TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Seniorprofessor Dr.-Ing. habil. Rainer G. Spallek

Diplomarbeit

Entwicklung eines SYCL-Backends für die Alpaka-Bibliothek und dessen Evaluation mit Schwerpunkt auf FPGAs

Jan Stephan

Geboren am: 8. Mai 1991 in Wilhelmshaven
Studiengang: Informatik
Matrikelnummer: 3755136
Immatrikulationsjahr: 2012

zur Erlangung des akademischen Grades

Diplom-Informatiker (Dipl.-Inf.)

Erstgutachter
Prof. Dr.-Ing. habil. Rainer G. Spallek

Zweitgutachter
Prof. Dr. rer. nat. Ulrich Schramm

Betreuer
Dr.-Ing. Oliver Knodel
Matthias Werner, M.Sc.

Eingereicht am: 16. Dezember 2019

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Seniorprofessor Dr.-Ing. habil. Rainer G. Spallek

Aufgabenstellung für die Anfertigung einer Diplomarbeit

Studiengang:	Informatik
Name:	Jan Stephan
Matrikelnummer:	3755136
Immatrikulationsjahr:	2012
Titel:	Entwicklung eines SYCL-Backends für die Alpaka-Bibliothek und dessen Evaluation mit Schwerpunkt auf FPGAs

Ziele der Arbeit

Alpaka ist eine plattformabstrahierende C++11-Bibliothek zur parallelen Programmierung von Multicore- und Manycore-Architekturen. SYCL liefert ein modernes C++11-Programmiermodell für OpenCL und ist aufgrund der zunehmenden Plattformunterstützung eine wünschenswerte Erweiterung für Alpaka. SYCL-Compiler erlauben u.a. Zugriff auf NVIDIA-GPUs (experimentell) und FPGAs (triSYCL).

Ziel dieser Arbeit ist es, SYCL als Backend-Variante für Alpaka zu implementieren und den Einsatz gängiger SYCL-Compiler hinsichtlich CPUs, GPUs und FPGAs an einer realen Alpaka-Anwendung zu evaluieren.

Schwerpunkte der Arbeit

- Literaturrecherche zu SYCL/OpenCL und FPGAs.
- Einarbeitung in die FPGA-Programmierung mittels des triSYCL- und Xilinx-SDAccel-Ökosystems.
- Untersuchung der Performance-Analysemöglichkeiten hinsichtlich der Nutzbarkeit und erreichten Leistung im Vergleich zu anderen Konzepten und Architekturen.
- Evaluierung des SYCL-Backends anhand eines in Alpaka entwickelten Verarbeitungsalgorithmus für Röntgenstrahlen-Pixeldetektordaten, u.a. hinsichtlich der erreichbaren Datenraten sowie der Nutzbarkeit von FPGAs.
- Zusammenstellung, Auswertung und Dokumentation der Ergebnisse.

Erstgutachter:	Prof. Dr.-Ing. habil. Rainer G. Spallek
Zweitgutachter:	Prof. Dr. rer. nat. Ulrich Schramm (HZDR)
Betreuer:	Dr.-Ing. Oliver Knodel (HZDR) Matthias Werner, M.Sc. (HZDR)
Ausgehändigt am:	15. April 2019
Einzureichen am:	16. Dezember 2019

Prof. Dr.-Ing. habil. Rainer G. Spallek
Betreuender Hochschullehrer

Zusammenfassung

Eine deutsche Zusammenfassung.

Abstract

An English summary.

Inhaltsverzeichnis

Glossar

CUDA GPGPU-Sprache der Firma NVIDIA

Device Alpaka- und SYCL-Bezeichnung für einen Beschleuniger

HIP *Heterogeneous-compute Interface for Portability*, von CUDA abgeleitete GPGPU-Sprache der Firma AMD

Host Alpaka- und SYCL-Bezeichnung für den Teil des Rechners, der den Kontrollfluss steuert

Initiation Interval Anzahl der Zyklen zwischen dem Ausführungsbeginn aufeinanderfolgender Schleifeniterationen

Kernel Programm, das auf einem Beschleuniger ausgeführt wird.

OpenCL *Open Computing Language*, offener Standard für die Programmierung von Beschleunigern

OpenMP *Open Multi-Processing*, offener Standard für die Shared-Memory-Programmierung

SPIR *Standard Portable Intermediate Language*, offene Zwischencode-Sprache

SYCL moderne C++-Abstraktionsschicht über OpenCL

TBB *Threading Building Blocks*, von der Firma Intel entwickelte Bibliothek für die parallele Programmierung von Mehrkern-CPUs

Akronyme

ALU *arithmetic logic unit*

API *application programming interface*

APU *accelerated processing unit*

BLAS *Basic Linear Algebra Subprograms*

CLB *configurable logic block*

CMT *clock management tile*

CPU *central processing unit*

DSP *digital signal processor*

FPGA *field programmable gate array*

GPGPU *general-purpose computing on graphics processing units*

GPU *graphics processing unit*

HPC *high"=performance computing*

II *initiation interval*

IOB *input/output block*

LUT Lookup-Tabelle

SLR *super logic region*

1. Einleitung

Die Hardware-Konfigurationen moderner Computersysteme sind in den letzten Jahren deutlich heterogener geworden. Mobiltelefone und PCs verfügen nicht nur über mehrkernige, leistungsstarke Prozessoren (*central processing unit* (CPU)), sondern auch über dedizierte Chips zur Grafikherstellung (*graphics processing unit* (GPU)). Dagegen verschwimmt bei Spielekonsolen und Laptops die Unterscheidung zwischen Haupt- und Grafikprozessoren in Form integrierter Chips, die beide Funktionalitäten performant abdecken können (*accelerated processing unit* (APU)). Im Bereich des *high"=performance computing* (HPC) sind verschiedene Beschleunigertypen seit jeher im Einsatz, seien es CPUs mit besonders vielen Kernen, von den GPUs abstammende Beschleuniger ohne grafische Ausgabe wie NVIDIAs Tesla-Reihe, oder Hybride wie Intels kurzlebiges Xeon-Phi-Projekt.

Allen gemeinsam ist die Idee der Beschleunigung eines Algorithmus durch dessen Parallelisierung, also die parallele Ausführung einzelner Schleifeniterationen auf den in den Beschleunigern vorhandenen Kernen, Multiprozessoren usw. Wie eine CPU sind die Beschleuniger allerdings darauf ausgelegt, von verschiedenen Algorithmen genutzt werden zu können, und entsprechend allgemein aufgebaut. Der Programmierer findet also möglicherweise eine Hardware-Plattform vor, die für seinen Algorithmus nicht ideal geeignet ist.

In dieses Feld stoßen seit einigen Jahren die Hersteller Intel und Xilinx mit programmierbarer Hardware (*field programmable gate array* (FPGA)) vor. Gegenüber den oben genannten Beschleunigern haben diese den Vorteil, dass die einzelnen Bestandteile der Hardware weitestgehend frei verschaltbar sind, wenngleich mit deutlich höherem Zeitaufwand. Dies ist vor allem für Programmierer interessant, die die Hardware für genau einen Zweck verwenden möchten. Lange Zeit stand dem jedoch die vergleichsweise schwierige Verwendung der FPGAs entgegen, da diese einen Schaltungsentwurf auf der Register-Ebene (im Gegensatz zur Programmierung auf der algorithmischen Ebene) benötigten. Durch das Aufkommen automatischer Synthese-Werkzeuge, die Algorithmen in Schaltungen umwandeln können, wurde der Entwicklungsprozess in jüngerer Zeit allerdings deutlich vereinfacht.

Ein weiteres Hindernis für die Entwicklung paralleler Programme ist die fehlende Portabilität der Werkzeuge: anfangs waren nur GPUs des Herstellers NVIDIA ohne Umwege über Schnittstellen zur Grafikdarstellung programmierbar, erforderten dafür aber die Nutzung der NVIDIA-eigenen CUDA-Schnittstelle, die mit Konkurrenzprodukten nicht kompatibel war. Für CPUs etablierte sich schnell die OpenMP-Schnittstelle als Mittel der Wahl für die Nutzung mehrerer Kerne. Die Vektorisierung hingegen erfordert noch immer entweder befehlssatzspezifische Erweiterungen (die auch innerhalb derselben Befehlssatzfamilie mitunter inkompatibel sind) oder sehr gut optimierende Compiler. Die Synthese-Werkzeuge der FPGA-Hersteller ermöglichten zwar die Nutzung der Hochsprachen, reicherten diese aber mit zahlreichen Erweiterungen an, was die Übertragung auf andere Plattformen erschwerte.

Um die Entwicklung portabler Programme zu ermöglichen, wurden früh verschiedene Ansätze entwickelt. Das Khronos-Industriekonsortium gab wenige Jahre nach der ersten CUDA-Veröffentlichung die Spezifikation der OpenCL-Schnittstelle heraus. Dahinter verbarg sich die Idee, eine einheitliche Schnittstelle für den Programmierer zu schaffen, die im Hintergrund von jedem Hardware-Hersteller für die eigenen Produkte implementiert und optimiert wird. Die OpenCL-Entwicklung wurde anfangs von zahlreichen namhaften Hard- und Software-Herstellern unterstützt (z.B. Apple, AMD, NVIDIA, Intel und Xilinx), flachte jedoch nach wenigen Jahren wieder ab und konnte sich in der GPU-Welt nicht gegen CUDA und auf CPUs nicht gegen OpenMP durchsetzen.

Ein anderer Ansatz liegt in der Entwicklung eines abstrakten Interfaces, das im Hintergrund auf die herstellereigenen Schnittstellen abgebildet wird. Ein solches Interface transformiert einen vom Programmierer entwickelten Quelltext in einen äquivalenten CUDA- oder OpenMP-Quelltext, ohne dass der Programmierer selbst weitere Anstrengungen in dieser Richtung unternehmen muss. So genügt ein einfacher Austausch der Zielplattform und eine erneute Kompilierung für die Nutzung eines anderen Hardware-Typs. Dieses Prinzip wird von mehreren parallelen Projekten verfolgt, wie etwa der vom Helmholtz-Zentrum Dresden-Rossendorf entwickelten Alpaka-Bibliothek oder der Kokkos-Bibliothek, die von einer Forschungseinrichtung des US-amerikanischen Energieministeriums stammt. Bisher bieten jedoch weder Alpaka noch Kokkos eine Unterstützung für FPGAs an.

Seit einigen Jahren versucht das Khronos-Konsortium erneut, einen Standard für die parallele Programmierung zu etablieren. Dieser SYCL genannte Ansatz basiert auf dem einige Jahre älteren OpenCL, bietet aber eine deutlich modernere und fortschrittlichere Programmierschnittstelle. Der neue Standard wird unter anderem von den FPGA-Herstellern Xilinx und Intel vorangetrieben und ist damit ein interessantes Implementierungsziel für die oben genannten Abstraktionsbibliotheken.

1.1. Forschungsstand

In den letzten Jahren befassten sich mehrere Forschungsgruppen mit der automatischen FPGA-Synthese für hochparallele Programme.

Schon 2009 veröffentlichten Papakonstantinou *et al.* einen Ansatz, der die Synthese einer Schaltung auf Basis der eigentlich für GPUs gedachten CUDA-Schnittstelle ermöglichte. [vgl. Pap+09]

Diese Arbeit konnte sich jedoch nicht langfristig durchsetzen. Seit der ersten Veröffentlichung einer OpenCL-Implementierung für FPGAs durch den Hersteller Altera (heute Intel) im Jahr 2013 verlagerte sich das Interesse der Forschung auf diese Plattform.

Eine der ersten Arbeiten in diesem Bereich wurde 2013 von Settle, einem damaligen Altera-Mitarbeiter, veröffentlicht. In ihr zeigte Settle den durch eine abstrakte Sprache wie OpenCL zu erreichenden Produktivitätsgewinn bei gleichzeitiger Beibehaltung der erreichten Leistung. [vgl. Set13]

Fifield *et al.* demonstrierten 2016 die Optimierung von OpenCL-Programmen für die im Vorjahr veröffentlichte Xilinx-OpenCL-Implementierung. [vgl. Fif+16]

Duarte *et al.* stellten 2018 das *hls4ml*-Projekt vor. Dabei handelt es sich um Implementierungen neuraler Netzwerke, die durch automatische Synthese in Schaltungen für Xilinx-FPGAs umgewandelt werden. In diesem Projekt kommt allerdings nicht OpenCL zum Einsatz, sondern die Xilinx-Erweiterungen für die Programmiersprache C++. [vgl. Dua+18]

Die SYCL-Spezifikation wurde in der Literatur in den ersten Jahren ihres Bestehens vornehmlich als einfacher Aufsatz für OpenCL betrachtet. Erst in jüngerer Zeit kam es zu eigenständigen Untersuchungen SYCLs.

Eine der frühen Arbeiten, die sich von dieser unscharfen Betrachtungsweise abhebt, ist

Žužeks Masterarbeit aus dem Jahre 2016, in deren Rahmen eine eigenständige SYCL-Implementierung entwickelt wurde. [vgl. Žuž16]

Wong *et al.*, Mitarbeiter der Firma Codeplay – einer der führenden Firmen im SYCL-Umfeld –, befassten sich ebenfalls 2016 mit den Wechselwirkungen zwischen dem C++-Standard und der auf diesem Standard aufbauenden SYCL-Spezifikation. Aufmerksamkeit wurde insbesondere den bei der Codeplay-SYCL-Implementierung aufgetretenen Probleme sowie Unzulänglichkeiten des C++-Standards zuteil. [vgl. Won+16]

Aliaga *et al.* veröffentlichten 2017 eine in C++ und SYCL geschriebene Implementierung der BLAS-Schnittstelle, die sie *SYCL-BLAS* nannten. [vgl. ARG17]

Burns *et al.* stellten 2019 das *SYCL-DNN*-Projekt vor, eine in C++ und SYCL geschriebene Bibliothek für die Beschleunigung von Operationen, die typischerweise in neuralen Netzwerken verwendet werden. Teil der Arbeit war auch ein Vergleich mit den konkurrierenden Bibliotheken cuDNN (NVIDIA) und MIOpen (AMD). Im Gegensatz zu diesen herstellerspezifischen Bibliotheken soll SYCL-DNN auf einer Vielzahl OpenCL-fähiger Beschleuniger lauffähig sein. [vgl. Bur+19]

Hinsichtlich der Verwendung von SYCL auf FPGAs ist in der Literatur – neben einigen weitestgehend inhaltsgleichen Vorträgen des Xilinx-Mitarbeiters Ronan Keryell – bisher nur der 2017 von Doumoulakis *et al.* veröffentlichte Artikel zu finden, der sich mit der Interoperabilität von SYCL und OpenCL auf Xilinx-FPGAs befasst. [vgl. DKO17]

Als Backend für Abstraktionsbibliotheken wie Alpaka oder Kokkos fand SYCL bisher keine Verwendung. Zwar veröffentlichten Copik *et al.* 2017 einen Artikel über die experimentelle Implementierung eines solchen Backends für die Kokkos und Alpaka sehr ähnliche Bibliothek *HPX.Compute*, bis heute ist dieser Entwicklungszweig aber nicht in das Hauptprojekt aufgenommen worden. [vgl. CK17]

Im Zusammenhang mit Kokkos findet SYCL bisher nur in Form einer von Hammond *et al.* 2019 durchgeführten Studie Erwähnung. Dabei handelt es sich jedoch um einen Vergleich der Programmiermodelle von Kokkos und SYCL und nicht um eine Implementierung eines Kokkos-Backends. [vgl. HKB19]

1.2. Ziel der Arbeit

In dieser Arbeit soll für die Alpaka-Bibliothek ein SYCL-Backend implementiert und ausgewertet werden. Aufgrund der bereits vorhandenen GPU-Backends erfolgt die Analyse vorrangig im Hinblick auf die Nutzbarkeit von FPGAs, womit aufgrund der während des Bearbeitungszeitruams verfügbaren und von SYCL unterstützten Hardware insbesondere FPGAs des Herstellers Xilinx gemeint sind.

2. FPGAs als Beschleuniger

Konzeption und Aufbau der FPGAs sowie der zugehörige Entwicklungsprozess werden in diesem Kapitel geschildert. Abschließend werden die auf FPGAs und GPUs zu findenden Parallelitätskonzepte miteinander verglichen.

2.1. Überblick

Für das Verständnis der Funktionsweise eines FPGAs ist es notwendig, die zugrunde liegenden Konzepte in Abgrenzung zu herkömmlicher Hardware darzustellen. Dieser Abschnitt definiert zunächst den FPGA-Begriff und erläutert im Anschluss daran den Aufbau moderner FPGA-Architekturen sowie traditionelle und neuartige Nutzungsmöglichkeiten dieses Hardware-Typus.

2.1.1. Definition

Field-programmable gate arrays sind, wie der Name andeutet, konzeptionell mit den *gate arrays* verwandt.

