Karlsruher Institut für Technologie

Institut für Technische Informatik
Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung
Prof. Dr. rer. nat. Wolfgang Karl

Automatische OpenCL-Code-Analyse zur Bestimmung von Speicherzugriffsmustern

Bachelorarbeit
von

Moritz Lüdecke

an der Fakultät für Informatik

Tag der Anmeldung: 11. Februar 2014
Tag der Fertigstellung: 10. Juni 2014

Aufgabensteller:
Prof. Dr. rer. nat. Wolfgang Karl

Betreuer:
Dipl.-Inform. Mario Kicherer

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 10.06.2014

Moritz Lüdecke

Zusammenfassung

OpenCL ist ein einheitliches Programmiermodell, mit dessen Hilfe Aufgaben auf unterschiedlichen Architekturen wie CPU, GPU oder FPGA ausgeführt werden können. Der OpenCL-Quellcode wird hierzu mit der Anwendung mitgeliefert und zur Laufzeit mittels Just-In-Time-Kompilierung für eine vorhandene Recheneinheit übersetzt.

Zur Reduzierung der Rechenzeit sollen nun aber mehrere Einheiten gleichzeitig zur Berechnung genutzt werden. Aufgrund der geteilten Adressräume müssen dafür aber die benötigten Daten pro Arbeitspaket bestimmt werden, um eine korrekte Aufteilung der Gesamtlast zu ermöglichen und unnötig teure Speichertransfers zu vermeiden.

Um dieses Vorgehen auch in OpenCL umzusetzen, muss der Speicher so aufgeteilt werden, dass lediglich die zum Ausführen des Quellcodes benötigten Daten auf die OpenCL-Geräte transferiert werden. In dieser Arbeit wird auf die Problematik der Datenabhängigkeiten im Programmcode sowie auf die Art und Weise der Feststellung von Schreib- und Lesezugriffe hinsichtlich Daten eingegangen. Im ersten Schritt wird am Quelltext Mithilfe des LLVM-Frontends Clang eine statische Codeanalyse durchgeführt. Das Resultat der Codeanalyse wird anschließend zur Bildung des Speicherzugriffsmusters verwendet. Diese Arbeitsweise wird letztlich in einer Bibliothek zusammengefasst.

In dieser Arbeit werden die grundlegenden Mechanismen der Analyse entwickelt und anhand des OpenCL-Codes der Rodinia Benchmark verifiziert. Damit ein größerer Teil an Benchmarks unterstützt werden kann, müssen während der Analyse mehr Codekonstrukte berücksichtigt werden.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

1.1 Motivation

Durch die voranschreitende Heterogenisierung der Rechnersysteme ist es sinnvoll, eine gemeinsame Basis einzusetzen, die insbesondere hersteller- und architekturübergreifend ist. Dadurch kann sich der Programmierer auf ein einheitliches Architekturkonzept konzentrieren und muss nicht mehr wie bisher verschiedene Architekturen im Quelltext berücksichtigen. Mit OpenCL hat sich in den vergangen Jahren immer mehr eine solche Schnittstelle zur einheitlichen Programmierung von heterogenen Systemen durchgesetzt. Der Programmcode wird in Host- und Kernelcode aufgeteilt. Der Kernelcode kann auf beliebig vielen Systemen und Hardwarekomponenten verschiedenster Art ausgeführt werden. Der Hostcode hingegen wird lediglich auf einem System bzw. Gerät, meist der CPU, ausgeführt. Dieser kommuniziert mit den verschiedenen Hardwarekomponenten, auf denen der Kernelcode ausgeführt wird. Durch das stark parallelisierbare Konzept von OpenCL sind gewisse Rahmenbedingungen unabdingbar. So wird der Kernelcode in einem C-Dialekt geschrieben und beispielsweise auf Grafikkarten mehrmals parallel ausgeführt.

Heutzutage findet sich in den meisten Computern eine Grafikkarte wieder, welche zu vielen Zwecken genutzt werden kann und insbesondere aufgrund der Architektur ihrer GPU effizient in parallelem Rechnen sind. Dadurch sind Grafikkarten längst nicht mehr ausschließlich für Computerspiele interessant, sondern eignen sich hervorragend für verschiedenste Rechenprobleme und können somit ganz im Sinne des Gedankens der GPGPU die CPU entlasten. Daneben können auch andere Architekturen wie FPGAs herangezogen werden.

Ohne einen zusätzlichen Einsatz des Programmierers wird der Kernelcode jedoch nur auf einer Hardwarekomponente ausgeführt, d.h. entweder auf der CPU oder auf der GPU. Soll dieser Code auf mehreren Hardwarekomponenten ausgeführt werden, so ist von außen nicht ersichtlich, welche Daten an den verschiedenen Komponenten vom Kernel zur Verarbeitung benötigt werden und welche aufgrund der Arbeitsaufteilung ungenutzt bleiben. Im schlechtesten Fall werden die Daten vollständig von der Hardwarekomponente, auf der der Hostcode ausgeführt wird, an jede Hardwarekomponente, die den Kernelcode ausführt, hin kopiert und nach der Berechnung wieder zurück zum Hostgerät kopiert. Kopiervorgänge sind in Rechnersystemen jedoch eine der teuersten Operationen und sollten wenn möglich vermieden oder zumindest minimiert werden.

Folglich gilt es im ersten Schritt, die Datenabhängigkeiten eines OpenCL-Kernels zu bestimmen und somit auch dessen Speicherzugriffmuster. Anschließend können mit diesem Ergebnisgewinn die zu kopierenden Daten auf das Minimum reduziert werden. Es werden ausschließlich die von den Hardwarekomponenten zu verarbeitenden Daten kopiert.

1.2 Aufgabenstellung

Das Ziel dieser Arbeit ist, das Speicherzugriffsmuster anhand des zur Laufzeit vorliegenden OpenCL-Kernels zu bestimmen. Dabei sollen verschiedene Herangehensweisen in Betracht gezogen werden und schließlich diejenigen mit den vorteilhaftesten Kompromissen weiter verfolgt werden. Als Basis kann die LLVM-Compiler-Infrastruktur zur Analyse des Kernelcodes dienen. Des Weiteren werden über eine statische Codeanalyse die verschiedenen Abhängigkeiten innerhalb der OpenCL-Kernelfunktionen detektiert und mittels diesen das Speicherzugriffsmuster der einzelnen Parameter der Kernelfunktionen bestimmt. Mit dem Zugriffsmuster der Funktionsparameter bildet sich schließlich das Speicherzugriffsmuster der Kernelfunktionen und im Weiteren auch des OpenCL-Kernels.

Das Speicherzugriffsmuster kann im Nachhinein durch Änderungen der Rahmenbedingungen des OpenCL-Kernels, wie beispielsweise der Work-Group-Größe, beeinflusst und neu bestimmt werden. Hierzu ist keine weitere Codeanalyse von Nöten, womit zusätzliche Rechenarbeit entfällt. Jeder Funktionsparameter besitzt außerdem jeweils ein Speicherzugriffsmuster für Lese- und Schreibzugriffe.

1.3 Struktur der Arbeit

Zu Beginn werden die Grundlagen zur Hardwarearchitektur, OpenCL und LLVM, sowie dessen Frontend Clang skizziert. Nachdem veröffentlichte Arbeiten mit verwandtem Schwerpunkt kurz dargelegt und mit der in dem Inhalt der vorliegenden Arbeit verglichen wird, folgt der Kern dieser Arbeit: der konkreten Umsetzung der Aufgabenstellung. Darin werden die nötigen Schritte von der Codeanalyse über die Datenabhängigkeiten der einzelnen Variablen und Arrays hin zur Bildung des Speicherzugriffmusters skizziert und erläutert. Zum Schluss wird die Arbeit evaluiert, noch einmal zusammengefasst und ein Ausblick auf zukünftige Arbeiten hergestellt.

2 Grundlagen

2.1 Aufbau und Architektur der Hardware

Jeder Computer ist mit einer CPU bestückt, die die restlichen Hardwarekomponenten verwaltet und Befehle des Betriebssystems ausführt. Eine CPU besitzt mindestens einen Prozessorkern, der wiederum für die eigentlichen Rechenoperationen zuständig ist. Daher wird der Prozessorkern auch als Recheneinheit betrachtet.

Eine serielle Ausführung von Instruktionen garantiert ein konflikt- und abhängigkeitsfreies Arbeiten, jedoch wird durch physikalische Grenzen schnell die Maximalleistung erreicht. Um dieses Problem zu lösen, wurde die CPU um mehrere Recheneinheiten erweitert. Zwar können damit Instruktionen schneller ausgeführt werden, dies bringt allerdings den Nachteil mit sich, dass die Instruktionen nicht voneinander abhängig sein dürfen. Ansonsten bremsen sich die Recheneinheiten gegenseitig aus, indem die eine Recheneinheit auf die benötigte Abhängigkeit der anderen warten muss.

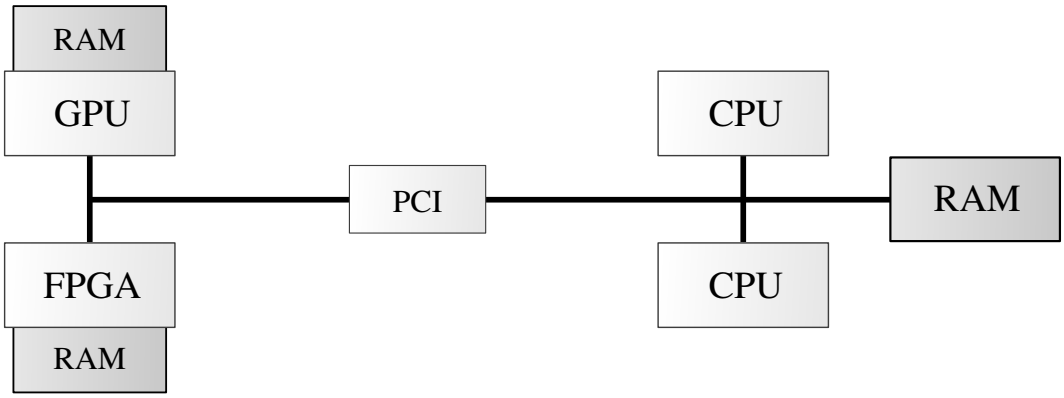

Abbildung 2.1: Vereinfachtes Modell eines Computersystems

Eine weitere Entwicklung ist die Auslagerung der Aufgaben weg von der CPU. So wird beispielsweise die Gleitkommaeinheit der CPU für die Berechnung von Gleitkommaoperationen herangezogen. Solche Einheiten werden auch Koprozessor genannt und dienen der Entlastung der CPU. Zugleich können diese aufgrund ihrer Architektur spezielle Operationen schneller als die CPU ausführen. Aber nicht nur Koprozessoren können die CPU

entlasten, sondern auch ganze Hardwarekomponenten wie die Grafikkarte. Diese besitzt sehr viele Recheneinheiten und stellt dadurch eine stark parallelisierte Rechnerkomponente dar. Infolgedessen müssen Daten von der CPU zur Hardwarekomponente transferiert und anschließend nach der Berechnung wieder zurück kopiert werden. Dabei muss beachtet werden, dass der Kopiervorgang eine teure Operation ist und sollte dahingehend wenn möglich vermieden oder zumindest minimiert werden. Abbildung 2.1 veranschaulicht dies.

2.2 Einführung in OpenCL

Ziel des offene Industriestandard OpenCL (***Open C****omputing* ***L****anguage*) war die Möglichkeit, den Code auf unterschiedlichen Systemkomponenten mit einer ebenfalls differenzierten Architektur ausführen zu können. Als prominentes Beispiel stehen hierfür CPUs und GPUs, jedoch können auch andere Hardwarekomponenten wie FPGAs diesen Code ausführen.

Initiator von OpenCL war die Firma Apple, die den Entwurf später an die Khronos Group [1], ein Industriekonsortium, eingereicht hatte. Daraufhin wurde OpenCL im Dezember 2008 als offener Standard deklariert und besitzt mittlerweile die Versionsnummer 2.0.

Zu Beginn wird der Code auf einem Host, zumeist einer CPU, ausgeführt. Dieser verwaltet die einzelnen OpenCL-Geräte, die Devices genannt werden. Auf ihnen wird der eigentliche OpenCL-Code, der Kernelcode, ausgeführt. Devices können also alle OpenCL-fähigen Computerkomponenten sein. Selbst die CPU, auf der der Host-Code ausgeführt wird, kann anschließend als Device zum Ausführen des Kernels herangezogen werden.

Abbildung 2.2 veranschaulicht die hierarchische Gliederung der OpenCL-Devices, die als ausführende Geräte *Compute Device* genannt werden. Diese bekommen ihre Instruktionen vom Host-Device und bestehen aus ein oder mehrere Recheneinheiten, auch CU (***C****ompute* ***U****nit*) genannt. Beispielsweise kann ein Kern eines Mehrkernprozessor eine Recheneinheit darstellen. Die CU fasst abermals ein oder mehrere ausführende Elemente, PE (***P****rocessing* ***E****lement*) genannt, zusammen. Im Hostcode wird die Anzahl und die Anordnung der Work-Items bestimmt, die weiter zu Work-Groups formiert werden.

Der OpenCL-Kernel wird in der Programmiersprache OpenCL C geschrieben, die ein C-Dialekt darstellt und auf ISO C99 [3] aufbaut. Eine Kernelfunktion wird mit dem reservierten Wort `__kernel` eingeleitet, gefolgt von dem in C gewohnten Rückgabetyp, Funktionsname und Funktionsparametern. In Listing 2.1 wird deutlich, dass die beiden Parameter `input` und `output` Pointer sind und in diesem Zusammenhang beide jeweils ein Array vom Typ Gleitkommazahl darstellen. Ferner ist `get_global_id` eine OpenCL-Funktion, die basierend auf den Rahmenbedingungen des OpenCL-Kernels

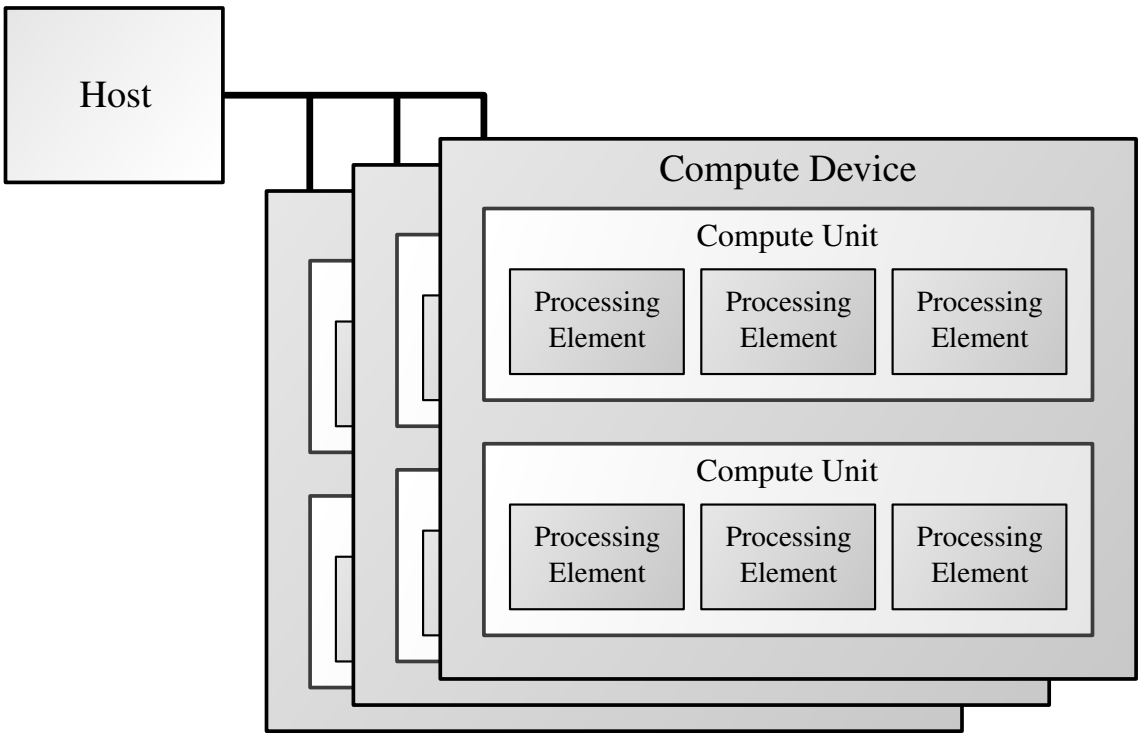

Abbildung 2.2: Ausführungsmodell von OpenCL, *Quelle:* [2]

und dem angegebenen Funktionsparameter einen Indexwert zurück gibt, mit dessen Hilfe schließlich auf die beiden Arrays zugegriffen wird. Dieser Beispielcode wird in jedem einzelnen Work-Item ausgeführt, sodass jedes Arrayelement idealerweise in jeweils einem PE gelesen, berechnet und beschrieben wird. Nach der Spezifikation kann ein Work-Item auch auf mehreren PEs ausgeführt werden, da es sich bei der PE um einen virtuellen Skalarprozessor handelt [4, S. 18].

```
__kernel void square(__global float* input,
                     __global float* output,
                     const unsigned int count) {
        int i = get_global_id(0);
        if (i < count) {
                output[i] = input[i] * input[i];
        }
}
```

Listing 2.1: Quadratfunktion im OpenCL-Kernelcode

Es sind in OpenCL C gewisse Einschränkungen gegenüber C anzumerken. So dürfen unter anderem Arrays keine variable Größe besitzen; Pointer auf Funktionen sind ebenso unzulässig wie Rekursionen.

