5 InBoundsPtrAccessChain 4 26 9 25
5 Store 26 24 2 4
5 IAdd 3 27 21 24
5 InBoundsPtrAccessChain 4 29 9 28
5 Store 29 27 2 4
5 IAdd 3 30 24 27
5 InBoundsPtrAccessChain 4 32 9 31
5 Store 32 30 2 4
5 IAdd 3 33 27 30
5 InBoundsPtrAccessChain 4 35 9 34
5 Store 35 33 2 4
5 IAdd 3 36 30 33
5 InBoundsPtrAccessChain 4 38 9 37
5 Store 38 36 2 4
1 Return

1 FunctionEnd
```

Listing A.4: SPIR-V textuelle Repräsentation des OpenCL-Kernels aus Listing A.1 (Fortsetzung)

```
0xdeadbeaf, 0xdeadbeaf,
// Kernel 'fibonacci', offset 16, with following parameters:
    i32 start0 (4 B, 1 items), i32 start1 (4 B, 1 items),
   __global out i32* out (4 B, 1 items)
0x00350010, 0x00030009,
0x00000000, 0x00000000,
0x6f626966, 0x6363616e,
0x00000069, 0x00000000,
0x00060104, 0x00100003,
0x72617473, 0x00003074,
0x00323369, 0x00000000,
0x00060104, 0x00100003,
0x72617473, 0x00003174,
0x00323369, 0x00000000,
0x00030104, 0x12200004,
0x0074756f, 0x00000000,
0x2a323369, 0x00000000,
0,0,
0,0,
; [...] see next attachment
```

Listing A.5: Hex-Ausgabe des kompilierten Beispiels zur Berechnung von Fibonaccizahlen aus Listing A.1 (Metadaten)

```
; [...] see previous attachment
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x100209e7, //or -, unif, unif
0x15827d80, 0x10020027, //or ra0, unif, unif
0x15827d80, 0x10020827, //or r0, unif, unif
0x15827d80, 0x100208a7, //or r2, unif, unif
0x0c027c00, 0x10020867, //add r1, ra0, r0
0x15ce7d80, 0x100209e7, //or -, mutex_acq, mutex_acq
0x85014000, 0xe0021c67, //ldi vpw_setup, 2231451648
0xc0010000, 0xe0021c67, //ldi vpw_setup, 3221291008
0x00001a00, 0xe0021c67, //ldi vpw_setup, 6656
0x159e7240, 0x10020c27, //or vpm, r1, r1
0x0c9e7200, 0x10020827, //add r0, r1, r0
0x159e7000, 0x10020c27, //or vpm, r0, r0
0x0c9e7200, 0x10020867, //add r1, r1, r0
0x159e7240, 0x10020c27, //or vpm, r1, r1
0x0c9e7040, 0x10020827, //add r0, r0, r1
0x159e7000, 0x10020c27, //or vpm, r0, r0
0x0c9e7200, 0x10020867, //add r1, r1, r0
0x159e7240, 0x10020c27, //or vpm, r1, r1
0x0c9e7040, 0x10020827, //add r0, r0, r1
0x159e7000, 0x10020c27, //or vpm, r0, r0
0x0c9e7200, 0x10020867, //add r1, r1, r0
0x159e7240, 0x10020c27, //or vpm, r1, r1
0x0c9e7040, 0x10020827, //add r0, r0, r1
0x159e7000, 0x10020c27, //or vpm, r0, r0
0x0c9e7200, 0x10020867, //add r1, r1, r0
0x159e7240, 0x10020c27, //or vpm, r1, r1
0x0c9e7040, 0x10020827, //add r0, r0, r1
0x159e7000, 0x10020c27, //or vpm, r0, r0
0x159e7480, 0x10021ca7, //or vpw_addr, r2, r2
0x159f2fc0, 0x100209e7, //or -, vpw_wait, vpw_wait
0x159c1fc0, 0xd0020ce7, //or mutex_rel, 1 (1), 1 (1)
; [...] see next attachment
```

Listing A.6: Hex-Ausgabe des kompilierten Beispiels zur Berechnung von Fibonaccizahlen aus Listing A.1 (Funktionsrumpf)

```
; [...] see previous attachment
0x15827d80, 0x10020827, //or r0, unif, unif
0x159a7c00, 0x100229e7, //or.setf -, elem_num, r0
0xfffffe78, 0xf01809e7, //brr.ifallzc (pc+4) + -49
0x009e7000, 0x100009e7, //nop.never
0x009e7000, 0x100009e7, //nop.never
0x009e7000, 0x100009e7, //nop.never
0x179e6e00, 0x100209a7, //not irq, qpu_num
0x009e7000, 0x300009e7, //nop.thrend.never
0x009e7000, 0x100009e7, //nop.never
0x009e7000, 0x100009e7, //nop.never
```

Listing A.7: Hex-Ausgabe des kompilierten Beispiels zur Berechnung von Fibonaccizahlen aus Listing A.1 (Stopsegment)

```
/*
  Taylor series (https://en.wikipedia.org/wiki/
   Natural_logarithm#Derivative.2C_Taylor_series)

  Has a maximum error of 4.1 * 10^7 (at x ~ 1.329*10^36 (x =
    2^120)) and an maximum allowed relative error of 4.7 *
   10^-7
 */
COMPLEX_1(float, log, float, val,
{
  /* log(M * 2^E) = log(M) + E log(2) */
  result_t logE = vc4cl_itof(ilogb(val)) * M_LN2_F;

  /* extract mantissa, set exponent to 2^0 -> 127, keep sign
    */
  result_t M = vc4cl_bitcast_float(
    (vc4cl_bitcast_uint(val) & 0xFFFFFFU) | 0x3FC00000U);
  M = copysign(M, val);
  /* move from range [1, 2[ to [0, 1[ */
  /* M/2 instead of M - 1 is on purpose! */
  M /= 2.0f;

  /* approximate log(M) in [0, 1[ via Taylor-series with 17
   steps */
  result_t logM = 0.0f;
  result_t MpowN = M - 1.0f;
  for(uint n = 1; n < 18; ++n)
  {
    /* ((-1)^(n+1))/n * (x-1)^n */
    logM += (((n % 2 == 1) ? +1.0f : -1.0f) /
      (result_t)n) * MpowN;
    MpowN *= M - 1.0f;
  }

  /* fix offset from log(M/2) to log(M) with adding log(2)
   */
  return logE + logM + M_LN2_F;
})
```

Listing A.8: Implementierung der `log`-Funktion zur Berechnung des natürlichen Logarithmus