Die klassischen *gate arrays* sind eine Untergruppe der integrierten Schaltkreise (engl. *integrated circuits*, IC) und gehören zur Gattung der anwendungsspezifischen ICs (engl. *application specific IC*, ASIC). Unter ASICs versteht man jene Chips, die bereits bei der Herstellung mit einer kundenspezifischen Schaltung versehen werden. Innerhalb dieser Kategorie gehören *gate arrays* zu den teil-vorgefertigten ASICs (engl. *semi-custom ASIC*). Diese werden zunächst in großer Menge mit demselben technischen Grundgerüst produziert und erst in einem späteren Herstellungsschritt in kleineren Mengen mit kundenspezifischen Schaltungen versehen. Im Gegensatz zu ASICs, die von Anfang an nach Kundenwunsch hergestellt wurden (engl. *full-custom ASIC*), lässt sich so eine Reduktion der Produktionskosten erreichen. [vgl. KB13, S. 123]

Allerdings haben *gate arrays* den Nachteil, dass sie nur vom Hersteller programmiert werden können. Eine Anpassung der Schaltung im Feld (engl. *field-programmable*) ist damit nicht möglich. Mit FPGAs wurde dieses Problem in den 1980er Jahren gelöst, indem man aus Gattern (engl. *gates*) bestehende Logikzellen von geringer Komplexität in einer regelmäßigen Feldstruktur (engl. *array*) anordnete und über programmierbare Verdrahtungen miteinander verband. [vgl. KB13, S. 208]

Mittlerweile gibt es viele verschiedene FPGA-Varianten, die jedoch einige Gemeinsamkeiten aufweisen. FPGAs bestehen stets aus einem Feld aus Blockzellen, die so konfiguriert werden, dass sie eine bestimmte Funktion ausführen. Diese Blockzellen integrieren durch ein dichtes Verbindungsnetz Logikgatter und Speicher über ein dichtes Verbindungsnetz. Dabei lassen sich vier zentrale Strukturen unterscheiden:

- konfigurierbare Logikblöcke,
- programmierbare Verbindungen,
- Puffer für die Ein- und Ausgabe (E/A) und
- weitere Elemente (Speicher, arithmetische Einheiten, Taktnetzwerke, usw.).

In Abbildung 2.1 ist eine abstrakte FPGA-Struktur dargestellt, die aus Logikblöcken, Verbindungen, E/A-Puffern und speziellen Speicher- und Multiplizierer-Blöcken aufgebaut ist. [vgl. HS10, S. 10–13]

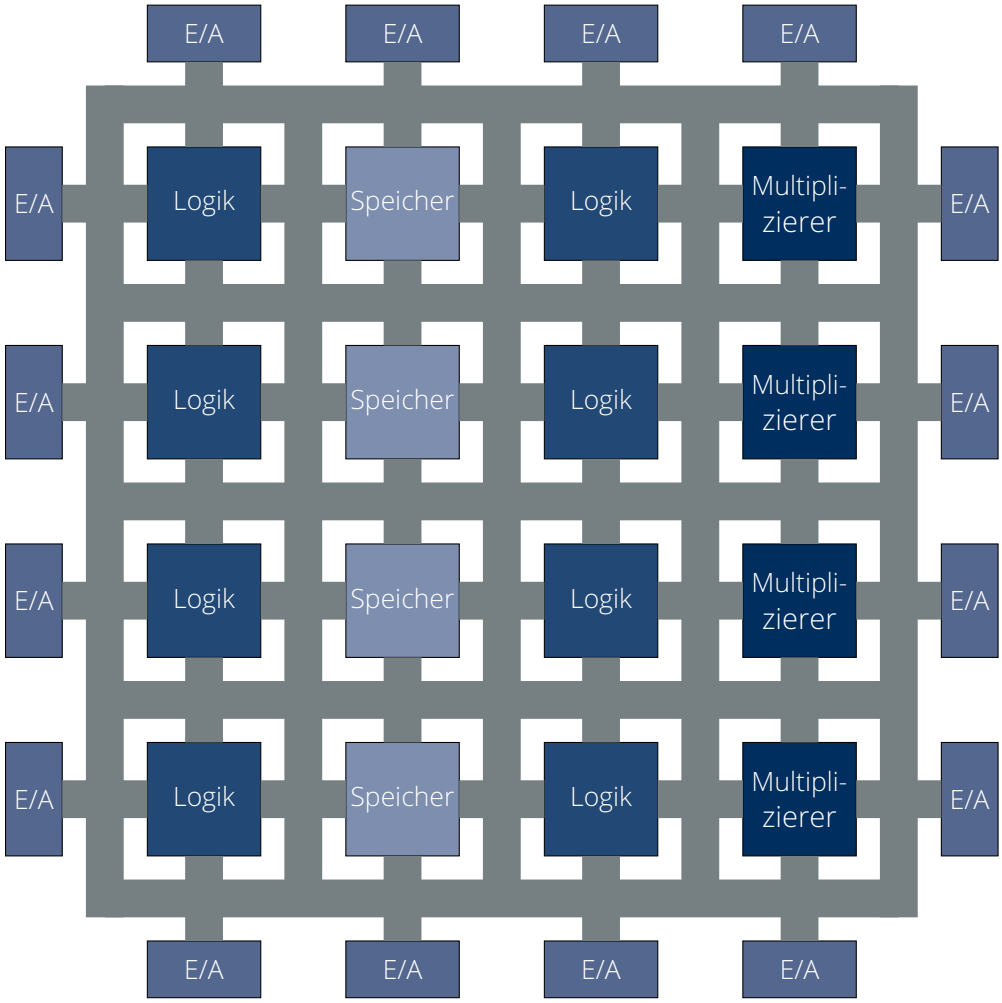

Abbildung 2.1.: abstrakter FPGA-Aufbau [nach HS10, S. 10–14]

2.1.2. Aufbau moderner FPGAs

Am Beispiel der Virtex-UltraScale+-Architektur der Firma Xilinx soll der Aufbau eines modernen FPGA verdeutlicht werden. FPGAs dieser Architektur bestehen aus sechs fundamentalen programmierbaren Elementen:

- Konfigurierbare Logikblöcke (engl. *configurable logic block* (CLB)) bestehen aus acht Logikeinheiten, die man als Lookup-Tabelle (LUT) bezeichnet und zur Generierung von Logikfunktionen verwendet werden können. Daneben sind in einem CLB Speicherelemente enthalten, die als Flipflop oder Latch verwendet werden können, sowie weitere Elemente wie Multiplexer oder Einheiten für den arithmetischen Übertrag. [vgl. Xil17, S. 6]

- Eingabe/Ausgabe-Blöcke (engl. *input/output block* (IOB)) werden zur Steuerung des Datenflusses zwischen den E/A-Pins und der internen Schaltkreise benutzt. Die UltraScale+-Architektur bietet verschiedene IOB-Typen, die z.B. verschiedene E/A-Standards oder uni- oder bidirektionale Kommunikation unterstützen. [vgl. die ausführliche E/A-Beschreibung in Xil19h, Kapitel 1 und 2]

- „Block RAM" kann bis zu 36 kbit speichern. Dabei lässt sich ein Block bei Bedarf auch in zwei unabhängige RAMs mit jeweils 18 kbit zerlegen. Zusätzlich sind in einem Taktschritt voneinander unabhängige Lese- und Schreibzugriffe möglich. Benachbarte Blöcke lassen sich darüber hinaus miteinander verbinden, um größere RAM-Bereiche zu generieren. [vgl. Xil19g, S. 6]

- UltraRAM-Blöcke können bis zu 288 kbit speichern, sind im Vergleich mit Block RAM aber unflexibler, da Lese- und Schreibzugriffe nicht parallel in einem Taktschritt möglich sind. Wie beim Block RAM lassen sich auch beim UltraRAM mehrere Blöcke zusammenschalten, um einen größeren Speicher zu erzeugen. [vgl. Xil19g, S. 92–94]

- Digitale Signalprozessoren (engl. *digital signal processor* (DSP)) sind Blöcke, die für die Ausführung fundamentaler mathematischer oder bitweiser Operationen der Signal-, Bild- und Videoverarbeitung besonders gut geeignet sind. Aus mehreren DSPs lassen sich durch Verbindungen komplexere Funktionen generieren. [vgl. Xil19f, S. 7–8]

- Blöcke für die Taktverwaltung (engl. *clock management tile* (CMT)) generieren den Takt für die restlichen Komponenten des FPGA. Sie sind ebenso dazu geeignet, Operationen auf einem von außen kommenden Takt durchzuführen, z.B. eine Phasenverschiebung oder eine Filterung. [vgl. Xil18a, S. 35–40]

Daneben können auf Beschleunigern, die mit einem FPGA ausgerüstet sind, noch weitere Komponenten hinzukommen. Ein Beispiel dafür ist der als DRAM bezeichnete Speicher, der mehrere GiB umfassen kann. Im Vergleich zu Block RAM und UltraRAM weist dieser Speichertyp aber deutlich geringere Speicherbandbreiten auf. Auf dem Beschleuniger *Alveo U200*, der mit einem UltraScale+-FPGA mit der Modellbezeichnung *XCU200* ausgestattet ist, finden sich beispielsweise vier DDR4-RAM-Module mit einer Bandbreite von 77 GiB s^{-1}. [vgl. Xil19a, S. 3; Xil18b, S. 2]

Ein XCU200-FPGA verteilt die oben genannten Elemente auf drei Abschnitte, die als *super logic region* (SLR) bezeichnet werden. Gemeinsam bilden die SLRs drei dynamische Regionen sowie eine statische Region, die alle mit dem DRAM des Beschleunigers verbunden sind (siehe Abbildung 2.2). Die dynamischen Regionen lassen sich vom Benutzer konfigurieren, während die statische Region der Laufzeitumgebung des FPGA-Host-Systems vorbehalten ist [vgl. Xil19a, S. 4]. Die Ressourcen verteilen sich in unterschiedlicher Anzahl auf die SLRs, wie Tabelle 2.1 zeigt.

Ressource	Gesamt	SLR0	SLR1	SLR2
CLB	111 500	45 625	20 250	45 625
Block RAM (36 KiB)	1766	695	376	695
UltraRAM (288 KiB)	800	320	160	320
DSP	5867	2275	1317	2275

Tabelle 2.1.: Ressourcen der dynamischen Regionen eines XCU200-FPGAs [siehe Xil19a, S. 5]

Innerhalb der SLRs sind die Ressourcen spaltenweise verteilt (wie in Abbildung 2.3 dargestellt). Zusätzlich werden die Spalten in vertikale Abschnitte von 60 CLBs bzw. der äquivalenten Anzahl der anderen Blocktypen unterteilt. Ein solcher Abschnitt bildet eine von Xilinx als *clock*

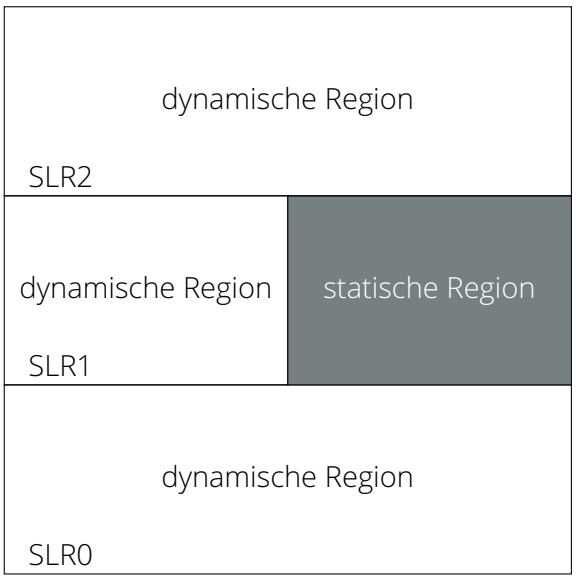

Abbildung 2.2.: Aufbau eines XCU200-FPGAs [nach Xil19a, S. 5]

region bezeichnete Struktur. Zusammengefasst ergibt sich dadurch eine spaltenorientierte Gitterstruktur, wie sie in Abbildung 2.4 zu sehen ist. [vgl. Xil19e, S. 22]

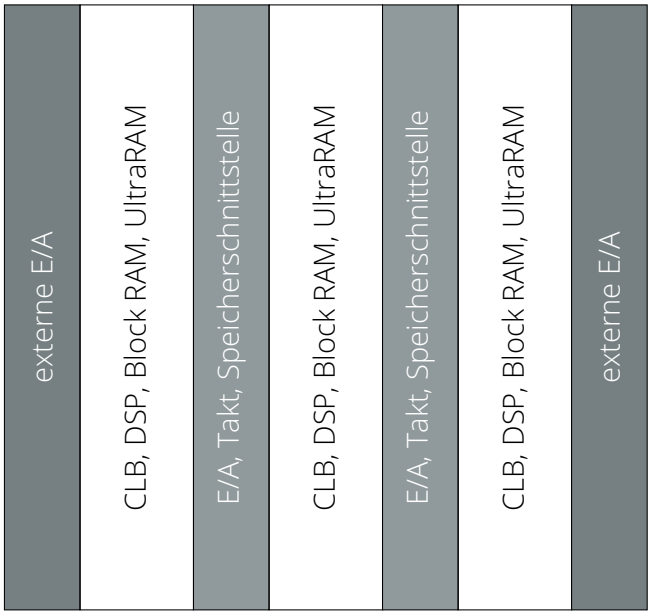

Abbildung 2.3.: spaltenweise Verteilung der FPGA-Ressourcen [nach Xil19e, S. 22]

2.1.3. Anwendungsfälle

Gegenüber ASICs bieten FPGAs einige Vorteile. Da sich Schaltungen ohne einen Produktionsprozess schneller in Hardware abbilden lassen, eignen sich FPGAs für die Entwicklung neuer Schaltungen durch die Methode des *rapid prototyping* und damit für eine schnellere Markteinführung. Durch die einfache Neuprogrammierung lassen sich Fehler außerdem während des Entwicklungsprozesses sowie während des Lebenszyklus des Produkts deutlich einfacher beheben, als dies bei ASICs der Fall wäre. [vgl. HS10, S. 10–1]

Dadurch eignen sich FPGAs sehr gut für den Einsatz als Schaltkreise, die in kleiner bis mitt-

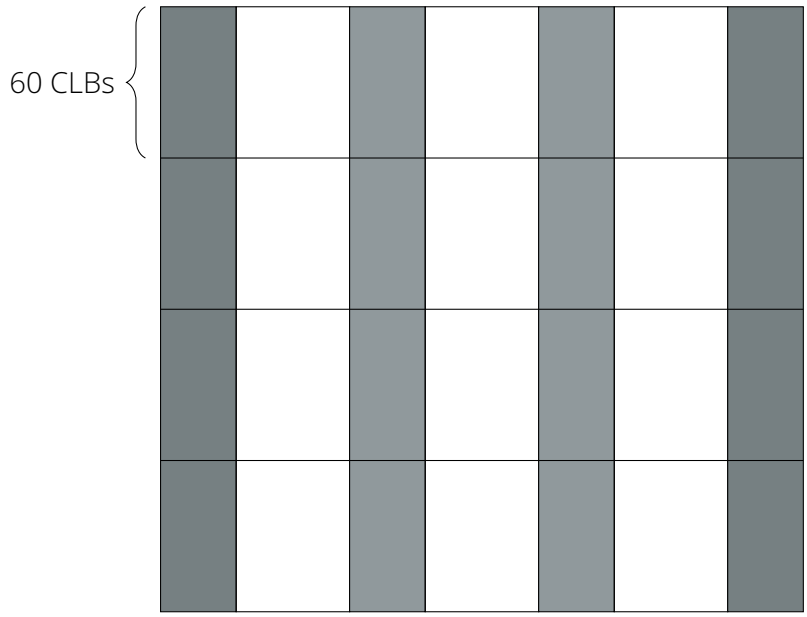

Abbildung 2.4.: Aufteilung der FPGA-Ressourcen auf *clock regions* [nach Xil19e, S. 22]

lerer Menge produziert werden sollen, weil die finanzielle Einstiegshürde deutlich geringer als bei ASICs ist. Umgekehrt sind ASICs bei hohen Produktionsvolumen überlegen, da die Kosten pro Chip geringer sind. [vgl. HS10, S. 10–2]

In jüngerer Zeit wurden FPGAs auch außerhalb des klassischen Schaltkreisentwurfs eingesetzt. So setzt die Firma Microsoft beispielsweise FPGAs des Herstellers Intel für die Inferenz tiefer neuraler Netzwerke [vgl. Fow+18; Chu+18] sowie als besonders schnelle Netzwerkkarten ein [vgl. Fir+18].