Ein weiterer erwähnenswerter, jedoch für diese Arbeit nebensächlicher Aspekt, ist das Speichermodell von OpenCL. Zur Verdeutlichung wird Abbildung 2.3 herangezogen. Der Speicher teilt sich in einen Host- und einen Gerätespeicher. Letzterer ist nochmals in vier Speicherarten aufgeteilt.

- **Hostspeicher** (*host memory*)
  Dieser ist lediglich für den Host sichtbar. In der Regel ist dies der Arbeitsspeicher des Systems.
- **Gerätespeicher** (*device memory*)
  Er steht dem Kernel während der Laufzeit zur Verfügung.
  - **Globaler Speicher** (*global memory*)
    Jede Kernelinstanz hat Lese- und Schreibzugriff auf diesen Arbeitsspeicher.
  - **Konstanter Speicher** (*constant memory*)
    Wird wie der globale Speicher behandelt, ist jedoch nur lesbar und kann somit nicht beschrieben werden.
  - **Lokaler Speicher** (*local memory*)
    Lediglich die Work-Group hat auf diesen Speicher Zugriff. Folglich darf eine Kernelinstanz einer Work-Group nicht auf den lokalen Speicher anderer Work-Groups zugreifen.
  - **Privater Speicher** (*private memory*)
    Jeder Kernelinstanz ist ein eigener Speicher zugeteilt, auf den ausschließlich diese Zugriff besitzt.

Was für eine Speicherart zur Laufzeit letztendlich benutzt werden soll, kann dem Compiler mit den Attributen `__global`, `__constant`, `__local` und `__private` mitgeteilt werden.

Die GPU als Rechenhilfe für die CPU

OpenCL bietet hier eine komfortable Lösung und erlaubt es, Rechenaufgaben an die GPU, sofern diese Schnittstelle unterstützt wird, zu verteilen. Dabei müssen zwei Aspekte genauer betrachtet werden. Zum einen sollte nie derselbe Code mit den gleichen Randbedingungen zwei Mal ausgerechnet werden. In diesem Fall beispielsweise einmal auf einem Prozessorkern der CPU und einmal auf einem Shader der GPU. Abstrakt betrachtet sollte also ein Work-Item immer nur auf einer CU ausgeführt werden. Dies hat zur Folge, dass

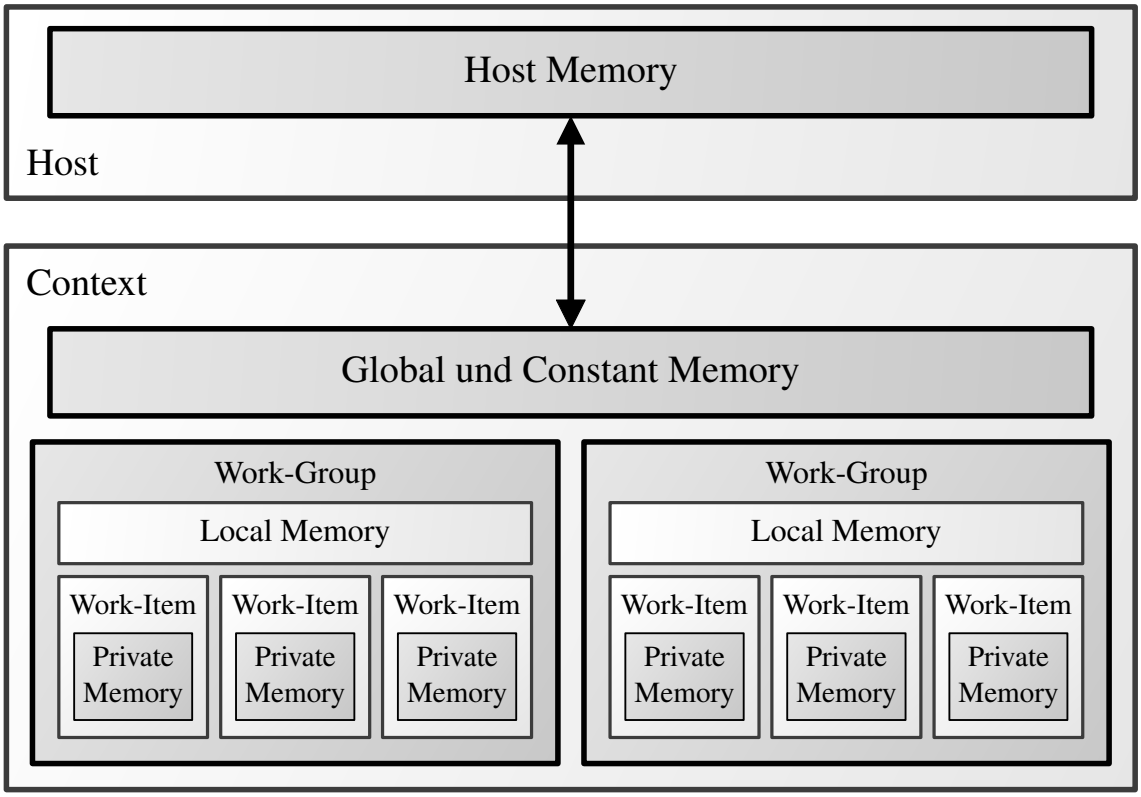

Abbildung 2.3: OpenCL-Speichermodell, *Quelle:* [5]

eventuell Daten von der GPU kopiert werden, die während der Ausführung des Codes nicht verarbeitet wurden, da diese lediglich auf der CPU verarbeitet werden. Zum anderen, im ersten Aspekt begründet, muss vermieden werden, dass unnötig Daten kopiert werden.

2.3 Der Compiler LLVM

LLVM [6], früher auch unter **L**ow **L**evel **V**irtual **M**achine bekannt, ist ein modular aufgebauter Compiler, der seine Wurzeln als ein Forschungsprojekt an der Universität von Illinois hat. Das LLVM-Projekt ist mittlerweile nicht mehr ausschließlich eine „Low Level Virtual Machine“, sondern wurde nach und nach zu einem Gesamtprojekt, das mehrere Compilerpratiken als Unterprojekte vereint.

So gibt es mittlerweile den „LLVM Core“, der Codeoptimierung und Codeübersetzung für die gängigen CPU-Architekturen vollzieht. Dazu wird die eigens entwickelte Zwischensprache LLVM IR verwendet, die im Unterkapitel 2.3.1 näher behandelt wird. Daneben gibt es noch das LLVM-Frontend Clang. Hierauf wird im nachfolgenden Kapi-

tel 2.4 eingegangen. Neben vielen anderen nennenswerten Besonderheiten, die in dieser Arbeit jedoch keine größere Rolle spielen, besitzt das LLVM-Projekt mit LLDB sogar einen nativen Debugger.

Darüber hinaus benutzen viele weitere Projekte die Infrastruktur von LLVM, um den Sourcecode für die entsprechenden Zielplattformen zu kompilieren. Beispielsweise wird bei CUDA der Sourcecode in LLVM IR übersetzt, um diesen anschließend für verschiedene Zielplattformen kompilieren zu können [7]. So ist man mittlerweile bei CUDA nicht mehr an Nvidia-GPUs gebunden, sondern kann den CUDA-Code auch auf CPUs mit einer x86-Architektur lauffähig machen. Einen groben Überblick, wie ein Code in LLVM intern verarbeitet bzw. kompiliert wird, wird in Abbildung 2.4 gegeben. Zudem kann der Prozess um weitere Frontends und Backends ergänzt werden.

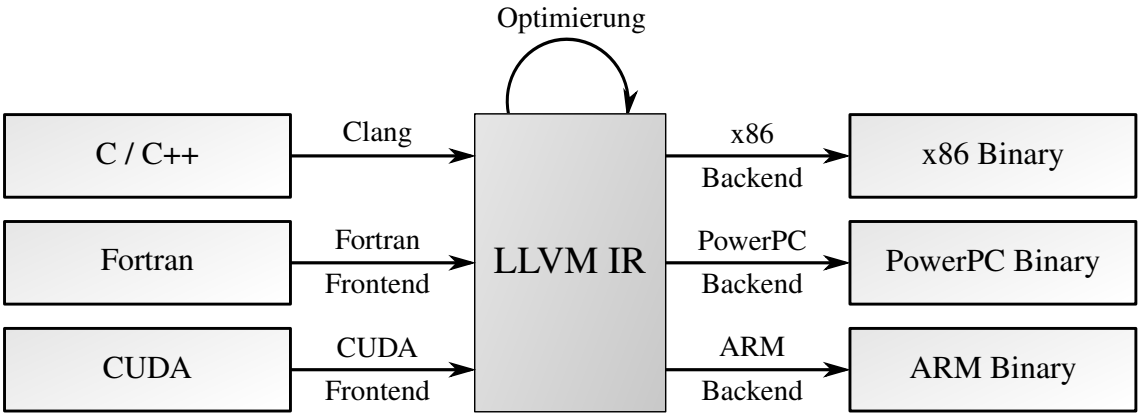

Abbildung 2.4: Compiliervorgang in LLVM

2.3.1 LLVM Intermediate Representation

Die LLVM Intermediate Representation, kurz LLVM IR, stellt einen Bytecode dar, also eine assemblerähnliche Zwischensprache, die jedoch maschinenabhängig ist. Mit diesem Code wird in LLVM hauptsächlich gearbeitet. Als Beispiel wird in Listing 2.2 ein hello-world-Code in C herangezogen. Mithilfe des C-Frontends Clang kann nun via `clang file.c -S -emit-llvm -o -` ein nicht optimierter Bytecode in LLVM IR erstellt werden, der in Listing 2.3 zu sehen ist.

Ebenfalls ist es möglich in LLVM einen maschinenunabhängigen Bytecode zu generieren. Dieser nennt sich Bitfield, mit dem Techniken wie ein Just-In-Time-Compiler realisiert werden kann.

In LLVM wird der IR-Code über sogenannte Passes verarbeitet. Mit ihnen wird der Code im Wesentlichen analysiert, optimiert und transformiert. Dies geschieht in mehreren

```
#include <stdio.h>

int main(int argc, char **argv) {
        printf("hello world\n");
}
```

Listing 2.2: LLVM-IR-Beispiel: C-Code

Zyklen, sodass ein Code in seiner Lebenszeit in der IR mehrmals einen Pass durchlaufen kann. Da Passes aufeinander aufbauen, sind diese auch voneinander abhängig, weswegen die Ausführungsreihenfolge für das Ergebnis des einzelnen Passes wichtig ist. Schließlich kann ein Pass a, der vor Pass b ausgeführt ist, den Code so verändern, dass dieser Code gewisse Informationen nicht mehr enthält, die Pass b benötigt.

2.4 LLVM's C-Fronted Clang

Das Frontend [8] für C, C++, Objective C und Objective C++ ist ein primäres Unterprojekt von LLVM. Die Kommandozeilenprogramme `clang` und `clang++` verhalten sich größtenteils wie die von GCC [9]. So kann meist die Syntax von GCC zum Kompilieren eines Codes für Clang übernommen werden, ohne die Parameter anpassen zu müssen.

Darüber hinaus kann Clang auch über eine API Schnittstelle auf drei verschiedene Arten benutzt werden, auf die später noch näher eingegangen wird. Zudem wird der Code in einem AST-Format von Clang festgehalten, der über die gegebene API weiter verarbeitet werden kann. In diesem Punkt ist Clang gegenüber GCC fortschritlich und es existieren bereits heute viele Anwendungen, die von dieser API Gebrauch machen. Als ein einfaches Szenario wird hier auf die sehr aussagekräftigen Fehlernachrichten von Clang verwiesen, die in vielen Fällen nützlicher als die des GCC-Compilers sind. Aber auch die Fehlererkennung von Clang findet mittlerweile in einigen Programmen ihren Einsatz.

2.4.1 Clang AST

Clang AST [10] (***A**bstract **S**yntax **T**ree*) ist eines der Herzstücke von Clang bzw. dem LLVM-Projekt. Über ihn geschieht die statische Codeanalyse und dank seiner umfangreichen Funktionalität kann sehr fein zwischen unterschiedlichen Zuständen unterschieden werden [11]. Eine Repräsentation des abstrakten Syntaxbaums vom OpenCL-Code weiter oben in Listing 2.1 kann in Listing 2.4 begutachtet werden.

```
; ModuleID = 'main.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-
    f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8**, align 8
  store i32 %argc, i32* %1, align 4
  store i8** %argv, i8*** %2, align 8
  %3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([13 x i8]* @.str, i32 0,
       i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim
    "="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math
    "="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float
    "="false" }
attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-
    pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-
    protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

Listing 2.3: LLVM-IR-Beispiel: Bytecode

2.4.2 Clang und seine Programmierschnittstellen

Clang kann auf unterschiedliche Arten benutzt werden. Nicht nur ausschließlich zum Kompilieren von Code, sondern auch zur Codeanalyse. Im Laufe der Entwicklung von Clang wurde schnell klar, dass die Community das LLVM-Frontend vielseitig verwendet. Nachdem anfangs ein Hauptmerkmal auf der äußerst präzisen Codediagnose gelegen hat, die demzufolge auch oft in IDEs eingebaut wurde, wird Clang mittlerweile auch für routinierte Tätigkeiten verwendet, wie beispielsweise der Aktualisierung alter API-Funktionen einer externen Bibliothek im eigenen Programmcode.

Durch diesen Prozess haben sich im Wesentlichen drei Schnittstellen heraus entwickelt, die überwiegend eine spezielle Art von Aufgaben lösen sollen, sich jedoch in der Praxis oft auch überschneiden [12].

```
TranslationUnitDecl 0x1a1ad30 <<invalid sloc>>
|-TypedefDecl 0x1a1b330 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x1a1b390 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x1a1b6e0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x1a63ee0 <test.cl:1:1, line:8:1> square 'void (__global float *, __global float *, const unsigned int)
      '
| |-ParmVarDecl 0x1a1b7b0 <line:1:22, col:38> input '__global float *'
| |-ParmVarDecl 0x1a1b840 <line:2:22, col:38> output '__global float *'
| |-ParmVarDecl 0x1a1b8b0 <line:3:22, col:41> count 'const unsigned int'
| |-CompoundStmt 0x1a645a0 <col:48, line:8:1>
| | |-DeclStmt 0x1a64200 <line:4:2, col:26>
| | | `-VarDecl 0x1a63ff0 <col:2, col:25> i 'int'
| | |   `-CallExpr 0x1a641d0 <col:10, col:25> 'int'
| | |     |-ImplicitCastExpr 0x1a641b8 <col:10> 'int (*)()' <FunctionToPointerDecay>
| | |     | `-DeclRefExpr 0x1a64140 <col:10> 'int ()' Function 0x1a640a0 'get_global_id' 'int ()'
| | |     `-IntegerLiteral 0x1a64168 <col:24> 'int' 0
| | `-IfStmt 0x1a64570 <line:5:2, line:7:2>
| |   |-<<<NULL>>>
| |   |-BinaryOperator 0x1a642b0 <line:5:6, col:10> 'int' '<'
| |   | |-ImplicitCastExpr 0x1a64298 <col:6> 'unsigned int' <IntegralCast>
| |   | | `-ImplicitCastExpr 0x1a64268 <col:6> 'int' <LValueToRValue>
| |   | |   `-DeclRefExpr 0x1a64218 <col:6> 'int' lvalue Var 0x1a63ff0 'i' 'int'
| |   | `-ImplicitCastExpr 0x1a64280 <col:10> 'unsigned int' <LValueToRValue>
| |   |   `-DeclRefExpr 0x1a64240 <col:10> 'const unsigned int' lvalue ParmVar 0x1a1b8b0 'count' 'const unsigned int'
| |   |-CompoundStmt 0x1a64550 <col:17, line:7:2>
| |   | `-BinaryOperator 0x1a64528 <line:6:3, col:33> 'float' '='
| |   |   |-ArraySubscriptExpr 0x1a64358 <col:3, col:11> '__global float' lvalue
| |   |   | |-ImplicitCastExpr 0x1a64328 <col:3> '__global float *' <LValueToRValue>
| |   |   | | `-DeclRefExpr 0x1a642d8 <col:3> '__global float *' lvalue ParmVar 0x1a1b840 'output' '__global float *'
| |   |   | `-ImplicitCastExpr 0x1a64340 <col:10> 'int' <LValueToRValue>
| |   |   |   `-DeclRefExpr 0x1a64300 <col:10> 'int' lvalue Var 0x1a63ff0 'i' 'int'
| |   |   `-BinaryOperator 0x1a64500 <col:15, col:33> 'float' '*'
| |   |     |-ImplicitCastExpr 0x1a644d0 <col:15, col:22> 'float' <LValueToRValue>
| |   |     | `-ArraySubscriptExpr 0x1a64400 <col:15, col:22> '__global float' lvalue
| |   |     |   |-ImplicitCastExpr 0x1a643d0 <col:15> '__global float *' <LValueToRValue>
| |   |     |   | `-DeclRefExpr 0x1a64380 <col:15> '__global float *' lvalue ParmVar 0x1a1b7b0 'input' '__global float
      *'
| |   |     |   `-ImplicitCastExpr 0x1a643e8 <col:21> 'int' <LValueToRValue>
| |   |     |     `-DeclRefExpr 0x1a643a8 <col:21> 'int' lvalue Var 0x1a63ff0 'i' 'int'
| |   |     `-ImplicitCastExpr 0x1a644e8 <col:26, col:33> 'float' <LValueToRValue>
| |   |       `-ArraySubscriptExpr 0x1a644a8 <col:26, col:33> '__global float' lvalue
| |   |         |-ImplicitCastExpr 0x1a64478 <col:26> '__global float *' <LValueToRValue>
| |   |         | `-DeclRefExpr 0x1a64428 <col:26> '__global float *' lvalue ParmVar 0x1a1b7b0 'input' '__global float
      *'
| |   |         `-ImplicitCastExpr 0x1a64490 <col:32> 'int' <LValueToRValue>
| |   |           `-DeclRefExpr 0x1a64450 <col:32> 'int' lvalue Var 0x1a63ff0 'i' 'int'
| |   `-<<<NULL>>>
| `-OpenCLKernelAttr 0x1a63fa0 <line:1:1>
`-FunctionDecl 0x1a640a0 <<invalid sloc>> get_global_id 'int ()'
```

Listing 2.4: AST der Quadratfunktion im OpenCL-Kernelcode 2.1

LibClang

LibClang [13] ist eine C-Schnittstelle zu Clang, die für den Einstieg gut geeignet ist. Man muss für Clang Clang AST kein Experte sein, um die Vorteile von diesem nutzen zu können. Viele Funktionen sind bereits als Methoden in LibClang hinterlegt. Infolgedessen ist kein voller Zugriff auf den Clang AST nötig. Ein weiterer Vorteil von LibClang ist die stabile API, die sich im Laufe der Entwicklung von Clang wenig verändert hat. Damit soll sichergestellt werden, dass der Großteil des Codes anderer Projekte auch weiterhin mit neueren Clangversionen lauffähig bleibt.

Clang Plugins

Im Gegensatz zu LibClang ist mit Clang Plugins [14] ein voller Zugriff auf den Clang AST möglich, wodurch zusätzliche Aktionen darauf ausgeführt werden können. Indem Plugins dynamische Bibliotheken darstellen, können diese zur Laufzeit des Compilers geladen werden und Aktionen durchführen, wie beispielsweise eine Warnung bei einer falsch benutzten Syntax oder eine Anmerkung bei einer nicht beachteten Code Convention im Terminal.

LibTooling

Das C++-Interface LibTooling ist relativ neu und wurde geschaffen, um als Entwickler schnell ein Standalone-Programm schreiben zu können [15]. So muss sich dieser zukünftig nicht mehr um die Infrastruktur des Terminalprogramms wie das Parsen von Parametern kümmern, sondern überlässt diese Arbeit, wie in Listing 2.5 zu sehen, LibTooling.

Mit wenig Aufwand kann so ein kleines Standalone-Programm geschrieben werden. Deshalb müssen auf dem Zielrechner keine LLVM- oder Clang-Bibliotheken installiert werden. Dies bringt aber auch einen Nachteil mit sich: Der Quelltext muss zusammen mit dem von LLVM und Clang kompiliert werden [16]. Des Weiteren muss eine JSON Compilation Database [17] anlegt werden, in der vermerkt ist, wie die später zu analysierenden Quelltextdateien zu kompilieren sind. Die Datenbank kann entweder über cmake mit dem Parameter `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` oder dem Tool Bear [18] erstellt werden. Diese Datenbank muss in der Datei `compile_commands.json` im selben Ordner wie die vom LibTooling-Programm zu analysierende Datei hinterlegt sein.