2.2. Entwicklungsprozess

Es sind bei der Software-Entwicklung für FPGAs zwei Vorgehensweisen voneinander abzugrenzen: einerseits die Entwicklung durch Hardware-Beschreibungssprachen (die in dieser Arbeit nur kurz skizziert werden) und andererseits die High-Level-Synthese, die auf in Hochsprachen implementierten Algorithmen basiert. Die Ansätze unterscheiden sich durch ihre Abstraktion der zugrunde liegenden Hardware (die verschiedenen Abstraktionsebenen wurden von Gajski in seinem Y-Diagramm zusammengefasst, siehe Abbildung 2.5). Die Hardware-Beschreibungssprachen befinden sich auf einem mittleren Niveau und Detaillierungsgrad, der Register-Transfer-Ebene, während die Hochsprachene auf der algorithmischen oder der Systemebene anzusiedeln sind. [vgl. KB13, S. 10–11]

2.2.1. Hardware-Beschreibungssprachen

Eine häufig verwendete Hardware-Beschreibungssprache ist VHDL, was für *VHSIC Hardware Description Language* steht, wobei VHSIC eine Abkürzung für *Very High Speed Integrated Circuit* ist. Die Sprache geht auf ein Programm des US-amerikanischen Verteidigungsministerium zurück, das in den 1980er Jahren eine einheitliche Beschreibungs- bzw. Dokumentationssprache für komplexe Schaltungen wünschte, und ist an die Programmiersprache Ada angelehnt. [vgl. KB13, S. 22]

Eine weitere bekannte Beschreibungssprache ist Verilog (*Verifying Logic*), die ebenfalls in den 1980er Jahren entwickelt wurde und an der Programmiersprache C orientiert ist. Sie wurde

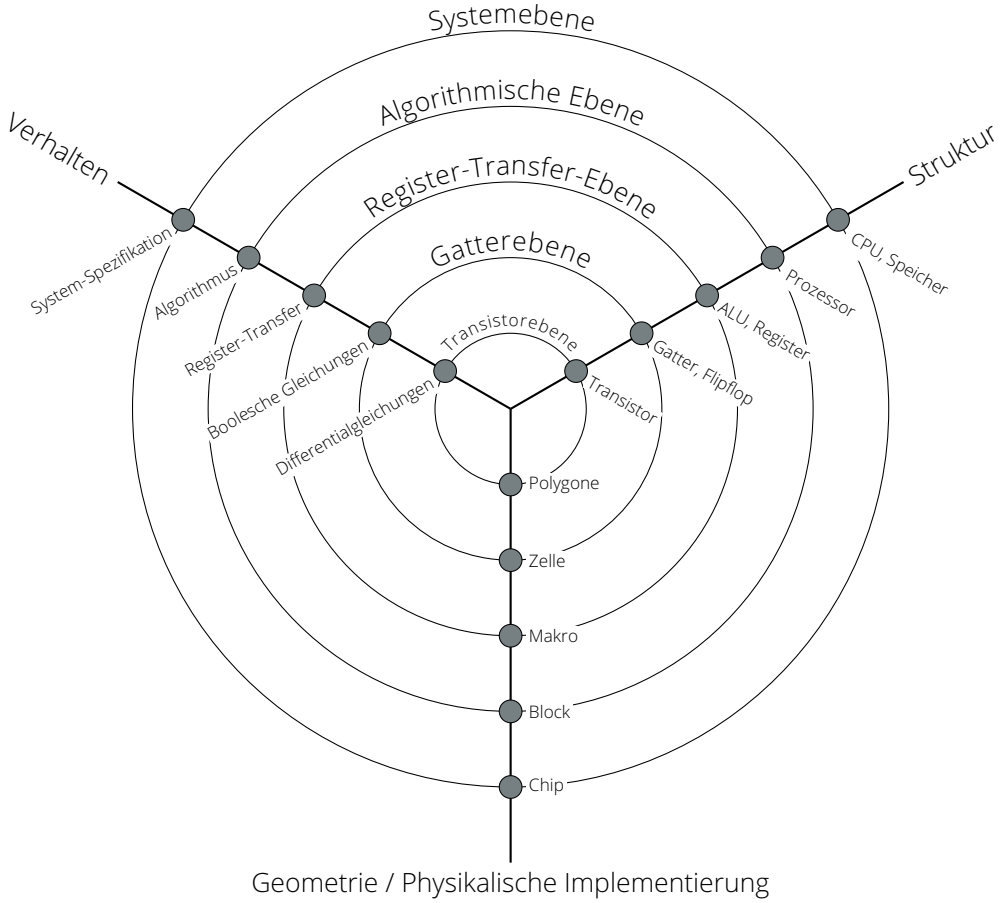

Abbildung 2.5.: Y-Diagramm nach Gajski [nach KB13, S. 10]

als ursprünglich proprietäre Sprache von der Firma *Gateway Design Automation* geschaffen und lässt sich in ihrem Umfang mit VHDL vergleichen. Verilog ist vor allem in den Vereinigten Staaten verbreitet, während europäische Entwickler eher auf VHDL setzen. In den folgenden Abschnitten werden die Konzepte der Beschreibungssprachen daher am Beispiel von VHDL erläutert. [vgl. KB13, S. 24–25]

Konzepte

VHDL ist eine Sprache für die Hardware-Modellierung und unterscheidet sich in einigen wichtigen Punkten von Hochsprachen wie C++. Stehen bei Hochsprachen die algorithmischen Aspekte im Vordergrund, ist bei VHDL entscheidend, wie der Algorithmus in eine Hardware-Beschreibung umgesetzt werden kann. Dabei wird die konzeptionelle Schaltung zunächst in mehrere *Komponenten* zerlegt. Die Beschreibung jeder *Komponente* erfolgt dann auf der Register-Transfer-Ebene, das heißt, dass zwischen speichernden (Register) und kombinatorischen (Transferfunktionen) Aspekten unterschieden wird. Aus der Digitaltechnik stammt eine Darstellungsweise als „endlicher Automat“ (siehe Abbildung 2.6): die Eingabe-Transferfunktion berechnet aus der Eingangsvariablen X und der Zustandsvariablen Z des Registers den neuen, im nächsten Takt zu übernehmenden Registerwert $Z+$. Sie ist damit eine boolesche Funktion:

$$Z+ = f(X, Z)$$

Die Ausgabe-Transferfunktion berechnet die Ausgabevariable Y aus Z und lässt sich ebenfalls als boolesche Funktion darstellen:

$$Y = g(Z)$$

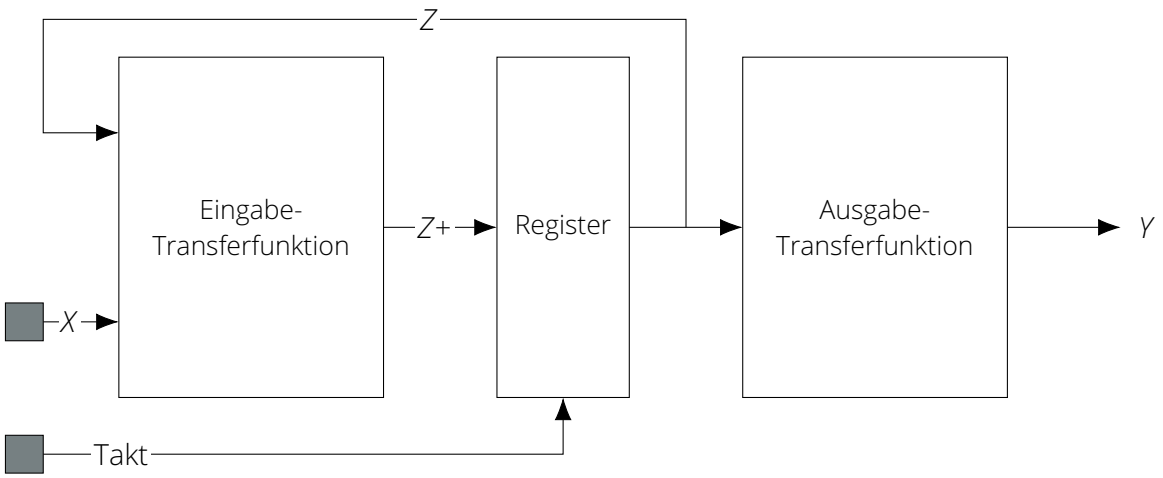

Abbildung 2.6.: Modell eines endlichen Automaten (Moore-Schaltwerk) [nach KB13, S. 35]

Diese Darstellung wird auch als Moore-Schaltwerk bezeichnet. [vgl. KB13, S. 34–35]

Die Anschlüsse einer einzelnen *Komponente* werden in VHDL als *Entity* bezeichnet. Sie sind notwendig, um mehrere *Komponenten* miteinander verschalten zu können. Der Quelltext 2.1 zeigt eine *Entity* für ein 2-Bit-Register. [vgl. KB13, S. 25]

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY reg2 IS
    PORT(
        clk  : IN  std_logic;
        d0   : IN  std_logic;
        d1   : IN  std_logic;
        load : IN  std_logic;
        res  : IN  std_logic;
        q0   : OUT std_logic;
        q1   : OUT std_logic
    );
END reg2 ;
```

Quelltext 2.1.: *Entity* eines 2-Bit-Registers [siehe KB13, S. 26]

Jeder *Entity* wird mindestens eine *Architecture* zugeordnet. Diese beschreibt entweder die innere Funktion, also das Verhalten, oder die Struktur, das heißt die Verschaltung von Teil-Komponenten. Eine *Architecture* ist also entweder eine Verhaltensbeschreibung oder eine Strukturbeschreibung. Strukturbeschreibungen können sowohl in Text- als auch in grafischer Form (mittels spezieller Schema-Editoren) angelegt werden, während Verhaltensbeschreibungen üblicherweise nur in Textform verfasst werden. [vgl. KB13, S. 27]

Einer *Entity* lassen sich auch mehrere unterschiedliche *Architectures* zuordnen. So lässt sich dieselbe *Entity* mit unterschiedlichem Verhalten oder internen Aufbau wiederverwenden. [vgl. KB13, S. 27]

Verhaltensbeschreibungen

Eine Verhaltensbeschreibung ist aus mehreren *Prozessen* aufgebaut; sie besteht dabei nicht aus weiteren (Teil-)Komponenten. Quelltext 2.2 zeigt eine solche Verhaltensbeschreibung für das in Quelltext 2.1 eingeführte 2-Bit-Register. Diese besteht aus zwei *Prozessen*, `reg` und `mux`.

Diese sind untereinander mit den internen Signalen q0_s, q0_ns, q1_s und q1_ns sowie nach außen über die (in Quelltext 2.1 deklarierten) Ports verbunden. [vgl. KB13, S. 29]

```
ARCHITECTURE beh OF reg2 IS
    SIGNAL q0_s, q0_ns, q1_s, q1_ns : std_logic;
BEGIN

    reg: PROCESS (clk, res)
    BEGIN
        IF res = '1' THEN
            q0_s <= '0';
            q1_s <= '0';
        ELSIF clk'event AND clk = '1' THEN
            q0_s <= q0_ns;
            q1_s <= q1_ns;
        END IF;
    END PROCESS reg;

    q0 <= q0_s AFTER 2 ns;
    q1 <= q1_s AFTER 2 ns;

    mux: PROCESS (load, q0_s, q1_s, d0, d1)
    BEGIN
        IF load = '1' THEN
            q0_ns <= d0 AFTER 3 ns;
            q1_ns <= d1 AFTER 3 ns;
        ELSE
            q0_ns <= q0_s AFTER 4 ns;
            q1_ns <= q1_s AFTER 4 ns;
        END IF;
    END PROCESS mux;

END beh;
```

Quelltext 2.2.: Verhaltensbeschreibung eines 2-Bit-Registers [siehe KB13, S. 28]

Der Prozess res reagiert auf die Ports clk und res: ist res logisch „1“ so werden die internen Signale q0_s und q1_s auf „0“ zurückgesetzt. Ist res logisch „0“ und existiert eine steigende Taktflanke (clk'event AND clk = '1'), werden die Werte von q0_ns und q1_ns übernommen. res entspricht damit dem *Register* des in Abbildung 2.6 dargestellten Moore-Schaltwerks. [vgl. KB13, S. 30–31]

Der Prozess mux beschreibt eine kombinatorische Funktion. Er nimmt die Signale load, q0_s, q1_s, d0 sowie d1 entgegen und gibt die Signale q0_ns und q1_ns aus. Ist load logisch „1“ werden d0 und d1 ausgegeben, ansonsten die gespeicherten Signale q0_s und q1_s. In Moore-Schaltwerk aus Abbildung 2.6 entspricht mux somit der Eingabe-Transferfunktion. [vgl. KB13, S. 31]

Die Ausgabe-Transferfunktion ist in diesem Beispiel als impliziter Prozess dargestellt: q0 und q1 sind die Ausgabe *Y* des Moore-Schaltwerks und wurden einfach mit den internen Signalen q0_s und q1_s verbunden. [vgl. KB13, S. 31]

Prozesse sind in VHDL als Modellierungen realer Hardware zu verstehen. So lässt sich für das oben entworfene 2-Bit-Register eine äquivalente Hardware-Schaltung aus zwei Multiplexern und zwei taktflankengesteuerten D-Flipflops mit asynchronen Set- und Reset-Eingängen aufbauen, wie Kesel und Bartholomä zeigen [siehe KB13, S. 32]. Die im Prozess reg verwendeten Signale q0_s und q1_s entsprechen dabei den Flipflops, während die Multiplexer den mux-Prozess implementieren. [vgl. KB13, S. 31]

Strukturbeschreibungen

Eine Strukturbeschreibung (auch Netzliste genannt) besteht aus mehreren Teil-Komponenten, die zu einer größeren, komplexeren Komponente zusammengeschaltet werden. So lässt sich für die aus den vorherigen Abschnitten bekannte `reg2`-*Entity* aus einem Modell eines Multiplexers sowie einem Modell eines Flipflops zusammensetzen (siehe Quelltext 2.3[1]).

```
ARCHITECTURE struct of reg2
    SIGNAL o1          : std_logic;
    SIGNAL o2          : std_logic;
    SIGNAL q0_internal : std_logic;
    SIGNAL q1_internal : std_logic;

    COMPONENT ff2
    PORT (
        clk : IN    std_logic;
        d0  : IN    std_logic;
        d1  : IN    std_logic;
        res : IN    std_logic;
        q0  : OUT   std_logic;
        q1  : OUT   std_logic
    );
    END COMPONENT;
    COMPONENT mux2
    PORT (
        a1  : IN    std_logic;
        a2  : IN    std_logic;
        b1  : IN    std_logic;
        b2  : IN    std_logic;
        sel : IN    std_logic;
        o1  : OUT   std_logic;
        o2  : OUT   std_logic
    );
    END COMPONENT;

BEGIN

    I1  : ff2
        PORT MAP(
            clk => clk, d0 => o1, d1 => o2,
            res => res, q0 => q0_internal, q1 => q1_internal
        );
    I0  : mux2
        PORT MAP(
            a1 => d0, a2 => d1, b1 => q0_internal,
            b2 => q1_internal, sel => load, o1 => o1, o2 => o2
        );

    q0 <= q0_internal;
    q1 <= q1_internal;

END struct;
```

Quelltext 2.3.: Strukturbeschreibung eines 2-Bit-Registers [siehe KB13, S. 36]

Wie bei der Verhaltensbeschreibung gibt es auch bei der Strukturbeschreibung Ports, das

[1] Die Verhaltensbeschreibungen der Komponenten `mux2` und `ff2` sind in Anhang A.1 zu finden.

heißt von außen ein- bzw. nach außen ausgehende Signale, sowie interne Signale für die Kommunikation der Teil-Komponenten untereinander. Die einzelnen Komponenten werden zunächst mit ihren lokalen Ports deklariert (`COMPONENT`-Abschnitte) und danach instanziiert; `I1` und `I0` sind dabei Bezeichnungen für eine *Instanz* der jeweiligen `Entity`. In der `PORT MAP` werden die Ports und Signale der Komponente den lokalen Ports der Teil-Komponenten zugeordnet. [vgl. KB13, S. 37]

Weitere Konzepte

Die hier vorgestellten Konzepte einer Hardware-Beschreibungssprache bilden nur einen sehr geringen Teil der zur Verfügung stehenden Möglichkeiten ab. Neben den gezeigten Grundbausteinen verfügen VHDL und Verilog noch über weitergehende Fähigkeiten, wie etwa Verzweigungen, Schleifen, Operatoren udn deren Überladung oder rudimentäre Objektorientierung. Aufgrund der Zielsetzung dieser Arbeit kann hier nicht weiter darauf eingegangen werden. Eine sehr gute Einführung in die Programmierung mit VHDL ist aber bei Kesel und Bartholomä zu finden [siehe KB13].

Nebenläufigkeit

Einer der wesentlichen Unterschiede zwischen VHDL und Hochsprachen, die auf der algorithmischen Ebene arbeiten, ist die von vornherein vorhandene Nebenläufigkeit. Während Befehle in C++ nacheinander abgearbeitet werden, gilt dies bei VHDL nur innerhalb eines *Prozesses*. *Prozesse* arbeiten unabhängig voneinander und werden nur über ihre Eingangssignale gesteuert, was sich so auch auf die reale Hardware abbilden lässt. [vgl. KB13, S. 25]

2.2.2. High-Level-Synthese und Parallelität

Aus den im vorherigen Abschnitt gezeigten Beispielen wird schnell ersichtlich, dass der Entwurf komplexerer Schaltungen mit einem hohen Konzeptions- und Arbeitsaufwand verbunden ist und nicht nur gute Kenntnisse des abzubildenden Algorithmus erfordert, sondern darüber hinaus auch Wissen über di Digitaltechnik erfordert. Mit der High-Level-Synthese (HLS) gibt es einen Ansatz, die konkrete Schaltung durch Abstraktion zu verbergen und sich dem Programmiermodell anderer Hardware-Typen (CPUs, GPUs) anzunähern. Bei der HLS findet der Entwurf vollständig auf der algorithmischen oder Systemebene statt. Anschließend wird aus dem erzeugten Modell durch HLS-Werkzeuge ein Modell auf der Register-Transfer-Ebene erzeugt, aus dem im nächsten Schritt die konkrete Schaltung synthetisiert werden kann. Auf herkömmliche Programmiersprachen übertragen entspricht der Entwurf mit einer Hardware-Beschreibungssprache also der Programmierung einer CPU mit einer Assemblersprache. [vgl. Xil19b, S. 7]

Während des algorithmischen Entwurfs kann der Nutzer zudem Randbedingungen einstellen, die die HLS steuern, wie z.B. den angestrebten Ressourcenbedarf oder den gewünschten Datendurchsatz. Bei der Erzeugung des Register-Transfer-Modells werden diese Randbedingungen dann von den HLS-Werkzeugen berücksichtigt. [vgl. KB13, S. 482]

Die HLS wird zur Zeit von einigen Herstellern für mehrere Programmiersprachen bzw. Erweiterungen bestehender Programmiersprachen unterstützt. Der Hersteller Xilinx bietet beispielsweise HLS-Werkzeuge für die Sprachen C, C++ und SystemC an und unterstützt darüber hinaus den OpenCL-Standard. Einen ähnlichen Weg geht die Firma Intel, die HLS-Werkzeuge für C und C++ sowie OpenCL anbietet.

Die folgenden Erläuterungen richten sich nach den Handbüchern des Herstellers Xilinx, die zugrunde liegenden Konzepte gelten jedoch für alle FPGAs.

Scheduling

FPGAs sind inhärent parallel. Anweisungen, die von einem bestimmten Hardware-Abschnitt ausgeführt werden (in VHDL durch *Prozesse* modelliert), sind sequentiell, während unterschiedliche Hardware-Abschnitte in Bezug aufeinander nebenläufig sind. Diese Eigenschaft unterscheidet FPGAs deutlich von Prozessorarchitekturen, wie das folgende Beispiel zeigt.

Jeder Prozessor führt ein Programm als eine Folge von Instruktionen aus. Diese Instruktionen werden von Compilern aus einer Hochsprache wie C++ in eine prozessorspezifische Assemblersprache übersetzt:

```
z = a + b;
```

Quelltext 2.4.: Addition in C++

wird wie folgt transformiert:

```
movl    -8(%rbp), %edx
movl    -4(%rbp), %edx
addl    %edx, %eax
movl    %eax, -12(%rbp)
```

Quelltext 2.5.: Addition in AMD64-Assembler

Selbst eine relativ simple Operation wie die Addition zweier Zahlen resultiert in vier sequentiell auszuführenden Maschineninstruktionen. Je nachdem, wo die Operanden liegen oder wo das Ergebnis gespeichert werden soll, können die Instruktionen für Laden und Speichern (im Verhältnis zur Addition) viele Taktzyklen benötigen. [vgl. Xil19b, S. 18]

Soll die Addition für viele Elemente ausgeführt werden, wie zum Beispiel in einer Schleife, müssen diese vier Instruktionen stets wiederholt werden, da sich alle Operationen dieselbe *arithmetic logic unit* (ALU) teilen.

Bei der HLS kann diese Operation auf Hardware abgebildet werden, die ausschließlich für diese Operation verwendet wird. Wenn a, b und z 32 bit groß sind, wird dieser Datentyp durch 32 LUT implementiert[2]. Im Gegensatz zu Prozessoren werden bei einer FPGA-Implementierung für jede Operation innerhalb des Algorithmus voneinander unabhängige LUT-Mengen instanziiert. [vgl. Xil19b, S. 19]

Eine Schleife ließe sich auf FPGAs also ganz oder teilweise ausrollen und, sofern zwischen den einzelnen Iterationen keine Abhängigkeiten bestehen, parallel ausführen. Die während der HLS vorgenommene Analyse der Daten- und Kontrollflussabhängigkeiten wird im FPGA-Kontext als Scheduling bezeichnet. [vgl. Xil19b, S. 19]

Pipelining

Das von Prozessoren bekannte Prinzip des *Pipelinings* findet bei FPGAs ebenfalls Anwendung. Dabei werden Datenabhängigkeiten so aufgeteilt, dass die ursprüngliche Verarbeitungsreihenfolge beibehalten wird, während die benötigten Hardware-Einheiten in eine Verkettung unabhängiger Stufen unterteilt werden. Jede Stufe erhält ihre Eingangsdaten von der vorherigen Stufe und reicht ihr Teilergebnis an die nächste Stufe weiter. Beispielsweise lässt sich die folgende Funktion auf einen Multiplizierer und zwei Addierer abbilden:

$$y = (a \cdot x) + b + c$$

Die resultierende Hardware-Schaltung ist in Abbildung 2.7 dargestellt. Die obere Schaltung berechnet y ohne Pipelining, die untere Schaltung zeigt die transformierte Schaltung mit Pipelining. Die grauen Kästen der unteren Schaltung entsprechen Registern, die in realer Hardware

[2] Ein LUT entspricht der Berechnung eines Bits.

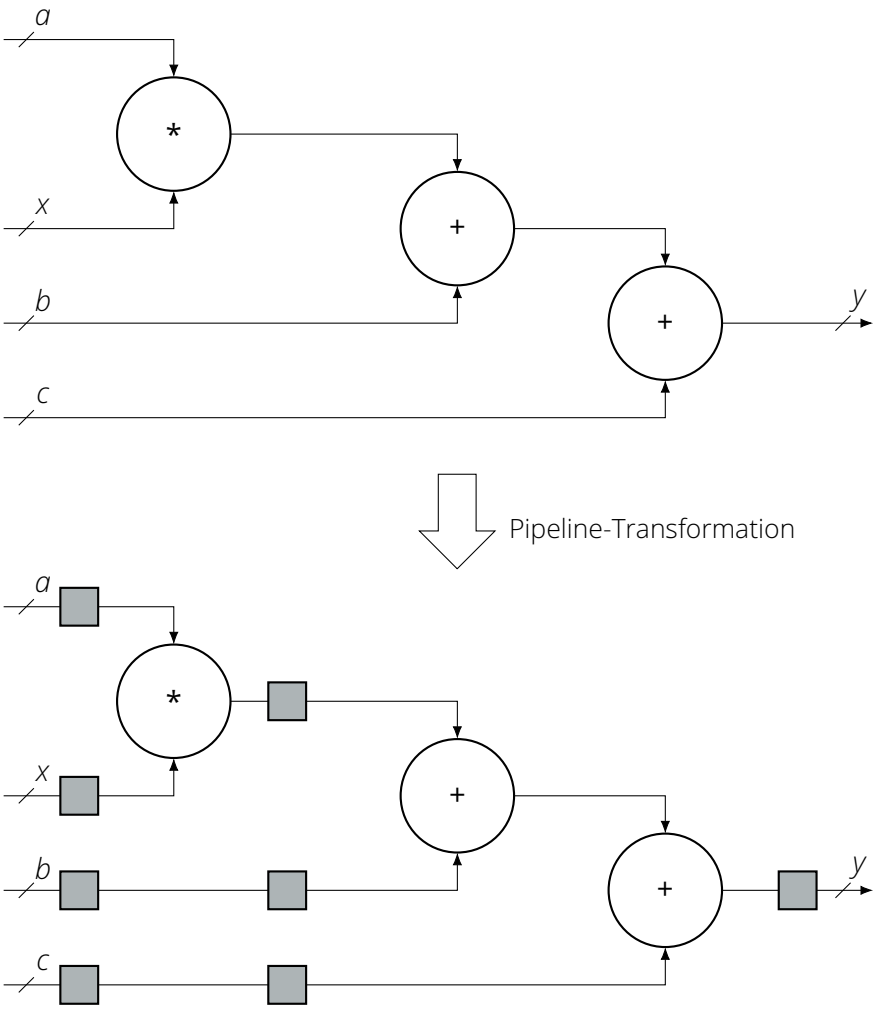

Abbildung 2.7.: FPGA-Implementierung einer mehrstufigen Berechnung [nach Xil19b, S. 21]

durch Flipflops realisiert werden. Jeder Kasten kann dabei als ein zusätzlicher Taktzyklus aufgefasst werden. Die Berechnung eines einzelnen Ergebnisses benötigt dadurch drei zusätzliche Taktzyklen. Durch die zusätzlichen Register lassen sich die einzelnen Schritte der Berechnung jedoch in unabhängige Abschnitte aufteilen, der Multiplizierer und die Addierer können also nebenläufig arbeiten. Die Ergebnisse y und y' lassen sich auf diese Weise (teilweise) parallel berechnen und lasten dabei die verfügbare Hardware besser aus: nach einem Overhead von 3 Taktzyklen zu Beginn der Berechnung (engl. *pipeline fill time*) berechnet die Schaltung in jedem weiteren Taktzyklus einen neuen Wert für y. Dieser Vorgang wird in Abbildung 2.8 illustriert. [vgl. Xil19b, S. 20–21]

Die Anwendung des Pipelining während der HLS unterscheidet sich zwischen den Herstellern: so versucht Intels OpenCL-Umgebung grundsätzlich, Pipelining auf jede vorhandene Schleife anzuwenden und erfordert ein explizites Deaktivieren des Verfahrens, wenn kein Pipelining gewünscht wird. Bei Xilinx' OpenCL-Umgebung verhält es sich dagegen genau umgekehrt, da hier Pipelining explizit angeschaltet werden muss.

Producer-Consumer

Neben dem Pipelining gibt es auf FPGAs eine weitere Möglichkeit, Parallelität auszudrücken. Diese ähnelt dem Pipelining-Prinzip, arbeitet jedoch auf einer groberen Ebene. Dahinter steht das Ziel, diese Funktionen weitestgehend parallel auszuführen. Dazu analysieren die HLS-

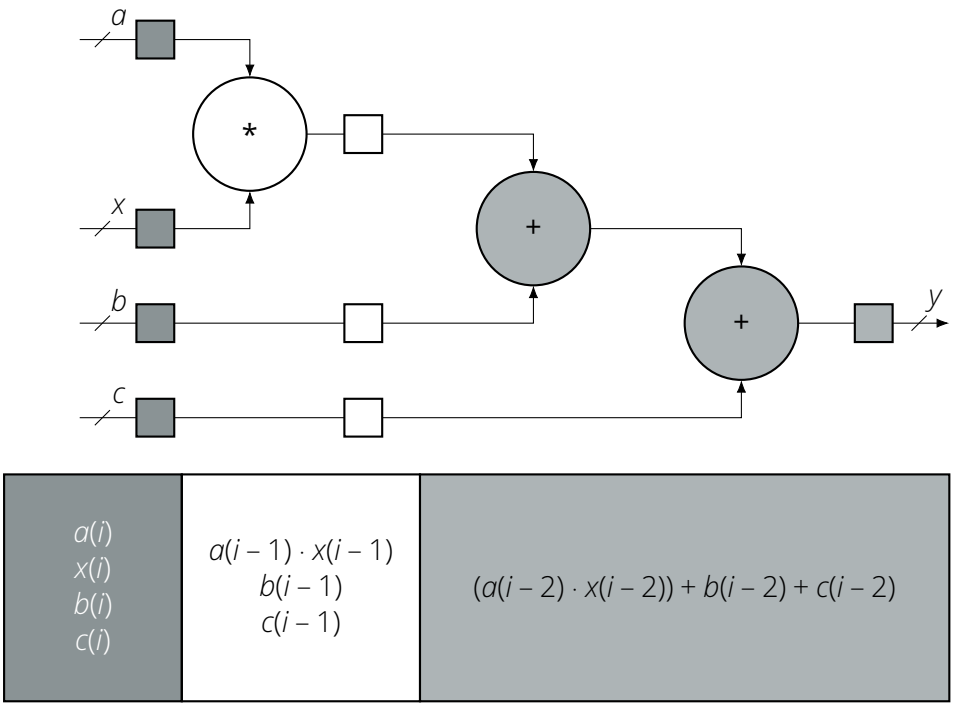

Abbildung 2.8.: FPGA-Pipeline-Architektur [nach Xil19b, S. 22]

Werkzeuge die Ein- und Ausgangsparameter der im Algorithmus vorhandenen Funktionen. Im einfachsten Fall gibt es keine gemeinsam bearbeiteten Daten, wodurch alle Funktionen parallel ausgeführt werden können. Naturgemäß ist dieser Fall recht selten, üblicherweise werden die Ergebnisse einer Funktion von einer oder mehreren nachfolgenden Funktionen verarbeitet. Dieses als *Producer-Consumer*-Prinzip bekannte Verfahren kann durch die Nebenläufigkeit der FPGA-Hardware parallelisiert werden. Durch den Einsatz von Block RAM als FIFO-Puffer zwischen den Hardware-Abschnitten, die aus der jeweiligen Funktion synthetisiert wurden, kann die produzierende Funktion während ihrer Ausführung Teilergebnisse speichern. Die konsumierende Funktion kann diese Teilergebnisse nach der initialen Wartezeit (engl. *initiation interval* (II)) direkt weiter verarbeiten, ohne auf das Ende der produzierenden Funktion bzw. das resultierende Gesamtergebnis warten zu müssen. [vgl. Xil19b, S. 22–23]

Das Producer-Consumer-Prinzip wird je nach Hersteller unterschiedlich benannt. So nennt sich dieses Verfahren bei Xilinx *Dataflow*, während es bei Intel *Channel* heißt.

3. Der SYCL-Standard

Die vor einigen Jahren veröffentlichte SYCL-Spezifikation bietet eine neue Möglichkeit, um ein Problem auf algorithmischer Ebene zu formulieren und dann über die HLS in eine FPGA-Schaltung zu synthetisieren. Eine einfache SYCL-Einführung sowie die für FPGAs wichtigen Besonderheiten sind daher das Thema dieses Kapitels.

3.1. Überblick

Mit dem SYCL-Standard[1] [vgl. KRH19] verfolgt die herausgebende Khronos-Gruppe das Ziel eines abstrakten C++-Programmiermodells für OpenCL, das die Modernität, Flexibilität und Einfachheit moderner C++-Standards bieten soll, während gleichzeitig Konzeption und Portabilität des OpenCL-1.2-Standards [vgl. Mun12] beibehalten werden.

Von OpenCL erbt SYCL damit den Anspruch, ein einheitliche Programmierschnittstelle für verschiedene Beschleunigertypen unterschiedlicher Hersteller zu bieten. Das heißt, dass ein einmal geschriebener Quelltext, der auf einem Beschleuniger ausgeführt werden soll, möglichst ohne große Änderungen sowohl auf einer CPU, einer GPU, einem DSP oder einem FPGA ausführbar sein soll.

SYCL unterscheidet sich von OpenCL im Hinblick auf die Struktur des Quelltextes: bei OpenCL sind die Quelltexte für das steuernde Programm (*Host*) und den Programmteil, der vom Beschleuniger (*Device*) ausgeführt wird, voneinander getrennt. Diese Design-Entscheidung des OpenCL-Standards ist durch das Ziel der Plattformunabhängigkeit begründet: ein Entwickler kennt während der Kompilierung des Hauptprogramms nicht notwendigerweise die vorhandenen Beschleuniger der Zielplattform. Dadurch wird der *Device*-spezifische Quelltext häufig erst zur Laufzeit des Programms kompiliert, da in diesem Moment der konkrete Beschleuniger bekannt ist. Dieser Ansatz hat jedoch den Nachteil, dass der Compiler des *Device*-Quelltexts (im Folgenden als *Kernel* bezeichnet) keine Annahmen mehr über das *Host*-Programm bzw. den den *Kernel* umgebenden Quelltext mehr treffen kann, was zu einem geringeren Optimierungspotential führt. OpenCL-Kernel lassen sich zwar auch vor der Laufzeit des Programms für eine konkrete Zielplattform kompilieren (dies geschieht aufgrund der langen Kompilierungszeiten bei FPGAs), büßen dadurch aber ihre Plattformunabhängigkeit ein.

Ein SYCL-Quelltext kennt dagegen keine strikte Trennung zwischen *Host*- und *Device*-Anweisungen. Neben der besseren Analyse des den *Kernel* umgebenden Kontexts hat dies den weiteren Vorteil, dass *Host* und *Device* Quelltext teilen können, wie etwa den beidseitig verwendeter Hilfsfunktionen. Der *Kernel* wird dabei vom *Device*-Compiler extrahiert und in eine Form umge-

[1] Entgegen des optischen Anscheins ist „SYCL“ kein Akronym, sondern ein Eigenname.

wandelt, die von der Ziel-Hardware zur Laufzeit kompiliert oder ausgeführt werden kann. [vgl KRH19, S. 35]

Ein weiterer wichtiger Unterschied zu OpenCL besteht darin, dass jedes SYCL-Programm mit einem Standard-C++-Compiler übersetzt werden kann, sofern keine direkten Interaktionen mit OpenCL selbst erfolgen. Damit lässt sich ein SYCL-Programm auf jeder CPU ausführen, für die ein (moderner) C++-Compiler existiert, wenngleich dies Einschränkungen bei der erreichbaren Leistung bedeuten kann. So schreibt die SYCL-Spezifikation für diesen Fall nur die Ausführbarkeit selbst vor, aber nicht die Nutzung aller CPU-Kerne oder SIMD-Register. [vgl. KRH19, S. 15]

Seit der ersten Veröffentlichung im März 2014 [vgl. Khr14] mit der Versionsnummer 1.2 wurde die SYCL-Spezifikation stetig weiterentwickelt; die zur Zeit aktuelle Spezifikation vom April 2019 trägt die Revisionsnummer 1.2.1 Revision 5. [vgl. KRH19, S. 1]

Teil der Khronos-Gruppe sind (unter anderen) die FPGA-Hersteller Xilinx und Intel. Xilinx unterstützt den SYCL-Standard in Form einer Erweiterung der bestehenden HLS-Werkzeuge bereits, während Intel dies für die eigenen FPGAs mittelfristig plant; für Intel-CPUs und -GPUs ist bereits eine SYCL-Implementierung verfügbar (der Abschnitt 3.3 befasst sich mit allen verfügbaren Implementierungen).

3.1.1. AXPY und SYCL

Ein im Bereich der Programmierung häufig verwendetes einführendes Beispiel ist der sogenannte AXPY-Algorithmus, der ursprünglich aus der Bibliothek *Basic Linear Algebra Subprograms* (BLAS) stammt [vgl. Law+79]. Dieser führt die Berechnung

$$\vec{y} = a \cdot \vec{x} + \vec{y}$$

aus und ist aufgrund seiner Einfachheit und hohen erreichbaren Parallelität (sofern $\vec{x}$ und $\vec{y}$ viele Elemente enthalten) sehr beliebt.

AXPY lässt sich auch für eine Einführung in SYCL gut verwenden und wird daher in den folgenden Abschnitten als illustrierendes Beispiel genutzt.

Ein SYCL-Programm lässt sich in mehrere aufeinander aufbauende Stufen unterteilen, wie in Quelltext 3.1 zu sehen ist. Die einzelnen Platzhalter im Quelltext werden in den nächsten Abschnitten mit Inhalt gefüllt.