```
int main(int argc, const char **argv) {
        CommonOptionsParser OptionsParser(argc, argv);
        ClangTool Tool(OptionsParser.getCompilations(),
           OptionsParser.getSourcePathList());
        return Tool.run(newFrontendActionFactory<clang::
           SyntaxOnlyAction>());
}
```

Listing 2.5: LibTooling Beispiel: Parameterverarbeitung

3 Verwandte Arbeiten

3.1 Unterschiede zwischen OpenCL und anderen Frameworks

Es gibt viele weitere Frameworks, die sich im Detail zu OpenCL abgrenzen. Aufgrund der Vielfältigkeit und der damit verbundenen Gefahr, hier den Rahmen der Arbeit zu überschreiten, werden hier die wichtigsten aufgezählt.

CUDA

Das proprietäre Framework CUDA [19] (***C**ompute **U**nified **D**evice **A**rchitecture*) wurde von dem Hardwarehersteller Nvidia initiiert und erstmals im Jahr 2006 der Öffentlichkeit vorgestellt. Somit ist CUDA wesentlich älter als OpenCL. CUDA kann im Gegensatz zu OpenCL und anderen vergleichbaren Lösungen nur auf Nvidia-Hardware ausgeführt werden.

CUDA gilt als größter Konkurrent von OpenCL und ist dank seiner frühen Entwicklung mittlerweile auch technisch weiterentwickelt. CUDA ist primär auf Nvidia-Grafikkarten zugeschnitten, was sich im Hinblick zu OpenCL bemerkbar macht, da OpenCL in diesem Punkt abstrakter entworfen wurde. CUDA hingegen besitzt neuere Funktionen, da diese lediglich auf einer Hardwarekomponente von einem Hersteller ausgeführt werden muss. Nichtsdestotrotz sind sich CUDA und OpenCL in vielen Punkten ähnlich und benutzen zum Teil auch gleiche Konzepte.

C++ AMP und DirectCompute

Microsoft bietet mit dem offenen Standard C++ AMP [20] (***C++ A**ccelerated **M**assive **P**arallelism*) und DirectCompute [21] gleich zwei Möglichkeiten an, GPGPU (***G**eneral **P**urpose Computation on **G**raphics **P**rocessing **U**nit*) zu verwirklichen. C++ AMP bildet eine C++-Spracherweiterung, die nicht nur auf GPUs beschränkt ist, sondern theoretisch auf anderen Computerkomponenten wie der CPU lauffähig wäre. Die beiden Frameworks sind indes in DirectX implementiert.

OpenACC

Wie in OpenMP [22] erlaubt es der Standard OpenACC [23] (***Open Acc**elerators*) Codestellen in C, C++ und Fortran zu markieren, die später vom Compiler parallelisiert werden. Im Gegensatz zu OpenMP kann der Code nicht ausschließlich auf der CPU, sondern auch auf der GPU ausgeführt werden. So können beispielsweise Schleifen im Quelltext markiert werden, damit deren Codeblock bei der Ausführung parallel auf mehreren Recheneinheiten abgearbeitet wird und das Programm insgesamt beschleunigt wird.

libWater

libWater [24] stellt eine einfache Schnittstelle für heterogene Systeme mit Mehrkernprozessoren bereit und nutzt dazu auf jedem einzelnen System sowohl die CPUs als auch die GPUs. Das Programmiermodell von libWater stimmt mit OpenCL überein und erweitert dieses sogar in der Funktionalität.

Ähnlich wie in OpenCL existiert bei libWater ein Host-Knoten. Dieser besitzt eine globale Befehlswarteschlange, die mittels dem Scheduler WTR an die einzelnen Knoten via MPI [25] verteilt wird. Die einzelnen Knoten wiederum nehmen die Befehle über WTR entgegen, der diese zum einen an die lokale Befehlswarteschlange und zum anderen an die verschiedenen OpenCL-Warteschlangen verteilt. Jedes OpenCL-Device des Systems besitzt dabei eine OpenCL-Warteschlange, sodass dafür gesorgt wird, dass jedes OpenCL-Device mit Arbeit versorgt wird.

3.2 Partitionierung der Problemgrößen

Die folgenden Arbeiten behandeln die Auf- bzw. Umverteilung eines vorhandenen Codes auf mehrere Hardwarekomponenten. Dazu wird der gegebene Quellcode analysiert und entsprechend transformiert.

SnuCL

Ebenso wie libWater setzt auch SnuCL [26, 27] sowohl auf MPI und als auch auf OpenCL in einem heterogenen CPU-GPU-Cluster. Um dies zu ermöglichen, wird auf das Speichermanagement inklusive des Speichermodells von OpenCL zurückgegriffen und zunutze gemacht (siehe Kapitel 2.2). So wird die Speicherkonsistenz zwischen den einzelnen Speicherbedingungen bzw. Speicherlokalitäten aufgegriffen, um unnötige Kopiervorgänge zwischen den einzelnen Devices zu vermeiden. Die Adressabhängigkeiten werden über eine Pointer-Analyse am OpenCL-Kernelcode beim Bau des Kernels ermittelt.

Die Problemgröße wird nach Adressräumen aufgeteilt, sodass nach der Partitionierung eine aufgeteilte Problemgröße nur noch innerhalb eines Adressraums arbeitet. Die Codetransformation wird durch LLVM realisiert.

An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning

In dieser Publikation [28] wird beschrieben, wie in einem heterogenen Mehrkernsystem die Aufgabenaufteilung durch Partitionierung der Problemgröße optimiert werden kann, um so einen Geschwindigkeitsschub zu erlangen. Dazu wird die Problemgröße, die in OpenCL beschrieben wird, in kleinere Aufgaben, also mehrere OpenCL-Codes, aufgeteilt und es werden lediglich die Daten an die einzelnen Systemkomponenten wie CPU und GPU verteilt, die zur Berechnung dieser einzelnen Aufgaben benötigt werden.

Der OpenCL-Code wird mit der Eigenentwicklung Insieme [29] bearbeitet, indem durch Mitwirkung von Clang ein AST gebildet und dieser weiter untersucht wird. Anschließend wird der Code durch den Erkenntnisgewinn der vorangegangen Codeanalyse aufgeteilt, welcher weiter auf die Recheneinheiten verteilt wird.

Achieving a Single Compute Device Image in OpenCL for Multiple GPUs

Schwerpunkt dieser Arbeit [30] ist die Aufteilung eines OpenCL-Kernels auf mehreren GPUs. Das Framework verhält sich nach außen hin wie ein OpenCL-Device, verwaltet intern jedoch mehrere GPUs. Um den OpenCL-Kernel auf mehreren GPUs ausführen zu können, wird dieser zuerst in mehrere OpenCL-Kernel zerlegt und diese hinterher in mehrere CUDA-Kernel transformiert. Zur Ermittlung des Speicherzugriffsmusters werden während der Laufzeit des Kernelcodes die Bufferzugriffe mitgeschrieben und anhand derer ein Zugriffsmuster für eine Work-Group erstellt.

A Translation System for Enabling Data Mining Applications on GPUs

Im Gegensatz zu den vorhergegangenen Arbeiten wird in dieser Publikation [31] nicht OpenCL-Code, sondern C-Code analysiert und in CUDA-Code transformiert. Dabei wird das Zugriffsmuster der einzelnen Variablen über eine Pointer-Analyse anhand ihrer LLVM IR festgestellt und zur Erzeugung des CUDA-Codes verwendet.

3.3 Codetransformierung auf Basis der LLVM Intermediate Representation

LLVM eignet sich hervorragend, um Code zu optimieren. Dies geschieht durch das Prinzip der Source-zu-Source Transformation auf Basis der LLVM IR. Am Ende des oft zyklischen Vorgangs wird der neue, optimierte Zwischencode kompiliert.

Twin Peaks

Das oft referenzierte Twin Peaks [32] bietet eine Plattform für heterogene Systeme, um ursprünglich für GPUs geschriebenen Code auch auf CPUs ausführen zu können. So kann Twin Peaks den Code sowohl auf einen gemischten Cluster mit CPUs und GPUs als auch auf einen reinen CPU-Cluster ausführen. Um die Aufgaben auf CPU und GPU zu verteilen, wird OpenCL verwendet.

Im Mittelpunkt stehen dabei die Speicher- und Low-Level-Optimierung der einzelnen Devices, was Mithilfe der OpenCL-Speicherverwaltung und LLVM ermöglicht wird.

Efficient Profiling in the LLVM Compiler Infrastructure

In der Diplomarbeit [33] von Andreas Neustifter wird auf das Modell der Passes in LLVM eingegangen. Darin wird erläutert, dass alle Arbeiten am LLVM-IR-Code wie beispielsweise das Analysieren, Optimieren oder gar die Erzeugung von Maschinencode durch Passes geschieht [33, S. 31]. Insbesondere bei der Codeoptimierung durchläuft der LLVM-IR-Code, wie im Kapitel zuvor bereits beschrieben (siehe Kapitel 2.3.1), mehrmals denselben Pass.

Polly

Seit 2012 gehört Polly [34] zum LLVM-Projekt und kann Code analysieren und optimieren. In erster Linie optimiert das Werkzeug die Speicherzugriffe und Parallelität. Polly wird zudem via Pass in LLVM eingebunden, was zur Folge hat, dass neben den höheren Sprachen, die von Clang unterstützt werden, auch LLVM-IR-Code analysiert und optimiert werden kann.

Der Nachteil von Polly liegt in der noch nicht vorhanden Implementieung von OpenCL [35]. Dies könnte umgangen werden, indem für die OpenCL Befehle eine C-Methode geschrieben wird, die die Funktionalität simuliert.

3.4 Codebeispiele zu Clang

Die Wichtigkeit von Codebeispielen darf nicht unterschätzt werden. An ihnen wird die Art und Weise der Verwendung von API verdeutlicht. Somit tragen diese zum Verständnis der API-Dokumentation bei. Im Folgenden werden zwei Projekte vorgestellt, die auf Clang aufbauen.

SourceWeb

SourceWeb [36] ist ein in C++11 geschriebener Codenavigator, mit dem es möglich ist, den Quelltext eines Projektes zu indizieren und die einzelnen Verweise miteinander zu verknüpfen. Wie bei anderen IDEs kann hier über einzelne Variablen, Methoden oder Klassen durch den Code navigiert werden. Das Werkzeug macht Gebrauch von LibTooling (siehe Unterkapitel 2.4.2). Es wird jedoch neben den `FrontendActions` auch auf den `RecursiveASTVisitor` zurückgegriffen, sodass sich der Code dieses Projekts als gutes Beispiel für die spätere Implementierung herausstellt.

Woboq Code Browser

Woboq Code Browser [37] ist ein web-basierter Codenavigator für C/C++ Projekte, der ebenso wie SourceWeb die Clang API LibTooling und somit auch die `FrontendActions` und den `RecursiveASTVisitor` verwendet.

4 Bestimmung des Speicherzugriffsmusters durch Codeanalyse

In diesem Kapitel werden alle grundlegenden und theoretischen Fragen rund um das Zugriffsmuster, OpenCL und Compiler geklärt. Darüber hinaus wird dargelegt, wie diese Überlegungen schließlich implementiert wurden. Es wird davon ausgegangen, dass der zu analysierende Kernelcode fehlerfrei ist. Die Aufgabe dieser Arbeit ist nicht, Fehler im Kernelcode zu erkennen, sondern diesen zu analysieren.

Der Kernelcode wird von der Bibliothek angenommen und an das LLVM-Frontend Clang weitergereicht, um einen AST zu bilden. Mithilfe dieses ASTs wird die Codeanalyse durchgeführt und darauf das Speicherzugriffsmuster gebildet. Abbildung 4.1 veranschaulicht diesen Vorgang noch einmal.

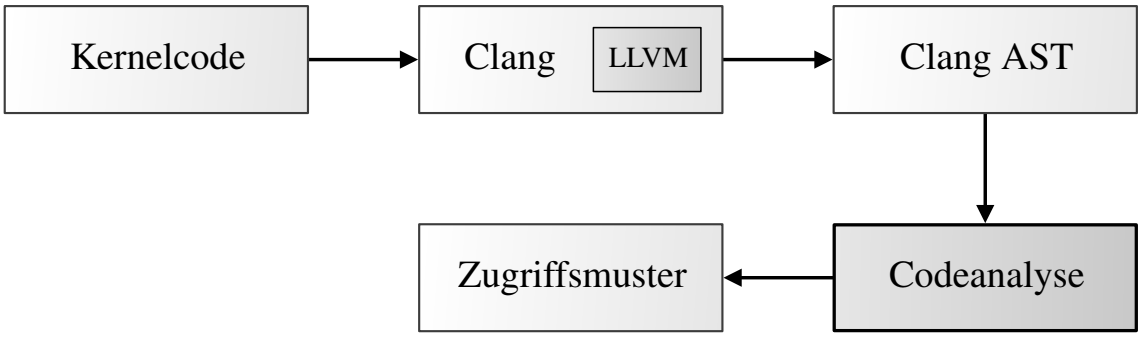

Abbildung 4.1: Verarbeitungskette des Kernelcodes zur Ermittlung des Zugriffsmusters

4.1 Die Darstellung eines Zugriffsmusters

Zur besseren Verständniss, was genau bei der Codeanalyse gemacht werden muss, damit das Zugriffsmuster erstellt werden kann, sollte zuvor geklärt werden, wie das Zugriffsmuster dargestellt wird und welche Informationen dafür benötigt werden.

Das Hauptziel besteht darin, unnötige Kopiervorgänge zwischen Host- und Device-Speicher zu verhindern, speziell im vorliegenden Fall zwischen der CPU und der GPU. Es müssen allein Daten von der CPU zur GPU transferiert werden, die in den Berechnungen der GPU auch benutzt werden. Zudem müssen ausschließlich die Daten wieder von der GPU zurück zur CPU übertragen werden, die verändert wurden.