```
#include <cstdlib>

#include <CL/sycl.hpp>

auto main() -> int
{
    // Beschleunigerwahl und Befehlswarteschlange

    // Speicherreservierung und -initialisierung

    // Kerneldefinition und -ausführung

    // Synchronisierung

    return EXIT_SUCCESS;
}
```

Quelltext 3.1.: Struktur eines SYCL-Programms

Beschleunigerwahl und Befehlswarteschlange

Von OpenCL erbt SYCL die Plattformunabhängigkeit. Es wird das Vorhandensein mindestens einer OpenCL-Plattform auf dem System angenommen[2], was im Umkehrschluss bedeutet, dass unter Umständen zwischen mehreren verschiedenen Plattformen konkurrierender Hersteller gewählt werden muss.

Die SYCL-Spezifikation bietet dem Programmierer mehrere Möglichkeiten, die gewünschte Plattform für sein Programm auszuwählen. Der einfachste Ansatz besteht darin, eine Befehlswarteschlange (die `queue` genannt wird) für einen Beschleuniger zu konstruieren. Über die Befehlswarteschlange erfolgt die Kommunikation zwischen dem *Host* und einem *Device*, also Kopieroperationen, das Starten eines *Kernels* sowie die Synchronisierung. Letztere ist notwendig, da es sich bei dem *Device* um eine vom *Host* weitestgehend unabhängige Hardware handelt, die Operationen also (aus Sicht des *Hosts*) asynchron ablaufen.

Jedes genutzte *Device* erhält in SYCL mindestens eine eigene `queue`, so dass auch die Nutzung mehrerer Beschleuniger möglich ist.

Eine `queue` kann durch das Übergeben eines Auswahlkriteriums für das gewünschte *Device* konstruiert werden. Die Auswahlkriterien werden in der SYCL-Spezifikation `device_selector` genannt. Neben den in der Spezifikation vorhandenen Kriterien (beispielsweise `cpu_selector`, `gpu_selector` oder `host_selector`) ist es Herstellern oder dem Programmierer selbst möglich, durch das Erben von der Elternklasse `device_selector` eigene Kriterien zu definieren. Beispielsweise findet sich in den Testfällen der von Xilinx entwickelten SYCL-Implementierung ein `device_selector` für die eigenen Geräte, der die FPGAs über den Firmennamen findet. Mit dessen Hilfe lässt sich die Befehlswarteschlange für einen Xilinx-FPGA wie in Quelltext 3.2 dargestellt erzeugen.

```
class XOCLDeviceSelector : public cl::sycl::device_selector {
    public:
        int operator()(const cl::sycl::device &Device) const override {
            const std::string DeviceVendor =
                            Device.get_info<cl::sycl::info::device::vendor>();
            return (DeviceVendor.find("Xilinx") != std::string::npos) ? 1 : -1;
        }
};

/* ... */

auto queue = cl::sycl::queue{XOCLDeviceSelector{}};
```

Quelltext 3.2.: Auswahl eines Xilinx-FPGAs und Erzeugung einer zugehörigen Befehlswarteschlange

Programmierern, die mehr Kontrolle über die Initialisierung des Beschleunigers oder der gesamten SYCL-Laufzeitumgebung wünschen, stellt die SYCL-Spezifikation das aus OpenCL bekannte Schema zur Verfügung. Der Programmierer kann zunächst eine Liste aller zur Verfügung stehenden OpenCL-Plattformen anfordern, aus denen er frei wählen kann. Auf dem Fundament der gewählten Plattform erzeugt der Programmierer im nächsten Schritt einen SYCL-Kontext (der einen OpenCL-Kontext kapselt). Der Kontext stellt wiederum eine Liste aller *Devices* der Plattform bereit, aus der ein oder mehrere Beschleuniger ausgesucht werden können. Die Auswahl dient dann als Parameter für die Erzeugung einer SYCL-Befehlswarteschlange. In jedem der genannten Schritte stehen dem Programmierer zahlreiche Informationen über die jeweilige Klasse zur Verfügung (Hersteller der Plattform oder des Beschleunigers,

[2] Aber nicht vorausgesetzt! Jede SYCL-Implementierung muss auch ohne eine OpenCL-Plattform auf dem Host lauffähig sein.

Hardware-Eigenschaften, verfügbare Erweiterungen, usw.), die eine Verfeinerung der Auswahl erlauben.

Der Quelltext 3.3 zeigt das ausführliche Schema, in den folgenden Abschnitten wird jedoch die oben gezeigte, einfachere und kürzere Variante verwendet.

```
auto platforms = cl::sycl::platform::get_platforms();
auto my_platform = /* ... */;

auto context = cl::sycl::context{my_platform};

auto devices = context.get_devices();
auto my_device = /* ... */;

auto queue = cl::sycl::queue{my_device};
```

Quelltext 3.3.: Ausführliche Beschleunigerwahl und Befehlswarteschlangen-Konstruktion

Speicherreservierung und -initialisierung

Für die Vektoren $\vec{x}$ und $\vec{y}$ ist eine Speicherreservierung auf dem *Device* sowie die Initialisierung des reservierten Speichers notwendig (die Konstante a kann als einfacher Parameter übergeben werden). SYCL stellt dafür zwei Klassen bereit:

- Ein `buffer` kapselt die auf dem *Device* reservierten Speicherbereiche. Dabei ist zu beachten, dass ein `buffer` keinem *Device* direkt zugeordnet ist, sondern dem gesamten Kontext zur Verfügung steht. Ein `buffer` kann so auch auf mehreren *Devices* verwendet werden, die notwendige Synchronisierung wird von der SYCL-Laufzeitumgebung vorgenommen.
- Ein `accessor` sorgt für den Zugriff auf den von einem `buffer` verwalteten Speicher. Es existieren verschiedene `accessor`-Typen, darunter auch einer für den Speicherzugriff auf der *Host*-Seite. Mit diesem lässt sich ein `buffer` direkt initialisieren, ohne eine explizite Kopie anstoßen zu müssen. Aus Sicht des Programmierers lässt sich ein `accessor` wie ein Zeiger oder Feld verwenden, d.h. über den [ ]-Operator.

Ein `buffer` besteht aus einer endlichen Anzahl von Elementen desselben Typs und kann ein-, zwei- oder dreidimensional sein. Der Elemente-Typ sowie die Dimension des Puffers sind als Template-Parameter zur Compile-Zeit anzugeben, während die Anzahl als Laufzeit-Parameter übergeben wird. Ein `accessor` lässt sich über die Methode `get_access` der `buffer`-Klasse erzeugen. Dabei wird als Template-Parameter der gewünschte Zugriffstyp angegeben. Diese Information wird von der SYCL-Laufzeitumgebung zur Sortierung der Abhängigkeiten zwischen Operationen auf dem *Device* genutzt (siehe auch Abschnitt 3.2.2).

SYCL unterscheidet sechs verschiedene Zugriffstypen:

- `read` gestattet ausschließlich lesenden Zugriff auf den Puffer.
- `write` ermöglicht ausschließlich schreibenden Zugriff.
- Durch `read_write` kann sowohl lesend als auch schreibend auf den `buffer` zugegriffen werden.
- `discard_write` ermöglicht ausschließlich schreibenden Zugriff und verwirft alle vorher im Puffer enthaltenen Elemente (also auch bei partiellem Zugriff).
- `discard_read_write` ist die Kombination aus `read_write` und `discard_write`.
- `atomic` ermöglicht atomaren Zugriff bei paralleler Nutzung des Puffers.

Für das AXPY-Beispiel lassen sich die benötigten Felder wie in Quelltext 3.4 dargestellt anlegen und initialisieren.

```
// Puffer enthalten 1024 Elemente
const auto buf_range = cl::sycl::range<1>{1024};

// erzeuge eindimensionale Puffer für int-Elemente
auto buf_x = cl::sycl::buffer<int, 1>{buf_range};
auto buf_y = cl::sycl::buffer<int, 1>{buf_range};

// greife auf x und y zu, verwirf vorherige Elemente
auto h_acc_x = buf_x.get_access<cl::sycl::access::mode::discard_write>();
auto h_acc_y = buf_x.get_access<cl::sycl::access::mode::discard_write>();

// initialisiere x und y
for(auto i = 0; i < 1024; ++i)
{
    h_acc_x[i] = /* ... */;
    h_acc_y[i] = /* ... */;
}
```

Quelltext 3.4.: Speicherreservierung und -initialisierung in SYCL

Kerneldefinition und -ausführung

Im nächsten Schritt wird der eigentliche Kernel definiert und ausgeführt. Ein SYCL-Kernel besteht aus zwei Teilen: der eigentlichen Kernel-Funktion, also der Abbildung des Algorithmus auf SYCL-C++-Quelltext, sowie den Abhängigkeiten (in Form von `accessor`-Variablen). Kernel-Funktion und Abhängigkeiten bilden gemeinsam eine *command group* und werden in dieser Form an die `queue` zur Ausführung übergeben. Dabei kann jede *command group* nur genau eine Kernel-Funktion (oder explizite Kopieroperation) enthalten. Es bildet sich damit für das AXPY-Beispiel das in Quelltext 3.5 gezeigte Grundgerüst einer *command group*, welche in diesem Fall als C++-Lambdafunktion notiert wird.

```
[&](cl::sycl::handler& cgh)
{
    auto d_acc_x = buf_x.get_access<cl::sycl::access::mode::read>(cgh);
    auto d_acc_y = buf_y.get_access<cl::sycl::access::mode::read_write>(cgh);

    // Kernel-Funktionsaufruf
}
```

Quelltext 3.5.: Struktur einer *command group*

SYCL bietet für verschiedene Anwendungsfälle unterschiedliche Methoden für den Aufruf der Kernel-Funktion. Für datenparallele Algorithmen bietet sich vor allem der Aufruf mittels der Methode `parallel_for` an. Diese entspricht dem aus OpenCL bekannten *NDRange*-Kernel und nutzt die SIMD-Eigenschaften der zur Verfügung stehenden Beschleuniger. Auf CPUs können so durch einen Aufruf mehrere Kerne und deren SIMD-Register genutzt werden oder auf einer GPU die Multiprozessoren und SIMD-Einheiten. Auf einem FPGA kann durch `parallel_for` ebenfalls eine SIMD-Schaltung synthetisiert werden.

Für aufgabenparallele Algorithmen steht in SYCL der Aufruf `single_task` zur Verfügung, was einem *Task*-Kernel in OpenCL entspricht. Dieser wird z.B. auf einer CPU nur auf einem einzelnen Kern ausgeführt. Mehrere Kernel dieses Typs lassen sich dann parallel auf den vorhandenen Kernen ausführen.

Für das AXPY-Beispiel bietet sich der datenparallele Fall an, weshalb die Kernel-Funktion wie in Quelltext 3.6 gezeigt aufgerufen wird. Die 1024 Elemente der Vektoren werden dabei als Arbeitsgröße mit übergeben. Die SYCL-Laufzeitumgebung generiert daraus einen Ausführungsraum mit 1024 *work-items*, einer Abstraktion der zugrundeliegenden Hardware-Features (SIMD-Register, Threads, usw.). Der Index des jeweiligen *work-items* wird als Parameter an die Kernel-Funktion übergeben und kapselt einen Index, der für den Zugriff auf ein Element im Speicher verwendet werden kann.

```
[&](cl::sycl::handler& cgh)
{
    auto d_acc_x = buf_x.get_access<cl::sycl::access::mode::read>(cgh);
    auto d_acc_y = buf_y.get_access<cl::sycl::access::mode::read_write>(cgh);

    cgh.parallel_for<class axpy>(cl::sycl::range<1>{1024},
    [=](cl::sycl::item<1> work_item)
    {
        auto idx = work_item.get_id();
        d_acc_y[idx] = a * d_acc_x[idx] + d_acc_y[idx];
    });
}
```

Quelltext 3.6.: Struktur einer *command group* mit Kernel-Aufruf

Synchronisierung

Nachdem der Kernel an die Befehlswarteschlange übergeben wurde, muss das Ergebnis überprüft werden. Um darauf zugreifen zu können, ist zunächst die Synchronisierung von *Host* und *Device* erforderlich, da beide asynchron zueinander arbeiten. Die `queue` verfügt jedoch über die Methode `wait`, die den *Host* so lange warten lässt, bis alle bis zu diesem Zeitpunkt eingereihten Befehle abgearbeitet wurden. Dies ist in Quelltext 3.7 dargestellt. Anschließend lassen sich die Elemente des Vektors $\vec{y}$ auf der *Host*-Seite über den während der Initialisierung der Puffer angelegten `accessor` überprüfen.

```
queue.wait();
```

Quelltext 3.7.: Struktur einer *command group* mit Kernel-Aufruf

Zusammenfassung

Das gesamte SYCL-AXPY-Beispiel findet sich in Quelltext 3.8, einschließlich einiger kleinerer, in den vorigen Abschnitten ausgelassener, Details.

```
#include <cstdlib>
#include <CL/sycl.hpp>

class XOCLDeviceSelector : public cl::sycl::device_selector {
    public:
        int operator()(const cl::sycl::device &Device) const override {
            const std::string DeviceVendor =
                            Device.get_info<cl::sycl::info::device::vendor>();
            return (DeviceVendor.find("Xilinx") != std::string::npos) ? 1 : -1;
        }
};

auto main() -> int {
    constexpr auto a = 42;

    // Beschleunigerwahl und Befehlswarteschlange
    auto queue = cl::sycl::queue{XOCLDeviceSelector{}};

    // Speicherreservierung und -initialisierung
    const auto buf_range = cl::sycl::range<1>{1024};

    auto buf_x = cl::sycl::buffer<int, 1>{buf_range};
    auto buf_y = cl::sycl::buffer<int, 1>{buf_range};

    auto h_acc_x = buf_x.get_access<cl::sycl::access::mode::discard_write>();
    auto h_acc_y = buf_x.get_access<cl::sycl::access::mode::discard_write>();

    for(auto i = 0; i < 1024; ++i)
    {
        h_acc_x[i] = /* ... */;
        h_acc_y[i] = /* ... */;
    }

    // Kerneldefinition und -ausführung
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto d_acc_x = buf_x.get_access<cl::sycl::access::mode::read>(cgh);
        auto d_acc_y = buf_y.get_access<cl::sycl::access::mode::read_write>(cgh);

        cgh.parallel_for<class axpy>(cl::sycl::range<1>{1024},
        [=](cl::sycl::item<1> work_item)
        {
            auto idx = work_item.get_id();
            d_acc_y[idx] = a * d_acc_x[idx] + d_acc_y[idx];
        });
    });

    // Synchronisierung
    queue.wait();

    return EXIT_SUCCESS;
}
```

Quelltext 3.8.: AXPY – vollständiges SYCL-Beispiel

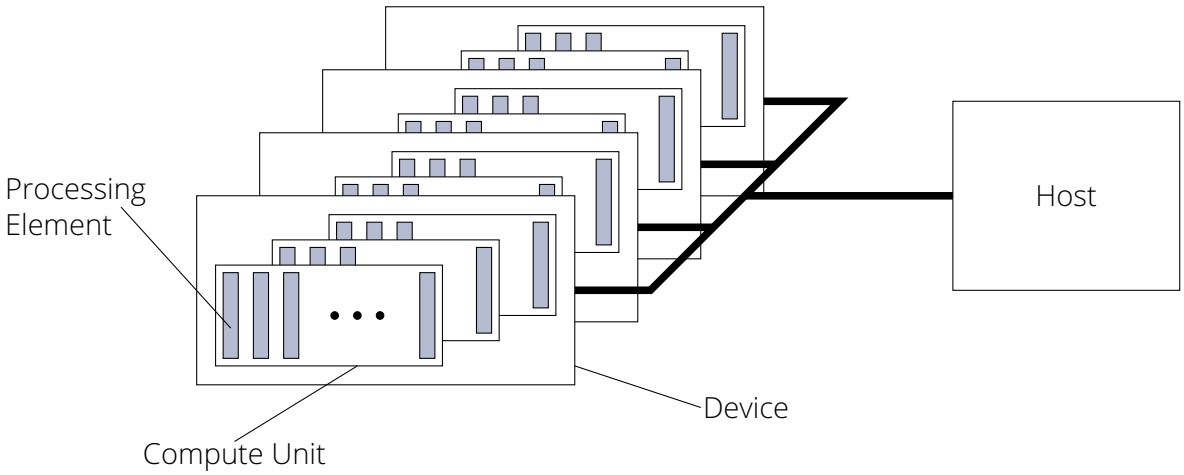

Abbildung 3.1.: SYCLs Plattform-Modell [nach Mun12, S. 23]

3.2. Weiterführende Konzepte

Die Entwicklung komplexerer Programme mit SYCL erfordert die Kenntnis einiger weiterer Konzepte, die in der obigen Einführung nicht berücksichtigt wurden. Dazu zählen die in SYCL vorhandene Hardware-Abstraktion sowie die Abhängigkeiten zwischen Kerneln. Für die Analyse des entwickelten Programms sind außerdem SYCLs Fähigkeiten zur Fehlerbehandlung und zum Profiling relevant.

3.2.1. Hardware-Abstraktion

Um eine bessere Anpassung des Programms auf die genutzte Hardware zu ermöglichen, ohne die Plattformunabhängigkeit aufzugeben, führte die OpenCL-Spezifikation eine Reihe von Abstraktionen ein. Diese entsprechen konzeptionell den in der Hardware vorhandenen Fähigkeiten und wurden ebenfalls von SYCL übernommen.

Eine *Plattform* ist in OpenCL und SYCL aus dem *Host* und mindestens einem *Device* zusammengesetzt. Jedes *Device* besteht wiederum aus mindestens einer *compute unit* (CU). Eine CU lässt sich auf einen oder mehrere Teile des Beschleunigers abbilden und ist in der Lage, einen Kernel auszuführen. Bei einer CPU lässt sich eine CU also auf einen Kern abbilden oder bei einer GPU auf einen Multiprozessor. Bei FPGAs ist die Abbildung dynamischer: hier hängt die Zahl der verfügbaren CUs davon ab, wie viele Ressourcen der Kernel verbraucht. Die Zahl der gleichzeitig platzierbaren Kernel entspricht damit der Zahl der möglichen CUs, sofern die Implementierung keine Obergrenze für die CU-Anzahl vorgibt. Eine CU besteht aus mindestens einem *processing element* (PE). Ein PE lässt sich dabei als Abstraktion der SIMD-Fähigkeiten einer CU verstehen, also z.B. als ein Element innerhalb eines SIMD-Vektorregisters. Die Abbildung 3.1 veranschaulicht dieses Modell.

Um die Parallelität mehrerer CUs nutzen zu können, ist es sinnvoll, die Arbeit des Kernels aufzuteilen. Bei acht verfügbaren CUs wäre es daher wünschenswert, die Berechnungen in mindestens acht Blöcken (oder einem Vielfachen davon) parallel durchzuführen. Diese Aufteilung wird in OpenCL und SYCL *work-group* genannt. *Work-groups* werden durch ihre Zuordnung zu unterschiedlichen CUs asynchron zueinander ausgeführt und eine Synchronisierung der Gruppen ist nicht ohne weiteres möglich.

Eine *work-group* besteht aus mindestens einem *work-item*, wobei die Implementierung auch eine maximale Anzahl von *work-items* festlegen kann. Ein *work-item* wird während der Ausführung einem PE zugeteilt. *Work-items* werden zueinander asynchron ausgeführt, lassen sich je-

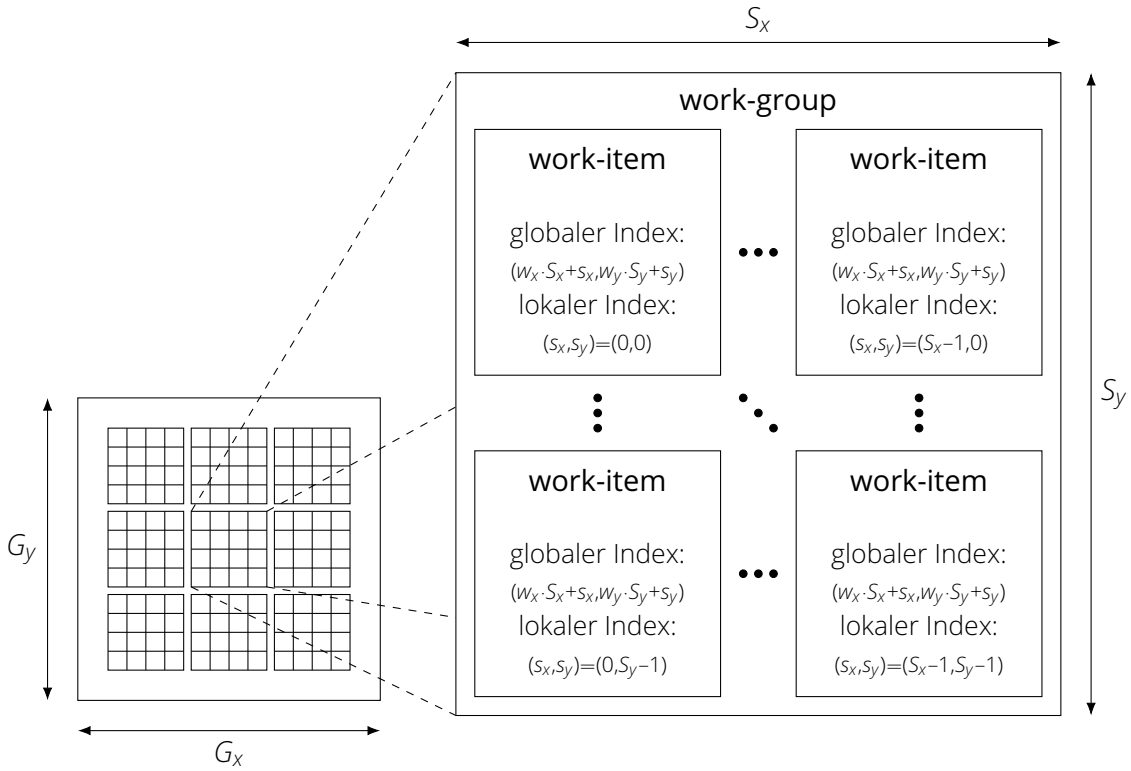

Abbildung 3.2.: SYCLs Indexraum [nach Mun12, S. 25]

doch über Funktionen der gemeinsamen *work-group* synchronisieren. Es ist jedoch nicht möglich, *work-items* verschiedener *work-groups* zu synchronisieren.

Durch diese Hardware-Abstraktion wird aus Sicht der Programmierers ein Indexraum aufgespannt – in OpenCL und SYCL *NDRange* genannt –, in dem jedem *work-item* ein eindeutiger Index innerhalb der *work-group* sowie der Menge aller *work-items* zugewiesen wird. Diese Indizes werden als lokale bzw. globale Indizes bezeichnet. Die Abbildung 3.2 zeigt diese Aufteilung am Beispiel einer zweidimensionalen *NDRange*. Dabei stehen G_x und G_y für die Gesamtzahl der *work-items* sowie S_x und S_y für die Zahl der *work-items* pro *work-group*, jeweils in x- und y-Richtung. w_x und w_y bezeichnen die Position der *work-group* innerhalb der *NDRange*, während s_x und s_y die Position eines *work-items* in der *work-group* – also den lokalen Index – darstellen. Der globale Index (g_x, g_y) eines *work-items* lässt sich demnach wie folgt berechnen [vgl. Mun12, S. 24]:

$$(g_x, g_y) = (w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$$

Die Zahl (W_x, W_y) der *work-groups* innerhalb der *NDRange* lässt sich ebenfalls bestimmen [vgl. Mun12, S. 25]:

$$(W_x, W_y) = \left(\frac{G_x}{S_x}, \frac{G_y}{S_y} \right)$$

Da eine *NDRange* bis zu drei Dimensionen umfassen kann, lassen sich mehrdimensionale Algorithmen auf diese Weise einfach implementieren.

Im Unterschied zu OpenCL kann bei SYCL die Angabe der Aufteilung in *work-groups* und *work-items* entfallen, lediglich die Gesamtzahl der *work-items* muss angegeben werden. In diesem Fall ist es Aufgabe der SYCL-Implementierung, die Zahl der nötigen *work-groups* zu bestimmen sowie die Zuordnung der *work-items* durchzuführen. Dadurch verliert der Programmierer jedoch die Möglichkeit, gruppenweite Operationen durchzuführen, wie z.B. die Synchronisierung innerhalb der *work-group*.

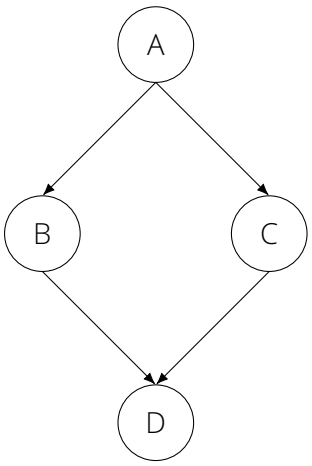

Abbildung 3.3.: Einfacher Aufgabengraph

3.2.2. Abhängigkeiten zwischen Kerneln

Alle *command groups*, die der Programmierer in eine SYCL-queue einreiht, werden grundsätzlich asynchron ausgeführt, sofern keine Abhängigkeiten zueinander bestehen. Vorhandene Abhängigkeiten werden über die von den Kerneln verwendeten Puffer durch die SYCL-Laufzeitumgebung automatisch erkannt und die Kernel in der richtigen Reihenfolge ausgeführt [vgl. KRH19, S. 21–23]. In diesem Aspekt unterscheidet sich SYCL von OpenCL, das neben vollständig seriellen Warteschlangen nur asynchrone Warteschlangen kennt, bei denen die richtige Sortierung der Kernel Aufgabe des Programmierers ist. Wichtig ist außerdem, dass die Abhängigkeiten über die Puffer-Verfügbarkeit ermittelt werden – und nicht über das Ende der Kernel [vgl. KRH19, S. 166].

Graphisch lässt sich dies in Form eines gerichteten Graphen veranschaulichen, wie für einen einfachen Fall in Abbildung 3.3gezeigt. Im Beispiel hängen die *command groups* B und C von der *command group* A ab, sind jedoch voneinander unabhängig. Dadurch müssen sie beide auf das Ende von A warten, können danach jedoch parallel ausgeführt werden. Die *command group* D benötigt wiederum die in B und C berechneten Ergebnisse und wartet daher auf deren Ende.

Quelltext 3.9 zeigt die dem Graphen entsprechende Verwendung einer SYCL-queue. Wie man leicht sieht, ist der Aufwand hinsichtlich der Abhängigkeitsverwaltung für den Programmierer sehr gering.

```
queue.submit(/* A */);
queue.submit(/* B */);
queue.submit(/* C */);
queue.submit(/* D */);
```

Quelltext 3.9.: Einfacher SYCL-Aufgabengraph

3.2.3. Fehlerbehandlung

SYCL übernimmt das System der Ausnahmefehler (engl. *exceptions*) aus der C++-Standardbibliothek. Das Fehlersystem ist in SYCL asynchron: grundsätzlich werden nur (synchrone) Fehler der Host-Seite ausgegeben, während auf der Device-Seite aufgetretene Fehler ignoriert werden. Das Abfangen der Device-Fehler erfordert einen weiteren Parameter für die queue. Dieser ist eine Datenstruktur, welche die asynchronen Fehler der Device-Seite abfängt und

in synchrone Host-Fehler umwandt, die dann vom Programmierer weiter verarbeitet werden können (siehe Quelltext 3.10).

```
#include <cstdlib>
#include <CL/sycl.hpp>

class XOCLDeviceSelector : public cl::sycl::device_selector {
    /* ... */
};

auto main() -> int
{
    try
    {
        auto exception_handler = [] (cl::sycl::exception_list exceptions)
        {
            for(std::exception_ptr e : exceptions)
            {
                try
                {
                    std::rethrow_exception(e);
                }
                catch(const cl::sycl::exception& err)
                {
                    /* Fehlerbehandlung Device */
                }
            }
        };

        // Beschleunigerwahl und Befehlswarteschlange
        auto queue = cl::sycl::queue{XOCLDeviceSelector{},
                                     exception_handler};

        /* ... */

        // Synchronisierung und Ausnahmefehler
        queue.wait_and_throw();
    }
    catch(const cl::sycl::exception& err)
    {
        /* Fehlerbehandlung Host */
    }

    return EXIT_SUCCESS;
}
```

Quelltext 3.10.: Verwendung von SYCL-Ausnahmefehlern

3.2.4. Profiling

SYCL ermöglicht über die `queue` ein rudimentäres Profiling. Dieses bietet dem Programmierer die Möglichkeit, über von der `queue` generierte `events` Informationen über Start- und Endzeitpunkt der Kernel sowie den gegenwärtigen Ausführungsstand eines Kernels zu erhalten. Dazu muss der `queue` ein besonderer Parameter während der Konstruktion übergeben werden (siehe Quelltext 3.11).

```
#include <cstdlib>
#include <CL/sycl.hpp>

class XOCLDeviceSelector : public cl::sycl::device_selector {
    /* ... */
};

auto main() -> int
{
    // Beschleunigerwahl und Befehlswarteschlange
    auto queue = cl::sycl::queue{XOCLDeviceSelector{},
                                 cl::sycl::property::queue::enable_profiling{}};

    // Kernel-Event generieren
    auto event = queue.submit(/* ... */);

    // Ausführungsstatus abfragen - Rückgabe: submitted, running oder complete
    auto status =
        event.get_info<cl::sycl::info::event::command_execution_status>();

    // Synchronisierung
    queue.wait();

    // Profilinginformationen abfragen - Rückgabe: Zeitpunkt in ns
    auto start =
        event.get_profiling_info<
                          cl::sycl::info::event_profiling::command_start>();
    auto stop =
        event.get_profiling_info<cl::sycl::info::event_profiling::command_end>();

    auto duration = stop - start;

    return EXIT_SUCCESS;
}
```

Quelltext 3.11.: Verwendung von SYCL-Profiling

3.2.5. Referenz-Semantik

Ein wichtiger Unterschied zur üblichen C++-Programmierung sind SYCLs Referenz-Semantiken. Die Spezifikation schreibt vor [siehe KRH19, Abschnitt 4.3.2]:

> Each of the following SYCL runtime classes: `device`, `context`, `queue`, `program`, `kernel`, `event`, `buffer`, `image`, `sampler`, `accessor` and `stream` must obey the following statements, where `T` is the runtime class: [...]
> Any instance of `T` that is constructed as a copy of another instance, via either the copy constructor or copy assignment operator, must behave as-if it were the original instance and as-if any action performed on it were also performed on the original instance and if said instance is not a host object must represent and continue to represent the same underlying OpenCL objects as the original instance where applicable.

Bemerkenswert ist, dass diese Semantik ebenfalls für die Typen `buffer` und `image` gilt, das heißt Datentypen, die größere Speicherbereiche kapseln. In der C++-Standardbibliothek werden die internen Felder vergleichbarer Typen (wie `vector`) ebenfalls kopiert. Nach dem Kopier-

vorgang existieren damit zwei voneinander verschiedene Objekte, die getrennte Speicherbereiche verwalten. Die SYCL-Objekte beziehen sich jedoch nach dem Kopiervorgang auf den selben Speicherbereich, es wird also bei der Objektkopie kein neuer Speicher angelegt. De facto handelt es sich bei der Kopie eines SYCL-Objekts daher lediglich um eine Referenz auf das ursprüngliche Objekt.

3.3. Implementierungen

Der ersten Veröffentlichung der SYCL-Spezifikation im Mai 2015 folgten im Laufe der Zeit einige Implementierungen. Diese werden in den folgenden Abschnitten vorgestellt.

Darüber hinaus existiert eine von der Firma Codeplay betreute Internet-Seite, die sich dem gesamten SYCL-Ökosystem widmet. [vgl. Codb]

3.3.1. ComputeCpp

Die schottische Firma Codeplay ist der zur Zeit einzige Anbieter einer kommerziellen SYCL-Implementierung, die unter dem Namen *ComputeCpp* vermarktet wird. Sie richtet sich in erster Linie an Hardware für die Bereiche Automotive und Embedded, unterstützt jedoch (bei einer bereits vorhandenen OpenCL-Implementierung) auch CPUs und GPUs der Firma Intel sowie (experimentell) NVIDIA-GPUs. Nach vorheriger Registrierung ist für nichtkommerzielle Zwecke auch eine kostenlose *community edition* verfügbar. [vgl. Coda]

3.3.2. Intel

Eine wichtige quelloffene Implementierung kommt von der Firma Intel. Strategisch soll diese Implementierung mit dem Compiler *clang* des LLVM-Projekts vereinigt werden. Zur Zeit handelt es sich jedoch noch um eine eigenständige Implementierung, die vor allem auf die Intel-OpenCL-Implementierungen für CPUs und GPUs abzielt. Aktivitäten innerhalb des öffentlich einsehbaren Quelltext-Repositoriums deuten jedoch darauf hin, dass auch die eigenen FPGAs unterstützt werden sollen. [vgl. Bad+]

3.3.3. triSYCL

Das Projekt triSYCL ist eine quelloffene Implementierung des SYCL-Standards, die früher von der Firma AMD und jetzt von Xilinx entwickelt wird. Nach eigener Aussage dient es vornehmlich experimentellen Zwecken, um dem SYCL-Komitee und dem OpenCL-C++-Komitee des Khronos-Konsortiums sowie dem C++-Standardisierungskomitee der ISO Feedback liefern zu können. Das Hauptprojekt unterstützt CPUs (über OpenMP oder TBB) sowie OpenCL-Implementierungen, die die Verarbeitung des SPIR-Zwischencodes unterstützen. [vgl. Ker+]

Daneben existiert ein von der Intel-Implementierung abgeleitetes Compiler-Projekt, das sich vornehmlich der besseren Unterstützung von Xilinx-FPGAs anzunehmen scheint. [vgl. KGL]

3.3.4. hipSYCL

Der Heidelberger Doktorand Aksel Alpay ist der Autor einer weiteren SYCL-Implementierung. Diese setzt auf dem CUDA-Klon der Firma AMD auf, der *general-purpose computing on graphics processing units* (GPGPU)-Sprache HIP. HIP ist sowohl auf AMD- als auch auf NVIDIA-GPUs ausführbar. Dadurch können auch mit hipSYCL entwickelte Programme auf diesen GPUs ausgeführt werden. hipSYCL war über weite Strecken ein Ein-Mann-Projekt, erst seit Februar 2019 ist die regelmäßige Mitarbeit eines weiteren Entwicklers zu verzeichnen. Aus diesem Grund ist

hipSYCL unvollständig implementiert, es fehlen unter anderem atomare Funktionen oder die Möglichkeit, Ausnahmefehler zu werfen und abzufangen. [vgl. Alp]

3.3.5. sycl-gtx

Eine weitere Open-Source-Implementierung ist das eingangs erwähnte *sycl-gtx*. Ursprünglich ist diese Implementierung im Rahmen einer Masterarbeit entstanden [vgl. Žuž16] und wird bis heute vom ursprünglichen Autoren weiterentwickelt. Aufgrund der begrenzten Entwicklerkapazitäten ist diese Variante aber immer noch sehr rudimentär und unterstützt nur eine Teilmenge der SYCL-Spezifikation.

Im Gegensatz zu den anderen Implementierungen wird der SYCL-Kernel erst zur Laufzeit des kompilierten Programms in einen OpenCL-Kernel umgewandelt und anschließend an die zugrundeliegende OpenCL-Laufzeitumgebung weitergereicht. Dadurch ist *sycl-gtx* sehr portabel, da es nicht auf eine bestimmte Hardware beschränkt ist; grundsätzlich soll es mit jeder OpenCL-Umgebung kompatibel sein, die mindestens den Standard in Version 1.2 unterstützt. [vgl. Žuž16, S. 47 ff.]

3.4. Erweiterungen für FPGAs

Für die FPGAs des Herstellers Xilinx steht bereits eine experimentelle SYCL-Implementierung zur Verfügung. Um die speziellen Eigenschaften dieses Hardware-Typs besser nutzen zu können, gibt es Erweiterungen, die SYCLs Funktionsumfang um FPGA-spezifische Funktionalität ergänzen. Diese sind in der Header-Datei `CL/sycl/xilinx/fpga.hpp` definiert und werden in den folgenden Abschnitten vorgestellt.

Datenflussorientierte Ausführung

Aus Xilinx' OpenCL-Implementierung übernimmt der triSYCL-Compiler eine datenflussbasierte Erweiterung. Diese Erweiterung ermöglicht die aufgabenparallele Ausführung aufeinanderfolgender Funktionen und Schleifen. Mit ihr wird der Compiler angewiesen, die Abhängigkeiten zwischen den einzelnen Schritten zu analysieren und für diese Schritte das *Producer/Consumer*-Prinzip durch eine Zwischenschaltung von Puffern durchzusetzen. [siehe Xil19c, S. 70 ff.]

In OpenCL ist diese Erweiterung als `xcl_dataflow` verfügbar und wird im OpenCL-C-Dialekt einem Kernel, einer Funktion oder einer Schleife als Attribut zugewiesen. Der SYCL-Implementierung steht diese Erweiterung unter dem Namen `dataflow` zur Verfügung. Mit ihr werden Funktionen markiert, auf deren innere Funktionen und Schleifen die entsprechenden Optimierungen angewandt werden (siehe Quelltext 3.12).