```
c[i] = a[i] + b[i];
```

Listing 4.1: Beispiel für das Zugriffsmuster

Um dies zu illustrieren, wird in Listing 4.1 in einem Work-Item `i` der Arrayinhalt der jeweils `i`-ten Stelle von `a` mit `b` addiert und in `c` geschrieben. Auf `a` und `b` wird nur lesend zugegriffen, wohingegen dies auf `c` schreibend geschieht. Es wird außerdem jeweils auf das `i`-te Element interagiert. So müssen bei diesem Beispiel vor der Berechnung die `i`-ten Elemente der Arrays `a` und `b` zur GPU kopiert und der Speicher für `c` muss allokiert werden. Anschließend wird der Kernelcode ausgeführt und danach allein das `i`-te Element von Array `c` von der GPU zurück zur CPU an die richtige Speicherstelle im Host-Speicher transferiert. Dabei muss berücksichtigt werden, dass die Arrayelemente nach der Durchführung alle ihren richtigen Platz im Host-Speicher wiederfinden.

In Abbildung 4.2 wird deutlich, dass sich das gesamte Zugriffsmuster aus mehreren Work-Groups zusammenstellt. Diese bilden sich wiederum aus den Zugriffsmustern ihrer Work-Items. Es ist anzumerken, dass sich je nach Kernelcode die Work-Items auch gegenseitig beeinflussen können. In dieser Arbeit wird aufgrund des Umfangs lediglich das Zugriffsmuster eines Work-Items betrachtet und anschließend auf eine Work-Group bzw. die gesamte Laufzeit hoch skaliert.

Um den Indexwert herauszufinden, ist es oft nötig, Rechnungen zurückzuverfolgen. Diese müssen aufgezeichnet werden und können erst ab dem Zeitpunkt ausgerechnet werden, an dem die Rahmenbedingungen der OpenCL-Devices bekannt sind. Eine ausführliche Erklärung folgt in Kapitel 4.6.

Der nächste Schritt ist die Aufteilung der Datenverarbeitung zwischen der CPU und GPU. Daten, die bereits auf der CPU verarbeitet werden, müssen kein zweites Mal auf der GPU berechnet werden. Der Buffer, d.h. die Daten, die zur GPU transferiert werden, muss dazu zerlegt werden. Bei einzelnen Arrayzugriffen, wie es in Listing 4.1 geschieht, ist dieser Schritt noch einfach durchzuführen. Bereits bei einer Matrixmultiplikation wird die Aufgabe um einiges komplexer.

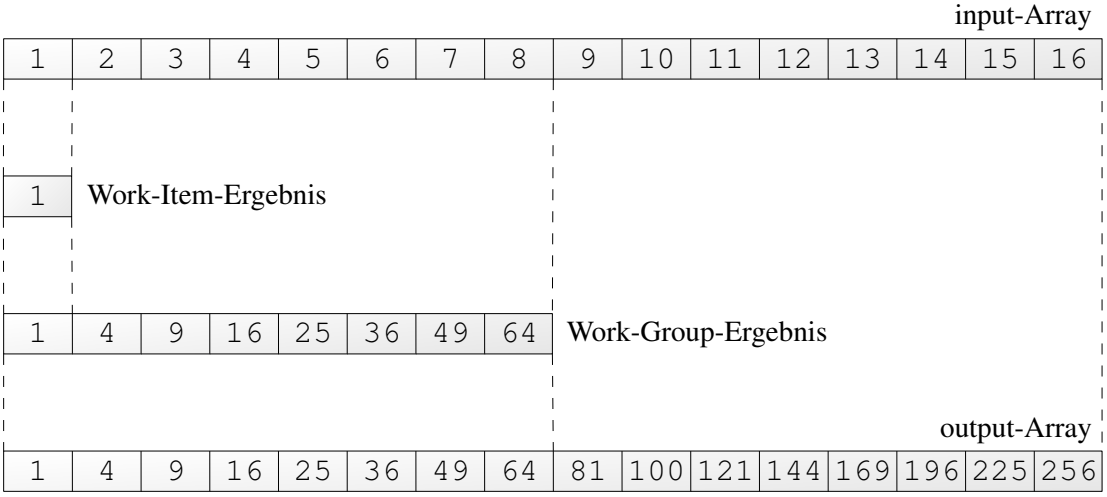

Abbildung 4.2: Veranschaulichung der verschiedenen Ergebnisse der Quadratfunktion in Listing 2.1: Der Code wird in zwei Work-Groups mit jeweils acht Work-Items ausgeführt.

4.2 Grundlagen zur Implementierung

Bevor mit dem Kernproblem dieser Arbeit fortgefahren werden kann, werden die Implementierungsmöglichkeiten aufgezählt sowie die daraus entstehenden Entscheidungen begründet. Schließlich wird der Zugriff auf den Clang AST beschrieben, mit dessen Hilfe der Kernelcode untersucht und die Kernprobleme gelöst werden.

4.2.1 Auflistung der Implementierungsmöglichkeiten

Es gibt mehrere Ansätze, die obige Überlegung zu verwirklichen. Einige davon sind mit weniger Aufwand zu implementieren, bringen jedoch Probleme mit sich, die bei den anderen Ansätzen von vornherein vermieden werden. Drei davon werden hier in kurzen Zügen vorgestellt.

Möglichkeit 1: Instrumentierung

Durch Instrumentieren des Kernelcodes an Codestellen, an denen ein Zugriff auf die Funktionsparameter der Kernelfunktion geschieht, kann das Muster in einem Testlauf ermittelt werden. Der Code würde dabei auf einem OpenCL-Device auf einer Work-Group und einem Work-Item ausgeführt werden. Zwar gelangt man so schnell an ein Speicherzugriffsmuster eines Work-Items, dieses wäre jedoch abhängig von der Eingabe und könnte dahingehend schlecht auf mehrere Work-Items und Work-Groups skaliert werden.

Als Beispiel kann hier der Kernelcode von Listing 2.1 herangezogen werden. Das Szenario, bei dem der Ausdruck `i < count` bei einem Testdurchlauf wahr ist, führt dazu, dass die darauf folgende Zeile und damit der Parameterzugriff in jedem Work-Item ausgeführt wird. Hieraus resultiert unweigerlich ein falsches Zugriffsmuster. Um diesen Faktor zu minimieren, müssten unter Umständen mehrere Testdurchläufe durchgeführt werden. Die Codeanalyse würde dadurch wiederum mehr Rechenzeit in Anspruch nehmen.

Möglichkeit 2: LLVM Passes

Eine weitere Möglichkeit bietet LLVM durch dessen Passes. LLVM setzt, wie bereits erläutert, eine Ebene tiefer an als Clang. Somit wird nicht mehr mit dem OpenCL-C-Code gearbeitet, sondern mit der LLVM IR. Dadurch ist die Codeanalyse um eine Problemgröße höher als bei der statischen Codeanalyse. Wird ein eigener Pass geschrieben, der die LLVM IR eines Kernelcodes analysiert, könnte dieser jedoch eventuell auch zur Codeoptimierung herangezogen werden.

Möglichkeit 3: Statische Codeanalyse mit Clang

Durch den Clang AST kann man sehr genau Zustände im Code analysieren und festhalten. Die Codeerkennung rückt hier mehr in den Hintergrund, da diese bereits von Clang AST übernommen wird. Viel elementarer dabei ist das Berechnen von Indexwerten, die später für einen Arrayzugriff benötigt werden. Während der statischen Codeanalyse wird kein Code ausgeführt, sondern lediglich betrachtet. Auf Grund dieser Tatsache müssen diese Berechnungen festgehalten und zu einem späteren Zeitpunkt ausgeführt werden können. Dies wird im Kapitel 4.6.1 genauer erläutert.

4.2.2 Auswahl und Begründung der Implementierungsmöglichkeit

Mit Clang AST und der daraus folgenden Funktionalität kann das Hauptmerkmal bei der Implementierung auf die Ergebnisverarbeitung gelegt werden, denn die eigentliche Syntaxanalyse übernimmt bereits Clang. Zudem ist dieser Ansatz gegenüber der ersten und zweiten Möglichkeit abstrakter gehalten, was abermals die Kernelanalyse erleichtert.

Die Implementierung selbst wird als Bibliothek verwirklicht. Dieses Vorgehen hat den Vorteil, dass die Kernelanalyse in einem größeren Framework leicht weiterverarbeitet werden kann. Folglich fällt die Entscheidung gegen LibTooling, da die Stärken dieser

API, wie in Kapitel 2.4.2 bereits angesprochen, hier nicht genutzt werden können. Ebenso entfällt „Clang Plugins“. Diese API ist vorzugsweise für den Kompiliervorgang geeignet und weniger zur Codeanalyse. So verbleibt lediglich LibClang, wobei hauptsächlich auf den Clang AST über den `RecursiveASTVisitor` gearbeitet wird. Hinzu kommt, dass alle anderen Clang APIs eine Abstraktionsebene höher angesiedelt sind und somit zum einen eine feinere Codeanalyse verhindern und zum anderen auf `RecursiveASTVisitor` zurückgreifen. Weil der Hauptteil von LLVM und Clang selbst in C++ geschrieben ist und aus Konsistenzgründen, wird die Bibliothek ebenfalls in C++ geschrieben.

4.2.3 Zugriff auf Clang AST

`RecursiveASTVisitor` arbeitet, wie der Name bereits suggeriert, rekursiv den AST ab. So wird der AST von oben nach unten, mit dem linken Kind eines Knotens zuerst, traversiert. Die zwei wichtigen Basisknotengruppen sind Statements (`Stmt`) und Declarations (`Decl`). Auf diesen Knotengruppen bauen alle anderen Knotentypen hierarchisch auf. Beispielsweise sind Expressions (`Expr`), die eine weitere wichtige Gruppe bilden, zugleich den Statements untergeordnet und können somit ebenso wie Statements behandelt werden.

Mit einem `ASTConsumer` kann das Verhalten des `RecursiveASTVisitor` beeinflusst werden. Dabei wird dem `ASTConsumer` eine eigene `RecursiveASTVisitor`-Klasse überreicht, die gewissermaßen die vorgegebene `RecursiveASTVisitor`-Klasse überschreibt. Wird ein Verhalten zu einer Knotengruppe nicht explizit in der eigenen `RecursiveASTVisitor`-Klasse definiert, so wird mit dem Standardverhalten fortgefahren.

```
bool KernelASTVisitor::VisitFunctionDecl(FunctionDecl *D) {
        if (!D->hasAttr<OpenCLKernelAttr>()) {
                return false;
        }

        kernelInfo.addFunction(*D);

        return true;
}
```

Listing 4.2: `VisitFunctionDecl` beschreibt was mit `FunctionDecl`-Knoten geschehen sollen.

Das Verhalten kann mit den zwei grundlegenden Funktionsarten `Visit` und `Traverse` beeinflusst werden. In der Regel wird die `Traverse`-Funktion zu einer Knotengruppe

```
bool KernelASTVisitor::TraverseBinAssign(BinaryOperator *S)
    {
        kernelInfo.getCurrentFunction()->updateVarDeclOp(
           clang::BO_Assign);
        {
                thisContext = CF_Write;
                TraverseStmt(S->getLHS());
        }
        {
                thisContext = CF_Read;
                TraverseStmt(S->getRHS());
        }

        return true;
}
```

Listing 4.3: `TraverseBinAssign` beschreibt wie `BinAssign`-Knoten abgearbeitet werden sollen.

immer vor derer `Visit`-Funktion aufgerufen. `VisitFunctionDecl` in Listing 4.2 beeinflusst beispielsweise die Verarbeitung von `FunctionDecl`-Knoten. Hier kann angesetzt werden, um wichtige Informationen an den Hauptcode weiter zu reichen. So wird in diesem Fall geprüft, ob die Funktionsdeklaration das Attribut `__kernel`, also ein `OpenCLKernelAttr` besitzt. Ist dem nicht so, wird `false` zurückgegeben und der Knoten und seine Kinder werden nicht weiter abgearbeitet und übersprungen. Andernfalls wird diese Funktion der Klasse `kernelInfo` mitgeteilt und wie gehabt fortgefahren.

Mit `TraverseBinAssign` in Listing 4.3 kann auf die Abarbeitung des `BinAssign`-Knotens Einfluss genommen werden. Im angeführten Beispiel wird der Knoten noch nicht besucht, sondern lediglich das Traversieren gesteuert. Durch die Funktion `TraverseStmt` wird zuerst das linke Kind und danach das rechte Kind abgearbeitet. Anschließend wird mit dem Nachbarknoten fortgefahren bzw. eine Ebene höher weitergearbeitet.

Zusammengefasst wird der Graphendurchlauf mit `Traverse` gesteuert und `Visit` dient zum Knotenaufruf.

4.3 Zu beachtende Teilaspekte von OpenCL

Wie bereits erwähnt, lässt sich der OpenCL-Code in Host- und Kernelcode einteilen. Auf der CPU wird der Hostcode ausgeführt, wohingegen der Kernelcode sowohl auf

der CPU als auch auf der GPU ausgeführt werden kann. Die GPU soll dabei die CPU entlasten und die Berechnungen beschleunigen. Natürlich können auch mehrere GPUs, CPUs oder andere OpenCL-Devices zur Berechnung herangezogen werden. Dies hätte jedoch keine weiteren Auswirkungen auf den oben genannten theoretischen Teil. Denn das Zugriffsmuster bildet sich weiterhin aus den Work-Items und Work-Groups. Allein der letzte Schritt unterscheidet sich dahingehend, dass die Anzahl der Work-Items und Work-Groups sich verändert hat.

4.3.1 Der OpenCL-Kernel

Der eigentliche OpenCL-Kernel bleibt unverändert. Dieser wird analysiert, um wie weiter oben beschrieben das Zugriffsmuster zu ermitteln. Folglich sind in ihm alle relevanten Informationen zur Bildung des Zugriffsmusters eines Work-Items zu finden.

Wie oben schon angedeutet, müssen ausschließlich Daten, die als Parameter an die Kernelfunktion übergeben werden, von der CPU zur GPU und wieder zurück kopiert werden. Daten bzw. Objekte, die innerhalb der Funktion initialisiert und verwendet werden, müssen in diesem Fall nicht zurück zur CPU transferiert werden.

Infolge dessen agiert die Analyse ausschließlich im Kernelcode. Der Hostcode wird in diesem Schritt komplett ignoriert. Erst am Schluss, wenn die Ergebnisse der Kernelanalyse weiter verarbeitet werden, um ein Zugriffsmuster anzulegen, werden die Rahmenbedingungen des Hostcodes benötigt.

Da die Implementierung als Bibliothek realisiert wird, wird der Kernelcode in Form eines Strings übergeben, womit ein zusätzliches Anlegen einer Datei entfällt. Dies hat den Hintergrund, dass der Kernelcode meist selbst im Hostcode als String hinterlegt ist.

Bevor dieser String als Code verwendet werden kann, muss bei Clang der Precompiler eingerichtet werden. Clang übersetzt mit den Standardeinstellungen den Kernelcode als C-Code. Dies reicht jedoch nicht aus, da OpenCL C, wie Anfangs in Kapitel 2.2 bereits erwähnt, keine Untermenge von C ist. Nichtsdestotrotz kann Clang mit der Parametereinstellung `getLangOpts().OpenCL = 1` eines `CompilerInstance`-Objekts OpenCL-Code kompilieren. Neben weiteren, hier nebensächlichen Precompilereinstellungen, ist die richtige Reihenfolge der Initialisierung der verschiedenen Klassen zu beachten.

4.3.2 OpenCL's Work-Item-Funktionen

Im Kernelcode ist es nötig, auf die Arrays sequentiell zuzugreifen. Dies geschieht mit den sogenannten Work-Item-Funktionen [38, S. 68], die einen Teil der Built-in Funktionen ausmachen. In OpenCL C werden diese für den Kernel zur Verfügung gestellt

und definieren sich durch die Rahmenbedingungen wie die Anzahl der Work-Items und Work-Groups im Hostcode. Die Funktion `get_global_id` in Listing 2.1 ist eine solche Work-Item-Funktion. Hier wird auch deutlich, dass zum Zeitpunkt der Kernelanalyse der Rückgabewert von `get_global_id(0)` unbekannt ist. Dieser lässt sich erst später bestimmen, sodass während der Analyse ein Platzhalter benötigt wird, der auf das Ergebnis von `get_global_id(0)` verweist.

Beim Parsen des Kernelcodes muss auf folgende Work-Item-Funktionen Rücksicht genommen werden, ohne deren Rückgabewert zu diesem Zeitpunkt zu kennen:

- `uint` **`get_work_dim`** `()`
- `size_t` **`get_global_size`** `(uint` *`dimindx`*`)`
- `size_t` **`get_global_id`** `(uint` *`dimindx`*`)`
- `size_t` **`get_local_size`** `(uint` *`dimindx`*`)`
- `size_t` **`get_enqueued_local_size`** `(uint` *`dimindx`*`)`
- `size_t` **`get_local_id`** `(uint` *`dimindx`*`)`
- `size_t` **`get_num_groups`** `(uint` *`dimindx`*`)`
- `size_t` **`get_group_id`** `(uint` *`dimindx`*`)`
- `size_t` **`get_global_offset`** `(uint` *`dimindx`*`)`
- `size_t` **`get_global_linear_id`** `()`
- `size_t` **`get_local_linear_id`** `()`

An dieser Stelle wird noch einmal darauf hingewiesen, dass die Work-Item-Funktionen innerhalb einer Work-Group je nach Work-Item unterschiedliche Werte zurückliefern kann. Um das Zugriffsmuster einer Work-Group zu bestimmen, muss folglich über seine Work-Items iteriert werden (siehe Abbildung 4.2). Eine Ebene höher muss zudem über die verschiedenen Work-Groups iteriert werden.

4.3.3 Rahmenbedingungen im Hostcode zur Bildung des Zugriffsmusters

Der Hostcode wird erst im letzten Schritt beim Erzeugen des gesamten Zugriffsmusters benötigt. Durch ihn wird die Anzahl der Work-Items und der Work-Groups festgelegt und die Work-Item-Funktionen näher definiert.

Damit das Zugriffsmuster nicht jedes mal aufs Neue berechnet werden muss, wenn sich ein Parameter im Hostcode verändert hat, muss das Zugriffsmuster eines Work-Items, also

die Analyse des Kernelcodes, ohne die Rahmenbedingungen des Hostcodes gebildet und festgehalten werden. Dies ist bereits im vorhergehenden Text durch das Referenzieren der Rückgabewerte der Work-Item-Funktionen geschehen.

Diese Referenzen werden über nachgebildete Work-Item-Funktionen mit Werten gefüllt, die wiederum anhand der Parameter der Funktion `clEnqueueNDRangeKernel` berechnet werden:

- `cl_uint` **`work_dim`**
- `const size_t *`**`global_work_offset`**
- `const size_t *`**`global_work_size`**
- `const size_t *`**`local_work_size`**

Für die Work-Item-Funktionen `get_global_size` ist es außerdem in den OpenCL-Versionen vor 2.0 wichtig zu wissen, ob der Kernel mit der Funktion `clEnqueueTask` im Hostcode in die Warteschlange eingereiht wird oder nicht.

4.4 Die Zugriffsarten auf ein Objekt

Um sicherzustellen, auf welche Indizes in einem Array zugegriffen wird und welchen Wert diese Indizes haben, müssen, wie anfangs erläutert, die Variablen im Code zurückverfolgt werden. Dies ist nur möglich, wenn bekannt ist, ob auf eine Variable lesend oder schreibend zugegriffen wird. Im Folgenden wird die Herangehensweise abstrakt gehalten. So wird zur Abdeckung von Sonderfällen von Objekten statt von Variablen gesprochen.

4.4.1 Der Lesezugriff

Damit festgestellt werden kann, welche Daten von der CPU zur GPU kopieren werden müssen, müssen diese beim Auslesen innerhalb des Kernelcodes markiert werden. Dies gilt zunächst einmal für alle Objekte, die in der Kernelfunktion als Parameter übergeben werden.

Weiter müssen auch diejenigen Objekte als lesend markiert werden, die innerhalb des Kernels verwendet werden, jedoch keine Funktionsparameter der Kernelfunktion sind. Dies hat den Hintergrund, dass beide Objektarten den Programmverlauf beeinflussen können und somit für eine erfolgreiche Berechnung benötigt werden.

Auf ein Objekt wird in folgenden Fällen lesend zugegriffen, wenn es

- auf der rechten Seite eines Zuweisungzeichens (*assign*) steht.
  z.B.: `int b = a` oder `b += a`
- als Parameter eines Funktionsaufrufs benutzt wird.
  z.B.: `function(a)`
- innerhalb eines Ausdrucks (*statement*) aufgerufen wird.
  z.B.: `a == 1`
- als Index in einem Array verwendet wird.
  z.B.: `array[a]`
- de- oder inkrementiert wird.
  z.B.: `a++`

4.4.2 Der Schreibzugriff

Nach der erfolgreichen Berechnung auf dem OpenCL-Device müssen die beschriebenen Objekte wieder zurück zur CPU transferiert werden. Hier reicht es im Gegensatz zu den lesenden Objekten nicht aus, diese einfach zu markieren. Es muss vielmehr der neue Wert zurückverfolgt und gespeichert werden, für den Fall, dass dieser Wert später bei einem Arrayzugriff als Index verwendet wird. Schließlich soll festgestellt werden, welches Arrayelement gelesen und welches beschrieben wird.

Hier sei darauf hingewiesen, dass der Unterschied zwischen der Deklaration und der Zuweisung eines Objektes bedeutungslos ist. Es geht allein darum, den aktuellen Wert eines Objekts zu kennen, wenn dieses später gelesen wird. Folglich werden beide Fälle als schreibend behandelt.

Deklaration

Eine Deklaration findet bei der Reservierung des Speichers statt. Im Programmcode drückt sich diese mit dem Objekttyp gefolgt vom Objekt aus (z.B.: `int a`).

Zuweisung

Bei einer Zuweisung nimmt ein Objekt einen neuen Wert an. Dies geschieht wie bei der Initialisierung auf der linken Seite eines Zuweisungszeichens, wobei der Objekttyp hierbei fehlt (z.B.: `a = 1` oder `a += 2`). Bei der De- und Inkrementierung wird das Objekt jedoch ebenfalls beschrieben (z.B.: `a++`).

Auf Grund der Vollständigkeit sei hier erwähnt, dass auch Objekte, die als Funktionsparameter übergeben werden, eine neue Zuweisung erfahren können (z.B.: `function(a)`). Um dies eindeutig sagen zu können, muss im Einzelnen die Funktion genauer betrachtet werden.

4.4.3 Ermittlung und Festhalten des Kontextes

Bei der Auswertung des AST muss oft der Zugriffsstatus festgehalten werden. Durch die hierarchische Struktur wird lediglich an der obersten Stelle im AST klar, ob ein Ausdruck links oder rechts von einem Zuweisungszeichen steht. Ebenso müssen Parameter von Funktionen, Array-Indizes und Variablen, die de- bzw. inkrementiert werden, mit der richtigen Zugriffsart für die Weiterverarbeitung markiert werden.

Um diesen Schritt zu verdeutlichen, wird als Beispiel Listing 4.4 herangezogen. Der zugehörige AST wird in Listing 4.5 dargestellt. Die Variable a wird mit dem Attribut `VarDecl` in der ersten Zeile als Variable markiert. Weiter ist a nicht nur eine Variable, sondern an dieser Stelle wird a deklariert. Dies hat zur Folge, dass a als schreibend markiert wird, unabhängig davon, ob a initialisiert wird oder nicht. Dies hat den Hintergrund, dass in der internen Datenstruktur der Bibliothek für die Variable a auch dann ein Wert hinterlegt werden muss, wenn die Variable nicht initialisiert wird, sondern lediglich deklariert. Liegt wie im Beispiel in der zweiten Zeile ein Unterzweig vor, so erfährt a eine Zuweisung. Alle weiteren Variablen, die im Unterzweig vorkommen, werden als lesend markiert. In diesem Fall sind das b, c und d.