```
auto body(/* ... */)
{
    /* Funktionskörper */
}

struct kernel
{
    auto operator()()
    {
        cl::sycl::xilinx::dataflow(body(/* ... */));
    }
};
```

Quelltext 3.12.: Datenfluss-Erweiterung in SYCL

Pipeline-basierte Ausführung

Die triSYCL-Implementierung übernimmt aus Xilinx' OpenCL-Umgebung eine pipeline-basierte Erweiterung. Mit dieser kann der Compiler angewiesen werden, die Iterationen einer Schleife zu überlappen. Dadurch können die Iterationen bestimmte Ressourcen zeitgleich nutzen, wodurch sich der Ressourcenverbrauch insgesamt sowie die Latenz verringern können. [siehe Xil19c, S. 67 ff.]

In der von Xilinx ausgelieferte OpenCL-Implementierung handelt es sich bei dieser Erweiterung um das Attribut `xcl_pipeline_loop`, mit dem Schleifen markiert werden. In SYCL ist sie unter dem Namen `pipeline` verfügbar und wird auf Funktionen angewendet, deren innere Schleifen dann dieser Optimierung unterzogen werden (siehe Quelltext 3.13).

```
auto body(/* ... */)
{
    for(int i = 0; i < 32; ++i)
    {
        /* Schleifenkörper */
    }
}

struct kernel
{
    auto operator()()
    {
        cl::sycl::xilinx::pipeline(body(/* ... */));
    }
};
```

Quelltext 3.13.: Pipeline-Erweiterung in SYCL

Feldpartitionierung

Durch die Verteilung eines Datenfeldes auf mehrere physische Speichersegmente lässt sich für manche Anwendungen eine höhere Speicherbandbreite erzielen. Mit Xilinx' High-Level-Synthese lässt sich ein logisches Datenfeld auf drei verschiedene Weisen zerlegen: *cyclic*, *block* und *complete*. [vgl. Xil19d, S. 16]

Der Typ *cyclic* führt eine zyklische Zerlegung des Feldes durch. Geht man von einem achtelementigen Feld aus und hat vier physische Speicher zur Verfügung, so werden die Elemente einzeln in aufsteigender Reihenfolge auf die Speicher aufgeteilt: Element 0 wird dem Speicher 0 zugeordnet, Element 1 dem Speicher 1, und so weiter. Ist Speicher 3 erreicht, beginnt die Zuteilung wieder von vorne, Element 4 wird dem Speicher 0 zugeordnet, Element dem Speicher 1, und so weiter. [vgl. Xil19d, S. 17]

Der Typ *block* zerlegt das Feld blockweise. Das heißt, dass zuerst der Speicher 0 mit den ersten Elementen des Feldes befüllt wird, dann der Speicher 1, und so weiter. [vgl. Xil19d, S. 17]

Beim Typ *complete* wird das Feld in einzelne Elemente zerlegt. Dies entspricht einer Verteilung des Feldes auf einzelne Register. [vgl. Xil19d, S. 17]

Das Attribut `xcl_array_partition(<Typ>, <Faktor>, <Dimension>)` steht als Erweiterung in Xilinx' OpenCL-Implementierung zur Verfügung, um die Partitionierung durchzuführen. Dabei bezeichnet `<Typ>` einen der drei oben genannten Typen. [vgl. Xil19d, S. 17]

`<Faktor>` gibt für *cyclic* die Anzahl der Speicher an, auf die das Feld verteilt werden soll, und für *block* die Anzahl der Elemente pro Speicher. Für den Typ *complete* ist dieser Parameter nicht definiert. [vgl. Xil19d, S. 17]

`<Dimension>` gibt an, welche Dimension des Feldes auf die beschriebene Weise partitioniert werden soll. [vgl. Xil19d, S. 17]

In SYCL steht diese Erweiterung unter dem Namen `partition_array` zur Verfügung, wobei die Zuweisung der oben aufgeführten Parameter hier über Templates erfolgt. Der Quelltext 3.14 zeigt die Anwendung dieser Erweiterung.

```
struct kernel
{
    auto operator()()
    {
        // zyklische Verteilung von a auf 4 physische Speicher
        auto a = cl::sycl::xilinx::partition_array<int, 16,
                    cl::sycl::xilinx::partition::cyclic<4, 1>>{};

        // blockweise Verteilung von b mit 4 Elementen pro physischem Speicher
        auto b = cl::sycl::xilinx::partition_array<int, 16,
                    cl::sycl::xilinx::partition::block<4, 1>>{};

        // Zerlegung von c in 16 Register
        auto c = cl::sycl::xilinx::partition_array<int, 16,
                    cl::sycl::xilinx::partition::complete<1>>{};
    }
};
```

Quelltext 3.14.: Feldpartitionierung in SYCL

4. Die Alpaka-Bibliothek

Dieses Kapitel führt in die Alpaka-Bibliothek ein. Wie im vorherigen SYCL-Kapitel wird der grundlegende Aufbau eines Alpaka-Programms anhand des AXPY-Beispiels dargestellt. Ein weiterer Abschnitt ist den darauf aufbauenden, erweiterten Konzepten, wie etwa der Hardware-Abstraktion, gewidmet.

4.1. Überblick

Alpaka (Eigenschreibweise: *alpaka*) steht für *Abstraction Library for Parallel Kernel Acceleration* und wurde ursprünglich von Benjamin Worpitz im Rahmen seiner Masterarbeit entwickelt [vgl. Wor15]. Mittlerweile wird die Entwicklung durch die Gruppe *Computergestützte Strahlenphysik* des *Instituts für Strahlenphysik* am *Helmholtz-Zentrum Dresden-Rossendorf* fortgeführt.

Die Alpaka-Bibliothek definiert eine abstrakte C++-Schnittstelle, mit deren Hilfe parallele Programme geschrieben werden können. Im Hintergrund wird Alpaka auf hersteller- oder hardware-spezifische Schnittstellen, wie CUDA oder OpenMP, – im Folgenden als *Backend* bezeichnet – abgebildet. Alpaka ist somit ein einheitliches Paket, das die abstrakte Schnittstelle nach außen und die konkrete Implementierung vereinigt. Damit unterscheidet sich die Bibliothek von ähnlichen Ansätzen wie OpenCL oder SYCL, die ebenfalls eine abstrakte Schnittstelle definieren, die Implementierung jedoch den Hardware- und Software-Herstellern überlassen.

Wie bei SYCL sind die Quelltexte für *Host* und *Device* nicht voneinander getrennt. Die Abbildung auf ein oder mehrere Backends erfolgt zur Compile-Zeit durch Template-Metaprogrammierung, wodurch ein Abstraktions-Overhead zur Laufzeit vermieden wird.

Ähnlich wie SYCL bietet auch Alpaka ein beschleunigerunabhängiges Backend, das sich prinzipiell mit jedem modernen C++-Compiler kompilieren lässt. Dieses wird in der Alpaka-Terminologie `CpuSerial` genannt und führt jeden Befehl seriell aus. Während der Kernel-Ausführung existiert also keinerlei Parallelität. Daneben gibt es weitere Implementierungen für CPUs auf Basis des OpenMP-Standards, der ebenfalls von den meisten modernen C++-Compilern unterstützt wird.

4.1.1. AXPY und Alpaka

Zum Zwecke der einfachen Vergleichbarkeit der von SYCL und Alpaka gebotenen Programmiermodelle wird das im vorigen Kapitel verwendete AXPY-Beispiel der BLAS-Bibliothek hier erneut aufgegriffen. Strukturell ähneln sich SYCL und Alpaka recht stark, wie der Platzhalter-Quelltext 4.1 zeigt. Wie im vorigen Kapitel wird auch dieser Quelltext nach und nach mit Inhalt gefüllt.

```
#include <cstdlib>

#include <alpaka/alpaka.hpp>

auto main() -> int
{
    // Beschleunigerwahl und Befehlswarteschlange

    // Speicherreservierung und -initialisierung

    // Kerneldefinition und -ausführung

    // Synchronisierung

    return EXIT_SUCCESS;
}
```

Quelltext 4.1.: Struktur eines Alpaka-Programms

Beschleunigerwahl und Befehlswarteschlange

Die Auswahl des Beschleunigers erfolgt bei Alpaka bereits zur Compile-Zeit. Der Programmierer muss also im Vorfeld eine Entscheidung darüber treffen, auf welchen Systemen sein Programm lauffähig sein soll. Dazu wählt der Programmierer zunächst einen der im Umfang von Alpaka vorhandenen beschleunigerspezifischen Datentypen aus und definiert, welche Datentypen für die Angabe der im Programm verwendeten Dimensionen und Indizes verwendet werden sollen. Dieser Vorgang ist in Quelltext 4.2 dargestellt.

```
// Definition für ein eindimensionales Problem
using Dim = alpaka::dim::DimInt<1u>;
using Idx = std::size_t;
using Acc = alpaka::acc::AccGpuCudaRt<Dim, Idx>;
```

Quelltext 4.2.: Auswahl der in Alpaka vorhandenen NVIDIA-CUDA-Implementierung

In der Folge wird der definierte Typ `Acc` als Parameter für weitere Datentypen verwendet. Bei letzteren handelt es sich um die Klassen für die Verwaltung konkreter Beschleuniger. Diese sind in Alpaka als abstrakte Template-Klassen mit einer aus Sicht des Programmierers einheitlichen Schnittstelle vorhanden. Durch den `Acc`-Parameter werden diese Klassen zur Compile-Zeit spezialisiert und enthalten die für die jeweilige Hardware korrekten Code-Pfade. Dadurch ist der Gesamtquelltext mit wenig Aufwand portabel, da ein einfacher Austausch des `Acc`-Parameters passenden Code für andere Hardware-Architekturen generieren kann. Neben der Auswahl des Beschleunigers ist außerdem die Auswahl eines Datentypen für den *Host* erforderlich, der später die Verwaltung einiger *Host*-seitiger Befehle übernehmen wird. Der Quelltext 4.3 zeigt die Verwendung der genannten Strukturen für die oben gewünschte CUDA-Implementierung.

```
using DevAcc = alpaka::dev::Dev<Acc>;
using PltfAcc = alpaka::pltf::Pltf<Acc>;
using PltfHost = alpaka::pltf::PltfCpu;
```

Quelltext 4.3.: Spezialisierung abstrakter Alpaka-Klassen

In einem weiteren Schritt muss die Befehlswarteschlange initialisiert werden. Diese dient – wie auch in SYCL – als Verbindungselement zwischen *Host* und *Device*. Alle das *Device* betreffenden Befehle werden in ihr eingereiht. Anders als in SYCL gibt es in Alpaka die Möglichkeit, eine zum *Host* synchrone Warteschlange zu verwenden – der *Host* wird von der Warteschlange also so lange blockiert, bis die Verarbeitung auf dem *Device* abgeschlossen ist. In diesem Beispiel ist dies jedoch nicht notwendig, weshalb eine asynchrone Warteschlange verwendet wird, wie Quelltext 4.4 zeigt. Wie man schnell erkennt, existiert kein `Acc`-Parameter für eine Alpaka-queue – der Programmierer muss diese also ebenfalls selbst passend wählen.

```
using Queue = alpaka::queue::QueueCudaRtNonBlocking;
```

Quelltext 4.4.: Auswahl der Alpaka-Befehlswarteschlange

Im letzten Schritt werden die oben definierten Datentypen instanziiert oder als Parameter für die Instanziierung weiterer Datenstrukturen verwendet. Mit dem in Quelltext 4.5 gezeigten Vorgehen ist der erste Abschnitt eines Alpaka-Programms beendet.

```
// instanziiere Host und Device
auto devHost = alpaka::pltf::getDevByIdx<PltfHost>(0u);
auto devAcc = alpaka::pltf::getDevByIdx<PltfAcc>(0u);

// instanziiere Befehlswarteschlange
auto queue = Queue{devAcc};
```

Quelltext 4.5.: Instanziierung der Alpaka-Datentypen

Speicherreservierung und -initialisierung

Die für die AXPY-Operation nötigen Vektoren müssen zunächst im Speicher angelegt und initialisiert werden. Dies erfolgt in drei Stufen:

1. Reserviere gleich große Speicherbereiche sowohl auf dem *Host* als auch auf dem *Device*.
2. Initialisiere den *Host*-Speicher mit den gewünschten Werten.
3. Kopiere die initialisierten Werte auf das *Device*.

Ähnlich wie bei SYCL werden Speicherbereiche in Alpaka durch Puffer dargestellt, die die reinen Zeiger kapseln. Im Gegensatz zu SYCL sind die Puffer auf dem Host und Device jedoch voneinander unabhängig. Es ist also nicht möglich, einen gemeinsamen Puffer zu erzeugen, der die notwendigen Kopien für den Programmierer unsichtbar im Hintergrund durchführt, wie dies in Abschnitt 3.1.1 für SYCL demonstriert wurde.

Für die Erzeugung von Puffern ist es zunächst nötig, die gewünschte Größe des Puffers durch eine spezielle Datenstruktur zu definieren, wie der Quelltext 4.6 zeigt.

```
auto extent = alpaka::vec::Vec<Dim, Idx>{numElements};
```

Quelltext 4.6.: Definition eines Größenvektors mit Alpaka

In der Folge kann die Größe verwendet werden, um den Speicher sowohl auf dem *Host* als auch auf dem *Device* zu reseriveren. Dies ist in Quelltext 4.7 dargestellt.

```
auto hostBufX = alpaka::mem::buf::alloc<int, Idx>(devHost, extent);
auto hostBufY = alpaka::mem::buf::alloc<int, Idx>(devHost, extent);

auto devBufX = alpaka::mem::buf::alloc<int, Idx>(devAcc, extent);
auto devBufY = alpaka::mem::buf::alloc<int, Idx>(devAcc, extent);
```

Quelltext 4.7.: Speicherallokation mit Alpaka

Die in Quelltext 4.8 durchgeführte Initialisierung erfolgt über die von den *Host*-Puffern gekapselten Zeiger, auf die man bei Alpaka direkt zugreifen kann.