```
int a = b + c + d;
```

Listing 4.4: Code-Beispiel für Abbildung 4.5

```
VarDecl 0x26e81c0 <col:2, col:18> a 'int'
`-BinaryOperator 0x26e8300 <col:10, col:18> 'int' '+'
  |-BinaryOperator 0x26e8298 <col:10, col:14> 'int' '+'
  | |-ImplicitCastExpr 0x26e8268 <col:10> 'int' <LValueToRValue>
  | | `-DeclRefExpr 0x26e8218 <col:10> 'int' lvalue Var 0x26e8020 'b' 'int'
  | `-ImplicitCastExpr 0x26e8280 <col:14> 'int' <LValueToRValue>
  |   `-DeclRefExpr 0x26e8240 <col:14> 'int' lvalue Var 0x26e8090 'c' 'int'
  `-ImplicitCastExpr 0x26e82e8 <col:18> 'int' <LValueToRValue>
    `-DeclRefExpr 0x26e82c0 <col:18> 'int' lvalue Var 0x26e8100 'd' 'int'
```

Listing 4.5: AST-Darstellung des Codes in Listing 4.4

Durch diesen hierarchischen Aufbau müssen Eigenschaften vom Elternzweig, speziell der Kontext, in dem die Variable steht, festgehalten werden, da Informationen wie beispielsweise diejenige, ob die Variable a auf der linken oder rechten Seite des Zuweisungszeichen steht, im Kinderzweig nicht mehr enthalten sind und somit verloren gingen.

Der Kontext ist jedoch nur für die Kinder vererbbar, Elternzweige dürfen dabei keine Eigenschaften der Kinder besitzen. Ansonsten besteht die Gefahr, dass Elternzweige falsche Kontextinformationen enthalten. Dieses Verfahren wird durch einen Stackspeicher gelöst, welcher auch in sourceweb (siehe Kapitel 3.4) verwendet wird.

4.5 Unterschiedliche Handhabung der Datentypen

OpenCL C unterstützt weit mehr Datentypen als die in C bereits bekannten. Prinzipiell kann jedoch davon ausgegangen werden, dass einfache Datentypen wie `bool`, `int`, `unsigned int` und `float` in OpenCL C unterstützt werden. Eine vollständige Liste ist in der OpenCL C Spezifikation [38, S. 6] enthalten. Im Weiteren werden lediglich die rudimentären und zugleich für diese Arbeit wichtigen Datentypen behandelt.

Hierbei sei erwähnt, dass Makros zum Zeitpunkt der Kernelcodeanalyse bereits durch den C-Präprozessor im Quelltext eingesetzt wurden. So müssen diese im Folgenden nicht weiter betrachtet werden.

Einfacher Datentyp

Obwohl OpenCL C viele einfache Datentypen unterstützt, werden in dieser Arbeit lediglich auf die einfachen Datentypen `int` und `unsigned int` eingegangen. Alle anderen Datentypen wie `float` werden als Integer angesehen und behandelt. Dies hat den Hintergrund, dass man letztlich das Zugriffsmuster eines Arrays herausfinden möchte. Auf die einzelnen Arrayelemente wird über den Index zugegriffen, der immer aus einem vorzeichenlosen Integer besteht. Datentypen wie `float` und `boolean` sind letztlich nur in Verbindung mit Bedingungen oder Typumkonvertierungen interessant und werden in dieser Arbeit nicht berücksichtigt. Nichtsdestotrotz wurde in der Spracheinstellung von Clang (`LangOptions`) das Flag `Bool` gesetzt. Dies bringt den Vorteil, dass Schreibzugriffe auf Booleans wahrgenommen werden, ohne Booleans in der Bibliothek implementiert zu haben.

Da Gleitkommazahlen als Integerzahlen behandelt werden, fallen die Nachkommastellen zur internen Weiterverarbeitung weg. Aus einer Gleitkommazahl mit dem Wert $0,2$ wird eine Integerzahl mit dem Wert 0. Dies hat weitreichende Konsequenzen: Ohne Vorbeugung wird in manchen Berechnungen durch Null geteilt und nicht durch $0,2$. Um Laufzeitfehler zu vermeiden, wird in diesem Fall nicht durch Null, sondern durch Eins geteilt. Dadurch können Folgefehler entstehen, die jedoch nicht weiter berücksichtigt werden!

Arrays

Arrays besitzen in Clang AST ebenso wie einfache Integer eindeutige IDs. Elemente eines Arrays hingegen besitzen keine eindeutige ID, sodass diese über die ID des Arrays inklusive dem Index bestimmt werden müssen. Auf diese Weise werden Arrays schließlich auch in der Bibliothek verwaltet.

Ein weiterer Unterschied zwischen Arrays und einfachen Datentypen wie Integers liegt in der Form der Initialisierung. Wie in Listing 4.6 deutlich wird, können Arrays auf zwei verschiedene Formen initialisiert werden. `array1` wird direkt bei der Deklaration initialisiert. Wohingegen `array2` im ersten Schritt deklariert und anschließend die einzelnen Arrayelemente nacheinander initialisiert werden.

```
int array1[] = {1, 2, 3};
int array2[3];
array2[0] = 1;
array2[1] = 2;
array2[2] = 3;
```

Listing 4.6: Arrays können auf zwei verschiedene Arten initialisiert werden

Funktionen

Funktionen innerhalb einer Kernelfunktion werden lesend aufgerufen. Es findet nie eine Funktionsdefinition innerhalb einer Kernelfunktion statt. Hierbei wird noch einmal darauf hingewiesen, dass OpenCL C keine Funktionspointer kennt.

Gibt eine Funktion einen Wert zurück, so wird die Funktion wie ein Objekt behandelt, auf dem ein Lesezugriff im Sinne von Kapitel 4.4.1 stattfindet. Der Fall, dass eine Funktion de- oder inkrementiert wird, fällt hingegen weg. Funktionen ohne Rückgabewert können nur aufgerufen werden und stehen somit alleine in einer Zeile.

Objekte, die bei einem Funktionsaufruf als Funktionsparameter übergeben werden, werden der Einfachheit halber grundsätzlich als lesende Objekte behandelt. Zusätzlich könnten diese auch beschrieben werden, sodass man die Funktion näher untersuchen muss. In diesem Fall wird rekursiv wie bei einer Kernelfunktion verfahren (siehe Kapitel 4.4).

Alle nicht Built-in Funktionen müssen im Kernelcode definiert sein. Demzufolge muss bei einem Funktionsaufruf geprüft werden, ob die Funktion entweder eine Built-in Funktion oder eine im Kernelcode definierte Funktion ist. Letztlich werden ausschließlich Work-Item-Funktionen unterstützt.

4.6 Problematik unbekannter Größen

Wie schon weiter oben mehrfach angedeutet, ist zum Zeitpunkt der Kernelanalyse die Anzahl der Work-Items und Work-Groups unbekannt. Diese wird erst später im Hostcode durch weitere Parameter definiert. Um dieses Problem zu lösen, müssen Rechnungen und Abhängigkeiten der im Kernelcode vorkommenden Variablen gespeichert und zum späteren Zeitpunkt eingesetzt, aufgelöst und berechnet werden. Dies betrifft zwar nur Variablen, die direkt und indirekt den Speicherzugriff beeinflussen können, faktisch muss dieses Konzept bei der Codeanalyse aber auf jede Variable angewandt werden.

Damit diese Problematik etwas deutlicher wird, wird das Beispiel Listing 4.7 herangezogen und in den Unterkapiteln 4.6.1 und 4.6.2 näher erklärt.

```
int i = get_global_id(0) + 1;
if (i < count) {
        output[i] = input[i] * input[i];
}
```

Listing 4.7: Beispielcode für Abhängigkeiten

4.6.1 Referenzierung von Rechnungen

In Listing 4.7 ist zu sehen, dass die Variable `i` später für den Arrayzugriff auf `output` und `input` als Index verwendet wird. Um nun ein Zugriffsmuster für diese zwei Arrays bilden zu können, wird der Wert von `i` benötigt. Dieser Wert ist immer vor dem eigentlichen Aufruf zu finden. Denn bevor eine Variable verwendet werden kann, muss diese erst definiert und mit einem Wert belegt werden. Dieses Verfahren deckt sich mit der gängigen Art der Codeanalyse, die von oben nach unten verläuft. Durch die theoretischen Überlegungen in 4.4 ist auch bekannt, wann eine Variable gelesen und wann sie einen Wert zugewiesen bekommt.

In diesem Beispiel besitzt `i` den Wert `get_global_id(0) + 1`. Zum Zeitpunkt der Codeanalyse ist der Rückgabewert von `get_global_id(0)` unbekannt. Diese Problematik wird wie in 4.3.2 bereits angedeutet durch Referenzen gelöst. Es wird eine Referenz `ref_get_global_id` für `get_global_id(0)` in einer Liste angelegt. Um diese Liste später mit dem richtigen Wert füllen zu können, muss zudem der Funktionsparameter `0` mit abgespeichert werden. Ab hier wird mit der Referenz weiter gerechnet, sodass `i` fortan den Wert `ref_get_global_id + 1` besitzt.

Der Wert für die Variable `i` kann nun im Nachhinein berechnet werden, jedoch weiterhin nicht zum Zeitpunkt der Codeanalyse; folglich ist auch bei der Verarbeitung der zweiten

Zeile beim Ausdruck `i < count` der Wert für `i` weiterhin unbekannt. Die Berechnung für `i` muss also mitsamt der Referenz festgehalten werden und überall dort, wo diese Variable aufgerufen wird, durch die Rechnung `ref_get_global_id + 1` ersetzt werden. Aus dem vorherigen Ausdruck wird so `ref_get_global_id + 1 < count`.

Dieser Prozess wird in der nächsten Zeile wiederholt, sodass der daraus resultierende Code ab sofort wie in Listing 4.8 dargestellt wird. Mit dieser Codedarstellung kann der Index der Arrays `output` und `input` im Nachhinein berechnet werden.

```
size_t ref_get_global_id = get_global_id(0);
if (ref_get_global_id + 1 < count) {
        output[ref_get_global_id + 1] =
                input[ref_get_global_id + 1]
                * input[ref_get_global_id + 1];
}
```

Listing 4.8: Codedarstellung nach der Codeanalyse

4.6.2 Erzeugung des Maschinencodes zur späteren Ausführung

Durch die Problemlösung der unbekannten Variablen im vorhergegangenen Unterkapitel wurde ein neues Problem erschaffen: Der aktuelle Wert einer Variable muss zwischengespeichert werden und darf dabei nicht den Bezug zur Referenz verlieren. Gleichzeitig kann der wirkliche Wert erst nach der Codeanalyse berechnet werden.

Dies zieht das konsequente Speichern der Rechnungen für die verschiedenen Variablen mit sich. Durch eine effektive Infrastruktur soll zudem garantiert werden, dass zum einen so wenig Speicherplatz wie nötig verwendet wird und zum anderen das Zwischenergebnis stets berechnet werden kann.

```
int i = get_global_id(0) + 1;
i = 2 * i;
int j = i - 1;
```

Listing 4.9: Codebeispiel zum Maschinencode

Anhand des Beispiels Listing 4.9 wird das Prinzip und die spätere Umsetzung näher erläutert. In der ersten Zeile werden zwei Zahlen mit einander addiert. Der Rückgabewert von `get_global_id(0)` ist zu diesem Zeitpunkt unbekannt und so wird eine Referenz als Platzhalter für diesen Wert verwendet. `1` hingegen kann als konstanter Wert

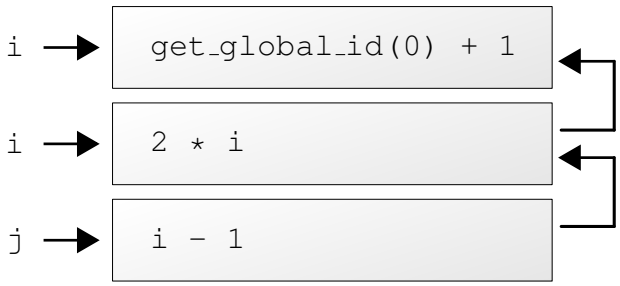

Abbildung 4.3: Illustration der Liste

übernommen werden. In einer Liste wird `i` mit seinem Wert als Addition hinzugefügt. Diese Liste wird in Zeile zwei ausgelesen, bei der `i` mit `2` multipliziert wird. Wie zuvor wird wieder `i` mit der neuen Berechnung am Ende der Liste eingetragen, mit dem Unterschied, dass dieses Mal nicht `get_global_id(0)` referenziert wird, sondern `i`. Dabei ist wichtig, dass der alte Wert weiterhin eingetragen und im Speicher festgehalten ist. Denn dieser wird zur Berechnung des zweiten Eintrags in Form einer Referenz benötigt. In der dritten Zeile wird der Wert bzw. die Berechnung von `i` in der Liste von unten nach oben hin gesucht. Dies ist nötig, damit der aktuellen Wert von `i` zurückgegeben wird. Schlussendlich wird `i` mit `1` subtrahiert und dies zusammen mit `j` wieder am Ende der Liste eingetragen. Das Resultat wird in Abbildung 4.3 illustriert.

Operatoren

Das Konzept gleicht dem der Funktionalen Programmierung. Der Operand wird als Funktion vor seinen eigentlichen Parametern geschrieben. Des Weiteren besitzt jeder Operand entweder ein oder zwei Parameter. Als Parameter kann eine Zahl, eine Variable oder ein weiterer Operand dienen. Durch dieses Grundkonzept lassen sich alle mathematischen Formeln in einer Art Hierarchie beschreiben. Als Beispiel wird hier die Formel in Listing 4.10 betrachtet. Ihre funktionale Beschreibung stellt 4.11 dar. Abbildung 4.4 verdeutlicht diesen Ansatz graphisch.

```
(++a + b * c) - a / 2
```

Listing 4.10: Beispiel zur funktionalen Darstellung (davor)

```
-(+(pre++(a), *(b, c)), /(a, 2))
```

Listing 4.11: Beispiel zur funktionalen Darstellung (danach)

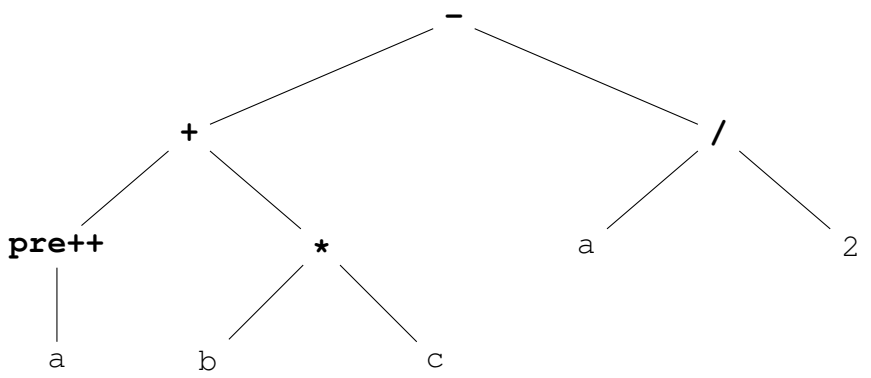

Abbildung 4.4: Graph als Beispiel zur funktionalen Darstellung

```
int result = (++a + b * c) - a / 2;
```

Listing 4.12: Beispielcode für Clang-AST-Darstellung

```
DeclStmt 0x27300f0 <line:5:2, col:36>
`-VarDecl 0x272feb0 <col:2, col:35> result 'int'
  `-BinaryOperator 0x27300c8 <col:15, col:35> 'int' '-'
    |-ParenExpr 0x2730020 <col:15, col:27> 'int'
    | `-BinaryOperator 0x272fff8 <col:16, col:26> 'int' '+'
    |   |-UnaryOperator 0x272ff30 <col:16, col:18> 'int' prefix '++'
    |   | `-DeclRefExpr 0x272ff08 <col:18> 'int' lvalue Var 0x26e78a0 'a' 'int'
    |   `-BinaryOperator 0x272ffd0 <col:22, col:26> 'int' '*'
    |     |-ImplicitCastExpr 0x272ffa0 <col:22> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x272ff50 <col:22> 'int' lvalue Var 0x26e7940 'b' 'int'
    |     `-ImplicitCastExpr 0x272ffb8 <col:26> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x272ff78 <col:26> 'int' lvalue Var 0x26e79e0 'c' 'int'
    `-BinaryOperator 0x27300a0 <col:31, col:35> 'int' '/'
      |-ImplicitCastExpr 0x2730088 <col:31> 'int' <LValueToRValue>
      | `-DeclRefExpr 0x2730040 <col:31> 'int' lvalue Var 0x26e78a0 'a' 'int'
      `-IntegerLiteral 0x2730068 <col:35> 'int' 2
```

Listing 4.13: Clang-AST-Darstellung zu Listing 4.12

Dies deckt sich mit der Verarbeitung des Clang AST, denn dieser stellt den Codeabschnitt in Listing 4.12 ebenso wie oben beschrieben in Listing 4.13 dar. Da der AST rekursiv abgearbeitet wird, wird zuerst der Knoten mit dem Operator besucht, bevor der zugehörige bzw. die zugehörigen Parameter besucht werden. Ein Parameter kann wiederum ein Operator oder eine Variable sein, wobei zu beachten ist, dass es zwei verschiedene Operatoren gibt. Ein unärer Operator besitzt, wie der Name bereits suggeriert, einen Parameter. Zu dieser Art von Operatoren zählen beispielsweise De- und Inkrementierungen. Der binäre Operator hingegen hält zwei Parameter fest. Hierzu zählen klassische Operatoren wie Addition und Subtraktion.

De- und Inkrementierungen

De- und Inkrementierungen bilden bei den Operationen eine Ausnahme, da diese sowohl lesend als auch schreibend aufgerufen werden müssen. Dabei ist zu beachten, dass der neue Variablenwert in der Liste aller festgehaltenen Variablenwerte richtig eingeordnet wird, sodass der Lesezugriff stets den richtigen Wert zurück gibt. Das Beispiel in Listing 4.14 beschreibt diese Problematik, denn in diesem Fall bindet das Zuweisungszeichen = stärker als die Postinkrementierung `k++`, sodass sich der Variablenwert von `k` nach der Ausführung der Programmzeile effektiv nicht ändert.

```
k = k++;
```

Listing 4.14: Der neue Variablenwert muss richtig in die Liste aller Variablenwerte eingeordnet werden.

Dieses Problem kann leicht umgangen werden, indem alle Prede- und Preinkrementierungen in der Berechnung vom neuen Wert von `k` durch `x - 1` bzw. `x + 1` ersetzt werden und alle Postde- und Postinkrementierungen fallen gelassen werden. Gleichzeitig werden alle De- und Inkrementierungen nacheinander abgearbeitet und in der Variablenliste chronologisch vor dem neuen Wert `k` eingegliedert. So wird die Programmzeile in Listing 4.15 intern nach Listing 4.16 umgesetzt.

```
k = k++ + --y;
```

Listing 4.15: Alle De- und Inkrementierungen müssen vor der Definition von `k` in die Variablenliste eingegliedert werden.

```
int k1 = k;
int y1 = y;

k = k1 + 1;
y = y1 - 1;
k = k1 + (y1 - 1);
```

Listing 4.16: Interne Umsetzung zum Programmcode in Listing 4.15

Variablen

Die zustande gekommene Infrastruktur erlaubt es also, Berechnungen zu einem späteren Zeitpunkt durchzuführen. So kann eine Variable in einer weiteren Rechnung wiederverwendet werden, ohne dass das konkrete Ergebnis dieser Variable bekannt ist. Wird eine

neue Variable innerhalb einer Rechnung angelegt, so wird zwischen konstanten Zahlen und Referenzen unterschieden. Konstante Zahlen können direkt in der neu angelegten Variable festgehalten werden. Sie besitzen stets denselben Wert. Referenzen hingegen müssen angelegt werden, wenn im Kernelcode beispielsweise eine Variable von einer Built-in Funktion wie `get_global_id()` abhängig ist. Der Rückgabewert einer solchen Funktion ist, wie oben bereits geschildert, erst nach der Codeanalyse bekannt. Diese Referenz nimmt automatisch den richtigen Wert der Built-in Funktion an, sobald die Rahmenbedingungen gesetzt wurden. Daraus folgt, dass auch eine Rechnung, die diese Variable als Parameter festhält, zu diesem Zeitpunkt den korrekten Zahlenwert berechnet.

Arrays

Während die Werte von Variablen in einer Liste festgehalten werden, müssen Arrays gesondert behandelt werden. Hierbei stößt man auf ein zusätzliches Problem, das in Listing 4.17 verdeutlicht wird. Wie weiter oben bereits erklärt, wird in der ersten Zeile `i` ein unbekannter Wert zugewiesen, mit dem auf das `i`-te Arrayelement in der Folgezeile zugegriffen wird. Zum Zeitpunkt der Kernelanalyse kann folglich nicht festgestellt werden, welchem Arrayelement der Wert 1 zugewiesen wird. Weiter ist unbekannt, ob das vierte Arrayelement, das in der letzten Zeile lesend aufgerufen wird, nun den neuen zugewiesenen Wert `1` besitzt.

```
int i = get_global_id();
array[i] = 1;
int result = array[3];
```

Listing 4.17: Codebeispiel zur Arrayproblematik

Aus diesem Beispiel wird klar, dass neben den Arrayelementen auch ihre Indizes variabel gespeichert werden müssen. Zudem kann der Wert eines Arrayelements erst zu einem späteren Zeitpunkt abgerufen werden, wenn der Wert der Variable `i` bekannt ist. Demzufolge kann dies erst nach der Codeanalyse geschehen.

```
int array[] = {1, 3, 7, 2};
array[2] = 9;
array[1] = 6;
array[3] = 8;
array[1] = 5;
```

Listing 4.18: Quelltext zur Abbildung 4.5

Jedes Array wird einmal in einer Liste von Arrays abgelegt. In dieser Liste wird das gesuchte Array über seine ID gefunden. Anders als bei der Variablenliste wird ein Array,

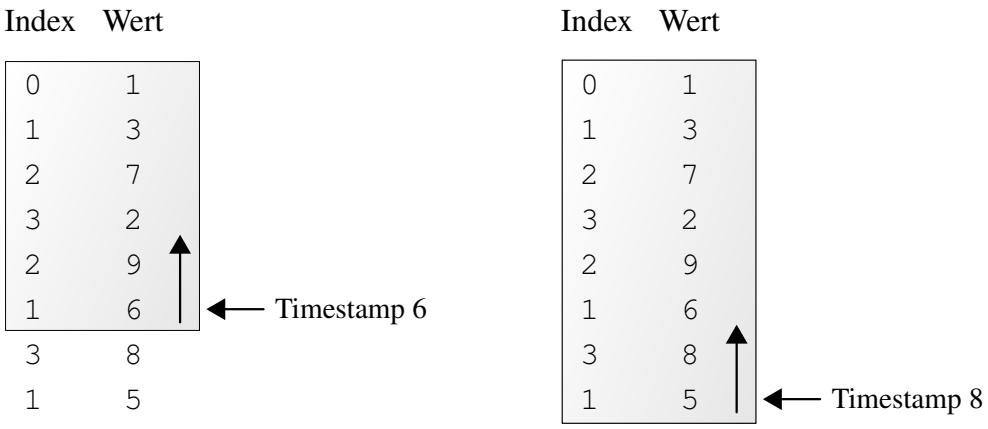

Abbildung 4.5: Speicherstruktur eines Arrays

dessen Element verändert wurde, nicht noch einmal in der Liste abgelegt. Dies würde unnötige Kopiervorgänge bzw. Referenzierungen der einzelnen Elemente des Arrays mit sich bringen. Der neue Wert des Arrayelements wird direkt in die Arraystruktur gespeichert. Abbildung 4.5 zeigt eine solche Arraystruktur für den Code in Listing 4.18. In dieser wird jedes Arrayelement zum einen mit seinem variablen Wert und zum andern mit seinem zugehörigen, variablen Index gespeichert. Wird einem Arrayelement ein neuer Wert zugewiesen, so wird dieser inklusive des Index am Ende der Liste hinzugefügt.

Wird der Wert des i-ten Arrayelements gesucht, so wird die Liste nach dem i-ten Eintrag von hinten nach vorne, also nach Arrayelement mit dem $(i-1)$-ten Index, durchsucht. Damit erhält man stets den aktuellen Elementwert. Dieser Vorgang geschieht nach der Codeanalyse, während dieser wird lediglich die ID des Arrays und der Index des Arrayelements gespeichert. Dies hat den Hintergrund, dass sich die Indizes wie weiter oben beschrieben, erst nach der Codeanalyse eindeutig bestimmen lassen.

```
int array[] = {1, 2, 3};
int k = array[0];
array[0] = 2;
```

Listing 4.19: Das erste Arrayelement erfährt zwei Zuweisungen

Ein weiteres Problem ist die Tatsache, dass diese Liste im Laufe der Codeanalyse ständig weiter wächst. In Listing 4.19 wurde dem ersten Arrayelement zweimal ein Wert zugewiesen. So reicht es nicht, ausschließlich am Ende der Liste nach dem Index des gewünschten Arrayelements zu suchen, da nach der Codeanalyse lediglich der aktuelle Wert des Arrayelements zurückgegeben wird. `k` würde fälschlicherweise den Wert `2` zugewiesen bekommen. Dieses Problem kann einfach durch einen Timestamp gelöst werden. Dieser wird bei einer Zuweisung eines Arrayelements auf eine Variable mit abgespeichert. Dabei han-

delt es sich schlicht um die aktuelle Anzahl der Arrayelemente, die in der Struktur des zugehörigen Arrays gehalten werden. Beim späteren Abrufen des Arrayelements wird an dieser Stelle in der Liste aufwärts nach dem passenden Arrayelement gesucht. In Abbildung 4.5 wird deutlich, dass das Array mit dem Timestamp 6 die Werte {`1, 6, 9, 2`}, wohingegen dasselbe Array mit Timestamp 8 die Werte {`1, 5, 9, 8`} enthält.

```
array[i++] = ++k;
k = array[++i / 2 + 1];
array[i] += 2;
array[2] += array[3]++;
```

Listing 4.20: Zu beachtende Fälle

Wie bei Funktionen im Parameterbereich, können auch im Indexbereich des Arrays Rechnungen jeder Art durchgeführt werden. Hinzu kommt, dass auch Arrayelement de- und inkrementiert werden können. So müssen während der Codeanalyse Fälle, wie in Listing 4.20 aufgeführt, beachtet werden.

Die Datenstruktur der Arrays erlaubt es außerdem, unabhängig von welchem Datentyp die Arrayelemente sind, jeden Arrayzugriff wahrzunehmen. Dies bedeutet im Einzelnen, dass Booleanarrays oder auch Floatingarrays wie Integerarrays behandelt werden, obwohl die Datentypen `boolean` und `float` nicht explizit in der Bibliothek implementiert sind.

4.7 Berechnung des Speicherzugriffsmusters

Nach der Kernelanalyse sind dank den vorgestellten Herangehensweisen und Techniken alle Variablen und Arrayelemente bekannt. Zudem sind diese entsprechend miteinander verknüpft und die Work-Item-Funktionen 4.3.2 können je nach Rahmenbedingung im Hostcode 4.3.3 verschiedene Werte zurückgeben. Der letzte und zugleich wichtigste Schritt ist das Bilden des Speicherzugriffsmusters. Dazu muss zu jedem Work-Item das Zugriffsmuster berechnet und festgehalten werden. Zu diesem Zweck wird über die `group_id` und `local_work_id` iteriert, wobei diese Einfluss auf den Rückgabewert der Work-Item-Funktionen nehmen 4.21.

Bei der Bildung des Work-Item-Speicherzugriffsmusters wird für alle Kernelfunktionsparameter das Zugriffsmuster abgefragt. Für einfache Variablen bedeutet dies lediglich, ob darauf lesend, schreibend, beides oder weder noch zugegriffen wird. Für Arrays hingegen muss dies pro Arrayelement geschehen: es werden zwei `boolean`-Arrays angelegt, eins für Lese- und das Zweite für Schreibzugriffe. Diese Arrays sind so ausgelegt, dass ein Arrayzugriff auf das erste Element im Kernelcode auch im ersten Element des entsprechenden `boolean`-Arrays festgehalten wird. Weiter sind die `boolean`-Arrays nur so

```
for (group_id = 0; group_id < global_work_size; ++group_id) {
        for (local_work_id = 0; local_work_id < local_work_size;
                        ++local_work_id) {

                /* Das Work-Item-Speicherzugriffsmuster
                   berechnet  und in einer Liste festgehalten */
        }
}
```

Listing 4.21: Berechnung des Speicherzugriffsmusters aller Work-Items

groß, dass das Zugriffsmuster für das Arrayelement mit dem größten Index am Ende des `boolean`-Arrays notiert wird. Letztlich wird dieses Speicherzugriffsmuster eines Work-Items im `struct ParameterPattern` festgehalten 4.22, wobei `name` den Parameternamen der Kernelfunktion enthält und die Arrays `reads` und `writes` bei Variablen eine Größe von `1` besitzen.

```
struct ParameterPattern {
        const char* name;

        PatternType type;

        bool* reads;
        uint readsSize;

        bool* writes;
        uint writesSize;
}
```

Listing 4.22: Vereinfachte Form des `ParameterPattern`

Die Speicherzugriffsmuster der einzelnen Work-Items pro Funktionsparameter werden im nächsten Schritt via `OR`s miteinander kombiniert, um an das Speicherzugriffsmuster aller Work-Items pro Funktionsparameter für das OpenCL-Gerät zu gelangen. Die Zugriffsmuster der Funktionsparameter werden wiederum in `FunctionPattern` festgehalten, dessen API so gestaltet wurde, dass diese reine C-Syntax enthält:

- `const char* **getName** ()`
  Gibt den Kernelfunktionsnamen zurück
- `uint **getParameterSize** ()`
  Gibt die Anzahl der von der Kernelfunktion festgehaltenen Parameter zurück

- `ParameterPattern* **getParameter** (uint *index*)`
  Gibt das Speicherzugriffsmuster des angegebenen Funktionsparameters zurück

Die Klasse `KernelInfo` hält die Ergebnisse der Kernelcodeanalyse fest und generiert schließlich für jede Kernelfunktion das Speicherzugriffsmuster. Nach außen hin besitzt sie eine ähnliche API wie die `FunctionPattern`:

- ```
  bool setSetting (size_t global_work_size,
                   size_t local_work_size,
                   size_t global_work_offset,
                   uint work_dim = 1,
                   bool clEnqueueTask = false)
  ```
  Rahmenbedingung für den OpenCL-Kernel setzen. Gibt `true` zurück, falls diese erfolgreich übernommen wurde.

- `uint **getFunctionSize** ()`
  Gibt die Anzahl der festgehaltenen Kernelfunktionen zurück

- `FunctionPattern* **getFunction** (uint *index*)`
  Gibt die `FunctionPattern` der angegebenen Kernelfunktion zurück

Die Speicherzugriffsmuster werden erst beim Funktionsaufruf `getFunction` generiert. Dies hat den einfachen Hintergrund, dass sich zum einen die Rahmenbedingung für den OpenCL-Kernel ändern könnte und zum anderen für das Zugriffsmuster nicht immer alle Kernelfunktionen benötigt werden. In diesem Fall wird immer nur das Zugriffsmuster für die erfragte Kernelfunktion berechnet. Ein möglicher Ablauf des Bibliotheksaufrufs könnte Listing 4.23 sein. Hierbei wird der OpenCL-Kernel in Stringform dem Konstruktor des Objekts `Analyse` in der ersten Zeile übergeben und zugleich die Kernelcodeanalyse initialisiert.
  ```

```
Analyse analyse(kernel);
KernelInfo kernelInfo = analyse.getKernelInfo();

kernelInfo.setSetting(1, 0, 64, 1024);

for (uint i = 0; i < kernelInfo.getFunctionSize(); i++) {
        FunctionPattern* function = kernelInfo.getFunction(i);

        for (uint j = 0; j < function->getParameterSize(); j++)
        {
                ParameterPattern* parameter =
                                function->getParameter(j);

                /* Verarbeitung des Speicherzugriffsmusters */
        }
}
```

Listing 4.23: Möglicher Ablauf einer Kernelcodeanalyse

5 Evaluation

Während der Implementation der einzelnen Funktionalitäten sind einige selbst geschriebene Testklassen entstanden, die zum einen zur Fehlersuche dienen und zum anderen den Fortbestand der Funktionalität sichern sollen, der durch neue Implementationen gefährdet werden könnte. Diese Testklassen testen jedoch nur auf erdachte Funktionalität. Die Gefahr selbst geschriebener Testklassen besteht darin, dass diese ausschließlich auf erdachte Codekonstellationen testen und nicht berücksichtigte Fälle außer Acht lassen. Umso wichtiger ist eine Evaluation der Bibliothek mit gegebenem, fremdem Kernelcode. Zuvor wird jedoch noch kurz auf die einzelnen Testklassen Bezug genommen:

- `ArrayTest`:
  Hiermit wird die Initialisierung, Deklaration und Zugriffe auf einem Array, sowie dessen Arrayelemente auf Korrektheit getestet.

- `DeAndIncrementTest`:
  Hier werden alle möglichen De- und Inkrementierungen inklusive Pre- und Post-Variationen getestet.

- `DependenceTest`:
  Sowohl die Abhängigkeit der einzelnen Variablen zueinander als auch die korrekte Zuweisung und Berechnung der Werte werden in dieser Testklasse vorgenommen.

- `MathTest`:
  In dieser Testklasse werden alle implementierten C-Operatoren getestet, inklusive einer Anzahl an Variationen mehrerer, hintereinander ausgeführter Operatoren.

Zur Auswertung der OpenCL-Kernel wurden die Einstellungen in Tabelle 5.1 verwendet. Für die `global_work_size` und `local_work_size` wurden außerdem kleine Werte gewählt, um bei der Auswertung nicht die Übersicht zu verlieren. Des Weiteren wurden Includes angepasst, sodass keine Umgebung separat geladen werden musste. Zur Einschätzung der Schnelligkeit der Kernelanalyse und der Generierung des Speicherzugriffsmusters wurde die gemittelte Laufzeit aus jeweils drei Messungen angegeben. Das System, auf dem die Laufzeit gemessen wurde, kann in Tabelle 5.2 betrachtet werden.

Parameter	Wert
global_work_size	32
local_work_size	8
global_work_offset	0
work_dim	1
clEnqueueTask	false

Tabelle 5.1: Alle OpenCL-Kernel wurden mit den gleichen Rahmenbedingungen ausgewertet. Zur Wahrung der Übersichtlichkeit wurden hier kleine Werte genommen.

Prozessor	AMD FX 6100 6-Core
Taktfrequenz	3,3 GHz
Arbeitsspeichergröße	8 GB DDR3
Betriebssystem	Arch Linux x86_64
Linuxkernel	3.14.5-1-ARCH

Tabelle 5.2: Systemkomponenten zur Bestimmung der Laufzeit

5.1 Vectoraddition

Zu Beginn wird eine einfache Vectoraddition evaluiert. Listing 5.1 zeigt den zu analysierenden Kernelcode. Wie in Tabelle 5.3 zu sehen, wird das Speicherzugriffsmuster der Vectoraddition von der Bibliothek richtig erkannt. Die Laufzeit der Codeanalyse inklusive der Zugriffsmustergenerierung liegt bei etwa 20 ms.

```
__kernel void vectorAdd(__global const float *a,
                        __global const float *b,
                        __global float *c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Listing 5.1: Eine Vectoraddition im OpenCL-Kernelcode

Parameter	Typ	Zugriffsmuster
a	Array	R: 11111111.11111111.11111111.11111111
b	Array	R: 11111111.11111111.11111111.11111111
c	Array	W: 11111111.11111111.11111111.11111111

Tabelle 5.3: Speicherzugriffsmuster zur Vectoraddition in Listing 5.1; R $\hat{=}$ Read, W $\hat{=}$ Write, 1 $\hat{=}$ Zugriff, 0 $\hat{=}$ Kein Zugriff, ? $\hat{=}$ Unbekannt

5.2 Die Benchmark Suite Rodinia

Rodinia [39] ist eine Benchmark Suite für heterogene Systeme mit OpenMP, OpenCL und CUDA Implementationen. Für die Evaluation lag die Version 2.4 mit insgesamt 18 OpenCL-Benchmarks vor. Eine Übersicht dazu bietet die Tabelle 5.7 am Ende des Kapitels. Es wurden alle 22 OpenCL-Kernel ausgewertet, wobei im Folgenden lediglich auf einzelne OpenCL-Kernel eingegangen wird, anhand derer die Probleme diskutiert werden. Der Benchmark leukocyte ist der Einzige, der in einem Laufzeitfehler endete. Für alle anderen Benchmarks konnte die Bibliothek ein Speicherzugriffsmuster ausgeben.

b+tree

```
offsetD[bid] = knodesD[offsetD[bid]].indices[thid];
```

Listing 5.2: Verschachtelte Arrays und Structs können nicht ausgewertet werden

Es wurde der Kernel *kernel_gpu_opencl.cl* ausgewertet. *kernel_gpu_opencl_2.cl* besitzt jedoch die gleichen Problemzonen, da sich der Code nur an einigen Stellen unterscheidet. Beide Kernel verwenden verschachtelte Arrays und Structs, wie in Listing 5.2 zu sehen ist. Die Bibliothek erkennt zwar, dass `knodesD` und `offsetD` Arrays sind, es werden aber nur vom letzteren die Zugriffe aufgezeichnet. Das Struct-Konstrukt wird komplett ignoriert und `offsetD[bid]` der Wert 0 zugewiesen.

```
for (i = 0; i < height; i++) {
        // Code
}
```

Listing 5.3: for-Schleife wird nicht richtig ausgewertet

Schleifen und If-Bedingungen wurden in der Bibliothek ebenfalls nicht implementiert. Diese werden einmalig ausgewertet, ohne ihre Funktionalität zu berücksichtigen. Aus der

```
i = 0;
i < size;
i++;
// Code
```

Listing 5.4: Auswertungsverlauf der for-Schleife in Listing 5.3

```
int variable = (i == 0) ? 0, i - 1;
```

Listing 5.5: Der bedingte Operator ? wurde nicht implementiert. Als Workaround wird die Bedingung immer als wahr ausgewertet. `variable` wird in diesem Beispiel also der Wert 0 zugewiesen.

Schleife in Listing 5.3 wird der Code wie in Listing 5.4 ausgewertet. Hierbei ist die fehlerhafte Inkrementierung von `i` zu beachten. Dies betrifft auch den bedingten Operator `?` in Listing 5.5, der zwar nicht in diesem, allerdings in einigen anderen Benchmarks verwendet wird. Es werden alle Ausdrücke lesend aufgerufen, jedoch wird als Workaround die Bedingung stets als wahr angenommen. Structs wurden ebenfalls in der Bibliothek nicht berücksichtigt und werden daher komplett ignoriert.

Tabelle 5.4 listet die Ergebnisse der Bibliothek auf. Hier ist anzumerken, dass für die Kernelfunktionsparameter `knodesD` und `recordsD` das „echte“ Zugriffsmuster nicht bestimmt werden konnte, da diese von verschachtelten Arrays und von Werten anderer Kernelfunktionsparameter abhängig sind. Für `currKnodeD` erkannte die Bibliothek den Lesezugriff nicht, was ebenfalls dem verschachtelten Arrayzugriff geschuldet ist. Zudem besitzen die Arrays `ansD`, `knodesD` und `recordsD` Structs als Arrayelement.

backprop

Wie im Benchmark b+tree beeinflusst auch in backprop ein Kernelfunktionsparameter den Arrayzugriff auf einen anderen Kernelfunktionsparameter. Sowohl `hidden_partial_sum` als auch `hid` sind im Codeabschnitt in Listing 5.6 Parameter der Kernelfunktion. Die Variable `hid` wird dabei zur Berechnung des Indexwertes für das Array `hidden_partial_sum` verwendet. Hier ist gut zu sehen, dass die Bibliothek falsche Ergebnisse liefern kann, da die Variable `hid` ein Kernelfunktionsparameter ist und somit undefiniert bleibt. Standardmäßig haben undefinierte Variablen den Wert 0, mit dem auch weiter gerechnet wird. Die Bibliothek gibt als Speicherzugriffsmuster für das Array `hidden_partial_sum` einen Schreibzugriff auf das erste Arrayelement an, die anderen Arrayelemente werden als nicht beschrieben angegeben, was jedoch nur der Fall bei `hid` mit dem Wert 0 ist.

Parameter	Typ	Zugriffsmuster
`height`	Variable	R
`knodesD` Korrektur	Array	R: ???? W: ????
`knodes_elem`	Variable	R
`recordsD` Korrektur	Array	R: ????
`currKnodeD` Korrektur	Array	W: 1111 R: 1111 W: 1111
`offsetD`	Array	R: 1111 W: 1111
`keysD`	Array	R: 1111
`ansD`	Array	W: 1111

Tabelle 5.4: Speicherzugriffsmuster der Benchmark b+tree; R $\hat{=}$ Read, W $\hat{=}$ Write, 1 $\hat{=}$ Zugriff, 0 $\hat{=}$ Kein Zugriff, ? $\hat{=}$ Unbekannt

```
if (tx == 0) {
        hidden_partial_sum[by * hid + ty] =
                          weight_matrix[tx * WIDTH + ty];
}
```

Listing 5.6: Der Kernelfunktionsparameter `hid` nimmt Einfluss auf das Zugriffsmuster des Kernelfunktionsparameters `hidden_partial_sum`

Zugleich werden abermals If-Statements und an anderer Stelle auch for-Schleifen im Kernelcode verwendet und somit nicht bei der Ermittlung des Zugriffsmusters berücksichtigt. Der Kernel wurde mit einer eindimensionalen `work_dim` ausgewertet, jedoch ist der Kernel auf eine zweidimensionale `work_dim` ausgelegt. Dies ist an den Work-Item-Funktionsaufrufen `get_group_id(1)` und `get_local_id(1)` zu erkennen. Der Funktionsparameter darf hierbei laut OpenCL-C-Spezifikation nur zwischen 0 und $get\_work\_dim() - 1$ liegen [4, S. 69]. Für Werte darüber hinaus wird der Wert 0 zurückgegeben. Dies hat zur Folge, dass die Bibliothek für das oben angesprochene Array `hidden_partial_sum` zufälligerweise das richtige Zugriffsmuster berechnet, da `by` und `ty` den Wert 0 haben. Auf eine Kernelanalyse mit einer zweidimensionalen `work_dim` wird hier allerdings wegen des zu hohen Umfangs verzichtet.

In Tabelle 5.5 kann das Ergebnis des Speicherzugriffsmusters für die Kernelfunktion `bpnn_layerforward_ocl` und in Tabelle 5.6 für die Kernelfunktion `bpnn_adjust`

_weights_ocl inklusive des echten Zugriffsmusters betrachtet werden. Die Parameter output_hidden_cuda in der Kernelfunktion bpnn_layerforward_ocl und in in beiden Kernelfunktionen bleiben innerhalb des Funktionscodes ungenutzt und werden daher von der Bibliothek als Typ None markiert. Zugleich konnte für einige Arrays nicht das korrekte Zugriffsmuster ermittelt werden, da deren Indizes von anderen Kernelparametern abhängig sind, deren Werte unbekannt sind.

cfd

Neben bereits bekannten Problemzonen macht der Kernel der Benchmark cfd Gebrauch von Hilfsfunktionen, die innerhalb des OpenCL-Kernels definiert sind. Die Bibliothek unterstützt diese Art von Funktionen nicht und weist somit allen Nicht-Kernelfunktionen – mit Ausnahme der Work-Item-Funktionen – den Rückgabewert 0 zu.

hotspot

In hotspot werden zweidimensionale Arrays verwendet, die in der Bibliothek nicht implementiert wurden. Ein Arrayaufruf temp_t[ty][tx] wird von der Bibliothek wie ein eindimensionales Array behandelt. Das heißt, der Aufruf wird intern wie temp_t[ty] behandelt.

In manchen Fällen kann es passieren, dass durch Rundungen von Gleitkommazahlen und anderen Maßnahmen negative Indizes entstehen. Diese werden vor dem Arrayaufruf abgefangen und durch 0 ersetzt.

nn

Der Umstand, dass Pointer- * und Adressen-Operatoren & vor den Variablen ignoriert werden, macht es in der Benchmark nn in Listing 5.7 nicht möglich, die Variable dist als Array zu identifizieren. Der Grund liegt darin, dass in der Bibliothek dist intern als eine Variable erkannt und in deren Datenstruktur festgehalten wird. Ein anschließender Arrayzugriff via Index, wie es in der Benchmark leukocyte der Fall ist, verursacht zwangsweise einen Laufzeitfehler, da zu diesem Zeitpunkt die Variable als Array erkannt und in der Datenstruktur der Arrays gesucht wird, in der sie nicht hinterlegt ist.

Parameter	Typ	Zugriffsmuster
`input_cuda`	Array	R: 01
`output_hidden_cuda`	None	
`input_hidden_cuda`	Array	R: 00000000.00001111.1111 W: 00000000.00001111.1111
Korrektur		R: ?... W: ?...
`hidden_partial_sum`	Array	W: 1
`input_node`	Array	R: 1 W: 1
`weight_matrix`	Array	R: 11111111.00000000.11111111 00000000.10000000.00000000 10000000.00000000.10000000 00000000.10000000.00000000 10000000.00000000.1 W: 11111111
Korrektur		R: 11111111.00000000.11111111 00000000.11111111.00000000 00000000.00000000.11111111 00000000.00000000.00000000 00000000.00000000.00000000 00000000.11111111 W: 11111111
`in`	None	
`hid`	Variable	R

Tabelle 5.5: Speicherzugriffsmuster der Kernelfunktion `bpnn_layerforward_ocl` der Benchmark backprop; R $\hat{=}$ Read, W $\hat{=}$ Write, 1 $\hat{=}$ Zugriff, 0 $\hat{=}$ Kein Zugriff, ? $\hat{=}$ Unbekannt

Parameter	Typ	Zugriffsmuster
`delta`	Array	R: 01111111.1
`hid`	Variable	R
`ly`	Array	R: 01
`in`	None	
`w`	Array	R: 01111111.10001111.1111
		W: 01111111.10001111.1111
Korrektur		R: ?1111111.1?...
		W: ?1111111.1?...
`oldw`	Array	R: 01111111.10001111.1111
		W: 01111111.10001111.1111
Korrektur		R: ?1111111.1?...
		W: ?1111111.1?...

Tabelle 5.6: Speicherzugriffsmuster der Kernelfunktion `bpnn_adjust_weights_ocl` der Benchmark backprop; R $\hat{=}$ Read, W $\hat{=}$ Write, 1 $\hat{=}$ Zugriff, 0 $\hat{=}$ Kein Zugriff, ? $\hat{=}$ Unbekannt

```
__global float *dist = d_distances + globalId;
*dist = (float) sqrt(
                (lat - latLong->lat)
                                * (lat - latLong->lat)
                + (lng - latLong->lng)
                                * (lng - latLong->lng));
```

Listing 5.7: Pointer- und Adressen-Operatoren werden während der Codeanalyse ignoriert

5.3 Zusammenfassung der Evaluation

Schleifen und If-Bedingungen werden in den Kernelcodes von Rodinia häufig verwendet. Zudem müssten Structs und Hilfsfunktionen in der Bibliothek implementiert werden, damit diese genauere Zugriffsmuster liefern kann. Im nächsten Kapitel wird auf diese Punkte detaillierter eingegangen und es wird diskutiert, an welchen Stellen in der Bibliothek nachgebessert werden muss, damit diese praxistauglich wird.

Obwohl neben den obigen Funktionalitäten auch viele andere fehlen, kann schnell erkannt werden, auf welche Arrays nur lesend oder schreibend zugegriffen wird. Somit müssen Arrays, auf denen die Bibliothek keinen Schreibzugriff erkennt, nicht von einem OpenCL-Gerät zurück zum Hostgerät kopiert werden, da der Coderumpf von Schleifen

als auch von If-Bedingungen voll ausgewertet wird, inklusive derer Bedingungen. Lediglich ihre Funktionalität wird dabei nicht beachtet. Diese Feststellung kann jedoch nur für Arrays gemacht werden; Integers bzw. einfache Variablen wie Booleans werden dabei genauso ignoriert wie Arrayzugriffe, die nicht über den Index, sondern über die Adresse geschehen, wie es in den Benchmarks nn und leukocyte der Fall ist.

Benchmark	LoC	Schleifen	If-Bedingungen	Struct-Aufrufe	Bedinge Operatoren	Verschachtelte Arrayaufrufe	Mehrdimensionale Arrays	Funktionsaufrufe	Laufzeit [ms]
b+tree									
kernel_gpu_opencl.cl	109	1	4	8	-	7	-	-	23
kernel_gpu_opencl_2.cl	111	1	7	12	-	12	-	-	25
backprop	90	1	4	-	-	-	-	-	28
bfs	50	1	3	3	-	-	-	-	25
cfd	280	2	3	162	-	-	-	14	55
gaussian	49	-	3	-	-	-	-	-	30
heartwall	2.235	103	82	-	-	-	-	-	35
hotspot	117	1	5	-	8	-	18	-	31
kmeans	56	3	2	-	-	-	-	-	24
lavaMD	284	5	2	20	-	-	-	-	27
leukocyte									
find_ellipse_kernel.cl	144	4	5	-	-	-	-	-	43
track_ellipse_kernel.cl	212	5	15	-	4	-	-	8	-
lud	163	6	5	-	-	-	-	-	52
myocyte	1.445	-	8	-	-	-	-	4	66
nn	22	-	1	4	-	-	-	-	18
nw	203	8	8	-	-	-	-	4	302
particlefilter									
particle_double.cl	316	9	22	-	-	1	-	9	48
particle_naive.cl	81	4	11	-	-	-	-	-	22
particle_single.cl	353	9	26	-	-	1	-	9	42
pathfinder	116	1	6	-	6	-	-	-	34
srad	341	5	19	-	-	-	-	-	35
streamcluster	68	2	3	3	-	-	-	-	39

Tabelle 5.7: In Version 2.4 beinhaltet Rodinia 18 OpenCL-Benchmarks mit insgesamt 22 OpenCL-Kernel. Bei der Erhebung der Anzahl der Schleifen, If-Bedingungen etc. wurden Macros wie `#ifdef` nicht berücksichtigt. Diese Statistik soll lediglich einen groben Überblick über die nicht implementierten Codekonstrukte liefern.

6 Zusammenfassung und Ausblick

Zur Ermittlung des Speicherzugriffsmusters wurde die statische Codeanalyse unter Verwendung der LLVM-Compiler-Infrastruktur gewählt. Auf Basis von Clang AST wird der Kernelcode sequentiell traversiert und die nötigen Informationen heraus gefiltert und verarbeitet. Dadurch kann der Verlauf jede einzelner Variable innerhalb einer Kernelfunktion erfasst und gegebenenfalls referenziert werden. Berechnungen können so direkt in Maschinencode mit Referenz zu den verwendeten Variablen festgehalten werden, ohne das konkrete Ergebnis bereits zu berechnen. Dies ist nötig, um in Nachhinein den Rückgabewert der Work-Item-Funktionen zu bestimmen, der zum Zeitpunkt der Codeanalyse unbekannt ist.

Nach der vollständigen Codeanalyse werden die Rahmenbedingungen für den OpenCL-Kernel gesetzt und somit auch die Referenzen zu den Work-Item-Funktionen. Der Rückgabewert der einzelnen Work-Item-Funktionen ist je nach Work-Item und Work-Group unterschiedlich. Indem das Zugriffsmuster der einzelnen Work-Items aufsummiert wird, kann das Speicherzugriffsmuster der Kernelfunktion und folglich auch des Kernels generiert werden. Das Speicherzugriffsmuster gibt an, welche Kernelfunktionsparameter während der Laufzeit gelesen und beschrieben werden und falls ein Parameter ein Array ist, auf welche Arrayelemente ein derartiger Zugriff stattfindet.

Die Evaluation hat gezeigt, dass nicht immer das korrekte Zugriffsmuster ermittelt werden konnte. Damit das Speicherzugriffsmuster richtig bestimmt werden kann, müssen neben den bereits implementierten Codekonstrukte Weitere folgen. Die fehlenden Funktionalitäten werden im nächsten Unterkapitel 6.1 aufgelistet.

6.1 Zukünftige Arbeiten

In dieser Arbeit wurden nur die wichtigsten Codekonstrukte berücksichtigt. Das Speicherzugriffsmuster könnte wesentlich deutlicher bestimmt werden, wenn bei der Codeanalyse weitere Punkte der OpenCL-Spezifikation wie Structs und Nichtkernelfunktionen, die innerhalb einer Kernelfunktion aufgerufen werden, berücksichtigt würden.

Des Weiteren werden die Bedingungen bzw. Conditions der If-Statements, Schleifen und des bedingten Operators `?` als auch der Codeblock einmal analysiert und in das Ergebnis

```
array[array[array[0]]];
function1(function2(function3(0)));
array[function(2)];
function(array[0]);
array[2][3]
```

Listing 6.1: Verschachtelte Array- und Funktionsaufrufe können nicht verarbeitet werden

mit aufgenommen. If-Statements, Schleifen und der bedingte Operator besitzen also keinerlei Funktionalität. Auch können verschachtelte Aufrufe wie in Listing 6.1 nicht verarbeitet werden. Durch interne Umstrukturierung der Array- und Funktionsverwaltung könnte dies in absehbarer Zeit implementiert werden. Ein weiteres Problem, das in der kurzen Zeit nicht gelöst werden konnte, ist das gleichzeitige De- bzw. Inkrementieren eines Arrayelements und des Indexwertes: `++array[++i]`.

```
__kernel function(__global float* array, int offset) {
        int i = get_global_id(0) + offset;
        array[i];
}
```

Listing 6.2: Indexwert des Arrays muss zurückverfolgt und auf Abhängigkeit mit Kernelfunktionsparametern geprüft werden

Wie sich in der Evaluation herausgestellt hat, beeinflussen Kernelfunktionsparameter wie in Listing 6.2 oftmals direkt den Indexwert eines Arrays. Hier sollte ein Mechanismus eingebaut werden, der dies erkennt und entsprechende Maßnahmen einleitet. Variablen, die zur Berechnung des Arrayindex verwendet werden und dieses Array zugleich ein Kernelfunktionsparameter ist, können durch zuvor gesetzte Abhängigkeiten zurückverfolgt werden. Sind nun diese Variablen von mindestens einem Funktionsparameter der Kernelfunktion abhängig, so wird ein solcher Funktionsparameter entsprechend markiert. Zum Teil setzt die Bibliothek solche Abhängigkeiten bereits für Variablen. Diese Maßnahme könnte weiterentwickelt werden, indem die API dahingehend erweitert wird, dass Kernelfunktionsparameter einen Wert zugewiesen bekommen können. Intern verarbeitet die Bibliothek den Variablenwert von Kernelfunktionsparametern bereits, jedoch wird bei unbekannten Variablen, was ein Funktionsparmeter aktuell darstellt, `0` zugewiesen und mit diesem Wert schließlich weiter gerechnet. Eine weitere Möglichkeit könnte so aussehen, dass verschiedene Werte für diese Funktionsparameter durchprobiert und die verschiedenen resultierenden Zugriffsmuster auf Unterschiede und Gemeinsamkeiten untersucht werden.

Neben Work-Item-Funktionen 4.3.2 gibt es noch Sub-Groups-Funktionen, die zwar nicht sehr häufig verwendet werden, jedoch für eine erfolgreiche Codeanalyse manchmal benö-

tigt werden. Diese wurden im Code bereits berücksichtigt, jedoch fehlt die Funktionalität, die diese Methoden wie die Work-Item-Funktionen nachahmt und den richtigen Rückgabewert berechnet. Daneben könnten noch weitere Built-in-Funktionen berücksichtigt und implementiert werden.

Intern arbeitet die Bibliothek ausschließlich mit Integerwerten, obgleich im Kernelcode Gleitkommazahlen oder Booleans verwendet wurden. Letztere zwei Datentypen können leicht nachimplementiert werden. Zugleich können Integers nicht wie momentan intern in der Bibliothek über `int` realisiert werden, sondern über den maschinennäheren Datentyp `size_t`. OpenCL verwendet beispielsweise diesen Datentyp für Indizes, weshalb Work-Item-Funktionen wie `get_global_id` den Datentyp `size_t` und nicht `uint` zurückgeben.

Die Bibliothek arbeitet unabhängig von LLVM sequentiell. Während der Evaluation machte sich dieses Detail bei der Durchführung zwar nicht bemerkbar, jedoch kann die Performance der Bibliothek weiter gesteigert werden, wenn der Code parallelisiert wird.

Literaturverzeichnis

[1] KHRONOS GROUP: *Connecting Software to Silicon*. https://www.khronos.org/, Februar 2014

[2] WIKIPEDIA: *OpenCL platform architecture*. http://de.wikipedia.org/w/index.php?title=Datei:Platform_architecture_2009-11-08.svg&filetimestamp=20130209093717&, Februar 2014

[3] *ISO C99*. : *ISO C99*. ISO/IEC 9899:TC3. http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, Februar 2014

[4] *The OpenCL Specification*. : *The OpenCL Specification*. Version 2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, November 2013

[5] WIKIPEDIA: *OpenCL memory model*. http://de.wikipedia.org/w/index.php?title=Datei:OpenCL_Memory_model.svg&filetimestamp=20130209093539&, Februar 2014

[6] LLVM: *The LLVM Compiler Infrastructure*. http://llvm.org/, Februar 2014

[7] NVIDIA DEVELOPER ZONE: *CUDA LLVM Compiler*. https://developer.nvidia.com/cuda-llvm-compiler, Februar 2014

[8] CLANG: *A C language family frontend for LLVM*. http://clang.llvm.org, Februar 2014. – 17.02.2014

[9] GCC: *the GNU Compiler Collection*. http://gcc.gnu.org/, Februar 2014

[10] CLANG 3.4 DOKUMENTATION: *Introduction to the Clang AST*. http://www.llvm.org/releases/3.4/tools/clang/docs/IntroductionToTheClangAST.html, Februar 2014

[11] CLANG: *AST Matcher Reference*. http://clang.llvm.org/docs/LibASTMatchersReference.html, Februar 2014

[12] CLANG 3.4 DOKUMENTATION: *Choosing the Right Interface for Your Application*. http://www.llvm.org/releases/3.4/tools/clang/docs/Tooling.html, Februar 2014

[13] CLANG API DOCUMENTATION: *libclang: C Interface to Clang*. http://clang.llvm.org/doxygen/group__CINDEX.html, Februar 2014

[14] CLANG 3.4 DOKUMENTATION: *Clang Plugins*. http://www.llvm.org/releases/3.4/tools/clang/docs/ClangPlugins.html, Februar 2014

[15] CLANG 3.4 DOKUMENTATION: *LibTooling*. http://www.llvm.org/releases/3.4/tools/clang/docs/LibTooling.html, Februar 2014

[16] CLANG 3.4 DOKUMENTATION: *Tutorial for building tools using LibTooling and LibASTMatchers*. http://www.llvm.org/releases/3.4/tools/clang/docs/LibASTMatchersTutorial.html, 2014

[17] CLANG 3.4 DOKUMENTATION: *JSON Compilation Database Format Specification*. http://www.llvm.org/releases/3.4/tools/clang/docs/JSONCompilationDatabase.html, Februar 2014

[18] GITHUB: *rizsotto/Bear*. https://github.com/rizsotto/Bear, Februar 2014

[19] NVIDIA DEVELOPER ZONE: *CUDA Zone*. http://developer.nvidia.com/object/cuda.html, Februar 2014

[20] *C++ AMP Specification*. : *C++ AMP Specification*. Version 1.0. http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf, August 2012

[21] MICROSOFT DEVELOPER NETWORK: *Compute Shader Overview - DirectCompute*. http://msdn.microsoft.com/en-us/library/ff476331.aspx, November 2013

[22] THE OPENMP API: *Specification for Parallel Programming*. http://openmp.org/, Februar 2014

[23] OPENACC: *Directives for Accelerators*. http://www.openacc.org, Februar 2014

[24] GRASSO, Ivan ; PELLEGRINI, Simone ; COSENZA, Biagio ; FAHRINGER, Thomas: LibWater: heterogeneous distributed computing made easy. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. New York, NY, USA : ACM, 2013 (ICS '13). – ISBN 978–1–4503–2130–3, 161–172

[25] MPI: *The Message Passing Interface*. http://www.mcs.anl.gov/research/projects/mpi, Februar 2014

[26] KIM, Jungwon ; SEO, Sangmin ; LEE, Jun ; NAH, Jeongho ; JO, Gangwon ; LEE, Jaejin: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: *Proceedings of the 26th ACM international conference on Supercomputing*. New York, NY, USA : ACM, 2012 (ICS '12). – ISBN 978–1–4503–1316–2, 341–352

[27] SNUCL: *An OpenCL Framework for Heterogeneous Clusters*. http://snucl.snu.ac.kr, Februar 2014

[28] KOFLER, Klaus ; GRASSO, Ivan ; COSENZA, Biagio ; FAHRINGER, Thomas: An automatic input-sensitive approach for heterogeneous task partitioning. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. New York, NY, USA : ACM, 2013 (ICS '13). – ISBN 978–1–4503–2130–3, 149–160

[29] INSIEME: *A source-to-source compiler for C/C++ that supports portable parallel abstractions for heterogeneous multi-core architectures.* http://insieme-compiler.org, Februar 2014

[30] KIM, Jungwon ; KIM, Honggyu ; LEE, Joo H. ; LEE, Jaejin: Achieving a single compute device image in OpenCL for multiple GPUs. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. New York, NY, USA : ACM, 2011 (PPoPP '11). – ISBN 978–1–4503–0119–0, 277–288

[31] MA, W. ; AGRAWAL, G.: A translation system for enabling data mining applications on GPUs. In: *Proceedings of the 23rd international conference on Supercomputing* ACM, 2009, S. 400–409

[32] GUMMARAJU, Jayanth ; MORICHETTI, Laurent ; HOUSTON, Michael ; SANDER, Ben ; GASTER, Benedict R. ; ZHENG, Bixia: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2010 (PACT '10). – ISBN 978–1–4503–0178–7, 205–216

[33] NEUSTIFTER, Andreas: *Efficient Profiling in the LLVM Compiler Infrastructure*, Institut für Computersprachen (Complang) Technischen Universität Wien, Diplomarbeit, April 2010

[34] POLLY: *LLVM Framework for High-Level Loop and Data-Locality Optimizations*. http://polly.llvm.org, Februar 2014

[35] GROUPS, Google: *Polly for GPU*. https://groups.google.com/d/msg/polly-dev/o9bqI-_XSQQ/IWDXVNUulr0J, Februar 2014

[36] SOURCEWEB: *A source code indexer and code navigation tool for C/C++ code.* https://github.com/rprichard/sourceweb, Februar 2014

[37] WOBOQ CODE BROWSER: *A web-based code browser for C/C++ projects.* http://woboq.com/codebrowser.html, Februar 2014

[38] *The OpenCL C Specification*. : *The OpenCL C Specification*. Version 2.0. http://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf, November 2013

[39] RODINIA: *Benchmark Suite, Version 2.4*. https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page, Mai 2014