```
auto hostPtrX = alpaka::mem::view::getPtrNative(hostBufX);
auto hostPtrY = alpaka::mem::view::getPtrNative(hostBufY);

for(auto i = 0; i < numElements; ++i)
{
    hostPtrX[i] = /* ... */;
    hostPtrY[i] = /* ... */;
}
```

Quelltext 4.8.: Initialisierung eines Alpaka-Puffers

Im letzten Schritt (siehe Quelltext 4.9) müssen die Daten vom *Host* auf das *Device* kopiert werden. Dazu werden Kopieroperationen in der oben angelegten Warteschlange eingereiht.

```
alpaka::mem::view::copy(queue, devBufX, hostBufX, extent);
alpaka::mem::view::copy(queue, devBufY, hostBufY, extent);
```

Quelltext 4.9.: Kopie der initialisierten Daten mit Alpaka

Kerneldefinition und -ausführung

Um die parallelen Eigenschaften der Ziel-Hardware nutzen zu können, erfordert Alpaka eine Information, wie das zu bearbeitende Problem auf die Hardware-Ressourcen aufgeteilt werden soll. Dazu kann der Programmierer eine in Alpaka vorhandene Funktion nutzen, die eine gute Aufteilung schätzen kann, oder selbst Definitionen für die Größe des *Grids*, die Zahl der *Threads* sowie der *Elements* pro *Thread* angeben. Die Aufteilung in *Grid*, *Threads* und *Elements* ist ein Konzept der in Alpaka verwendeten Hardware-Abstraktion und wird detailliert in Abschnitt 4.2.1 behandelt. Für dieses Beispiel genügt die Verwendung der Schätzfunktion, wie sie in Quelltext 4.10 gezeigt wird.

```
auto workDiv = alpaka::workdiv::getValidWorkDiv<Acc>(
                devAcc,                 // für welches Device?
                extent,                 // für welche Problemgröße?
                static_cast<Idx>(1u));  // wie viele Elemente pro Thread?
```

Quelltext 4.10.: Arbeitsaufteilung durch Alpaka-Schätzfunktion

Alpaka-Kernel werden in Form von C++-Funktoren – das heißt Strukturen mit einem überladenen `()`-Operator – definiert. Der `()`-Operator wird mit einem speziellen Funktionsattribut versehen (`ALPAKA_FN_ACC`), der im Hintergrund dafür sorgt, dass diese Funktion für das *Device* und nicht den *Host* kompiliert wird.

Als Parameter nimmt der Kernel zunächst die oben definierte `devAcc`-Instanz entgegen. Dies ist nötig, um von dieser Struktur gekapselte *device*-seitige Funktionen im Kernel nutzen zu können. Daneben ist nur die Angabe der eigentlichen Funktionsparameter erforderlich, in diesem

Fall also die für den AXPY-Algorithmus nötige Konstante a sowie die Ein- bzw. Ausgangsvektoren. Wie auf dem *Host* erfolgt auch auf dem *Device* der Zugriff auf die Speicherbereiche durch reine Zeiger.

Innerhalb des Kernels wird ein Index-Raum aufgespannt (eine genaue Erläuterung dieses Prinzips folgt in Abschnitt 4.2.1), in dem jeder ausführenden Einheit des Beschleunigers ein eindeutiger Index sowie zu bearbeitende Elemente zugeordnet werden. Über in Alpaka vorhandene Funktionen ist eine Orientierung innerhalb des Index-Raums möglich, wodurch die AXPY-Funktion auf alle Vektor-Elemente angewendet werden kann.

Die gesamte Kernel-Definition ist in Quelltext 4.11 dargestellt.

```
struct AxpyKernel
{
    template <typename TAcc, typename TIdx>
    ALPAKA_FN_ACC auto operator()(
        const TAcc& acc,
        const TIdx& numElements,
        const int a,
        const int* X,
        int* Y
        ) const -> void
    {
        auto gridThreadIdx = alpaka::idx::getIdx<
                                    alpaka::Grid, alpaka::Threads>(acc)[0u];
        auto threadElemExtent = alpaka::workdiv::getWorkDiv<
                                    alpaka::Thread, alpaka::Elems>(acc)[0u];
        auto threadFirstElemIdx = gridThreadIdx * threadElemExtent;

        if(threadFirstElemIdx < numElements)
        {
            auto threadLastElemIdx = threadFirstElemIdx + threadElemExtent;

            for(auto i = threadFirstElemIdx; i < threadLastElemIdx; ++i)
            {
                Y[i] = a * X[i] + Y[i];
            }
        }
    }
};
```

Quelltext 4.11.: Kernel-Definition in Alpaka

Der so definierte Kernel wird im nächsten Schritt einem *Task* zugeordnet. Ein *Task* ist prinzipiell mit einer *command group* in SYCL vergleichbar und umfasst neben der Kernel-Definition die gewünschte Aufteilung der Arbeit sowie die konkreten Ein- und Ausgabeparameter für eine Kernel-Ausführung. Ein Alpaka-*Task* wird einer Alpaka-*Queue* übergeben und von dieser auf dem *Device* zur Ausführung gebracht.

Der gesamte Prozess ist in Quelltext 4.12 gezeigt.

```
auto taskKernel = alpaka::kernel::createTaskKernel<Acc>(
                    workDiv,
                    AxpyKernel{},
                    numElements,
                    a,
                    alpaka::mem::view::getPtrNative(devBufX),
                    alpaka::mem::view::getPtrNative(devBufY));

alpaka::queue::enqueue(queue, taskKernel);
```

Quelltext 4.12.: Task-Definition und -Ausführung in Alpaka

Synchronisierung

Nach dem Abschluss der Berechnung ist die Prüfung der Ergebnisse erforderlich. Dazu werden diese durch eine Kopieroperation zunächst wieder auf den *Host* übertragen. Dies geschieht wieder in einer *Queue*. Um auf den Abschluss aller ausstehenden Operationen zu warten, kann der *Host* durch eine Wartefunktion blockiert werden. Dieser Vorgang wird in Quelltext 4.13 demonstriert.

```
alpaka::mem::view::copy(queue, hostBufX, devBufX, extent);
alpaka::mem::view::copy(queue, hostBufY, devBufY, extent);

alpaka::wait::wait(queue); // warte auf ausstehende Operationen

/* ab hier Überprüfung der Ergebnisse möglich */
```

Quelltext 4.13.: Synchronisation zwischen Host und Device in Alpaka

Zusammenfassung

Der Quelltext 4.14 zeigt das gesamte Alpaka-AXPY-Beispiel, jedoch aus Platzgründen ohne die Kernel-Definition.

```cpp
#include <cstdlib>
#include <alpaka/alpaka.hpp>

struct AxpyKernel
{
    template <typename TAcc, typename TIdx>
    ALPAKA_FN_ACC auto operator()(const TAcc& acc, const TIdx& numElements,
                            const int a, const int* X, int* Y) const -> void
    {
        /* ... */
    }
};

auto main() -> int
{
    using Dim = alpaka::dim::DimInt<1u>;
    using Idx = std::size_t;
    using Acc = alpaka::acc::AccGpuCudaRt<Dim, Idx>;
    using DevAcc = alpaka::dev::Dev<Acc>;
    using PltfAcc = alpaka::pltf::Pltf<Acc>;
    using PltfHost = alpaka::pltf::PltfCpu;
    using Queue = alpaka::queue::QueueCudaRtNonBlocking;

    auto devHost = alpaka::pltf::getDevByIdx<PltfHost>(0u);
    auto devAcc = alpaka::pltf::getDevByIdx<PltfAcc>(0u);
    auto queue = Queue{devAcc};
    auto extent = alpaka::vec::Vec<Dim, Idx>{numElements};
    auto hostBufX = alpaka::mem::buf::alloc<int, Idx>(devHost, extent);
    auto hostBufY = alpaka::mem::buf::alloc<int, Idx>(devHost, extent);
    auto devBufX = alpaka::mem::buf::alloc<int, Idx>(devAcc, extent);
    auto devBufY = alpaka::mem::buf::alloc<int, Idx>(devAcc, extent);
    auto hostPtrX = alpaka::mem::view::getPtrNative(hostBufX);
    auto hostPtrY = alpaka::mem::view::getPtrNative(hostBufY);

    for(auto i = 0; i < numElements; ++i) {
        hostPtrX[i] = /* ... */;
        hostPtrY[i] = /* ... */;
    }
    alpaka::mem::view::copy(queue, devBufX, hostBufX, extent);
    alpaka::mem::view::copy(queue, devBufY, hostBufY, extent);

    auto workDiv = alpaka::workdiv::getValidWorkDiv<Acc>(
                    devAcc, extent, static_cast<Idx>(1u));
    auto taskKernel = alpaka::kernel::createTaskKernel<Acc>(
                        workDiv, AxpyKernel{}, numElements, a,
                        alpaka::mem::view::getPtrNative(devBufX),
                        alpaka::mem::view::getPtrNative(devBufY));
    alpaka::queue::enqueue(queue, taskKernel);
    alpaka::mem::view::copy(queue, hostBufX, devBufX, extent);
    alpaka::mem::view::copy(queue, hostBufY, devBufY, extent);
    alpaka::wait::wait(queue);

    return EXIT_SUCCESS;
}
```

Quelltext 4.14.: Vollständiges Alpaka-AXPY-Beispiel

4.2. Weiterführende Konzepte

Zum besseren Verständnis des obigen Beispiels sowie der in den nächsten Kapiteln geschilderten Schwierigkeiten beim Entwicklungsprozess des Backends ist die Kenntnis weiterführender Alpaka-Konzepte notwendig. Dazu gehören insbesondere die in Alpaka verwendete Hardware-Abstraktion und die Verwaltung der Abhängigkeiten zwischen Kerneln. Für den Entwicklungsprozess mit Alpaka sind außerdem die Methoden für die Fehlerbehandlung und das Profiling relevant.

4.2.1. Hardware-Abstraktion

Alpakas Hardware-Abstraktion trägt den Namen *Redundant Hierarchical Parallelism*[vgl. Wor15, S. 22]. Sie basiert auf dem Prinzip der Datenparallelität, das heißt der Idee, dass dieselbe Operation parallel auf verschiedene, gleichartige Daten angewendet werden kann. Ein triviales Beispiel für einen datenparallelen Algorithmus ist die in diesem Kapitel als Einführung verwendete AXPY-Methode: sie führt dieselbe Operation für alle Elemente der Vektoren $\vec{x}$ und $\vec{y}$ durch. Die im Folgenden erläuterten Konzepte wurden stark durch die in CUDA und OpenCL vorhandenen Abstraktionen beeinflusst [vgl. Wor15, S. 17].

Grid und Threads

Alpaka-*Threads* sind einer der Grundbausteine des Abstraktionskonzepts. Ein idealer datenparalleler Algorithmus lässt sich theoretisch optimal durch einen *Thread* pro Datenelement parallelisieren. Der Begriff „Thread“ ist in diesem Zusammenhang nicht mit einem Thread des Betriebssystems oder einem CUDA-Thread zu verwechseln – er repräsentiert lediglich eine Befehlssequenz, die den gewünschten Algorithmus auf ein einzelnes Datenelement anwendet [vgl. Wor15, S. 17]. Konzeptionell entspricht ein *Thread* also einem *work-item* aus SYCL und OpenCL. Die Menge aller *Threads* bildet ein *n*-dimensionales Gitter (engl. *Grid*). Dieses ist damit mit der aus SYCL und OpenCL bekannten *NDRange* vergleichbar.

Da manche Algorithmen die Kommunikation zwischen *Threads* erfordern, verfügen diese außerdem über entsprechende Mittel zur Synchronisation. Die genannte Befehlssequenz pro Element ist der *Kernel*. Die Abbildung 4.1 zeigt eine theoretische Hardware, auf die das *Thread-Grid*-Konzept ideal anwendbar wäre.

Blocks

Eine wie in Abbildung 4.1 dargestellte Hardware mit tausenden untereinander verbundenen Kernen kommt in der Realität selten vor. Es ist daher sinnvoll, eine Einschränkung der Kommunikations- und Synchronisationsfähigkeiten der *Threads* auf ein von handelsüblicher Hardware beherrschbares Niveau vorzunehmen.

Zu diesem Zweck verwendet Alpaka eine weitere Ebene innerhalb der Abstraktionshierarchie, die *Blocks*. Diese sind zwischen der *Thread*- und *Grid*-Ebene angesiedelt, wobei ein *Block* eine Teilmenge der *Threads* vereinigt. Das *Grid* wird dabei gleichmäßig auf die *Blocks* aufgeteilt, sodass alle *Block* gleich groß sind. Konzeptionell entspricht ein *Block* damit einer SYCL-*work-group*. Die schnelle Kommunikation und Synchronisation der *Threads* sind ausschließlich über einen *block*-internen kleinen gemeinsamen Speicher (*shared memory*) möglich, jedoch nicht mehr über das gesamte *Grid*. Eine für dieses Abstraktionsschema ideale theoretische Hardware ist in Abbildung 4.2 dargestellt. Diese erlaubt die *block*-interne Kommunikation, muss jedoch keine Fähigkeiten für die Kommunikation bzw. Synchronisation zwischen *Blocks* besitzen.

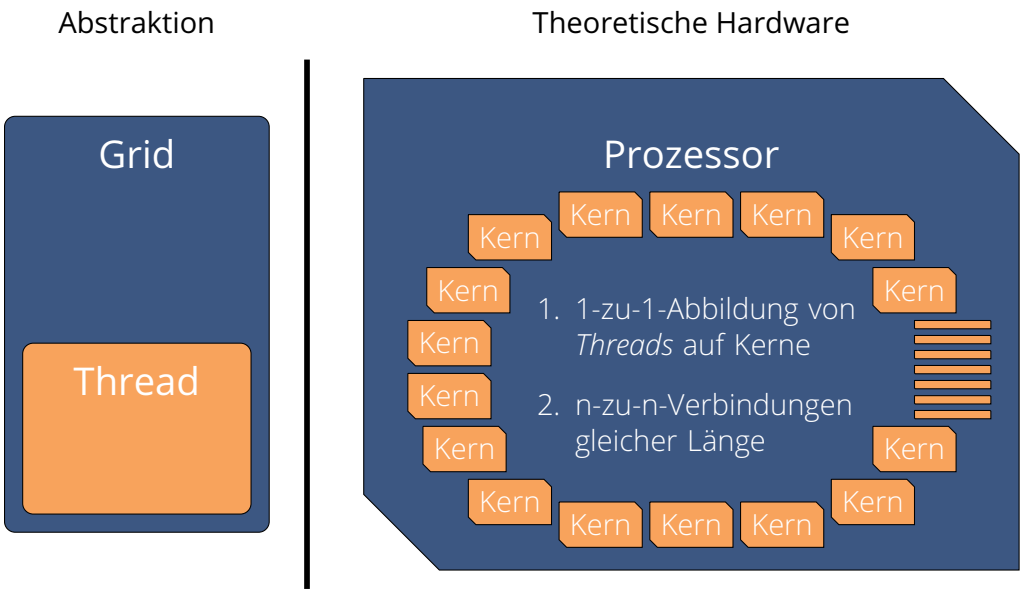

Abbildung 4.1.: Links: Abstraktionshierarchie mit einem aus *Threads* zusammengesetzten *Grid*. Rechts: Ein hypothetischer Prozessor, der datenparallele Anwendungen mit diesem Abstraktionsschema ideal ausführen könnte. [nach Wor15, S. 18]

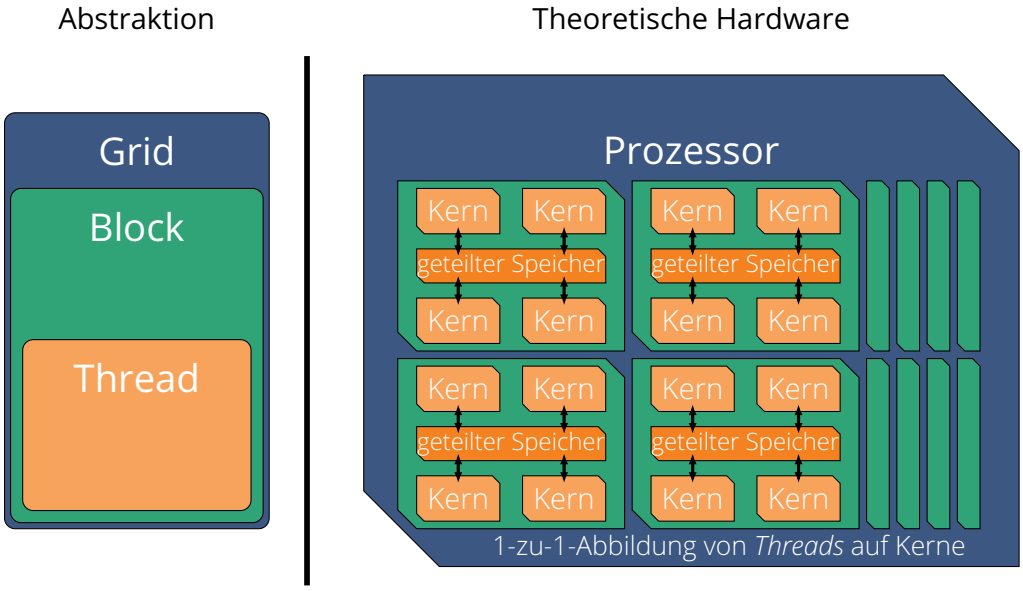

Abbildung 4.2.: Links: Abstraktionshierarchie mit einem *Grid*, das aus zu *Blocks* gruppierten *Threads* besteht. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von *Threads* auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der *Blocks* ermöglicht. [nach Wor15, S. 19]

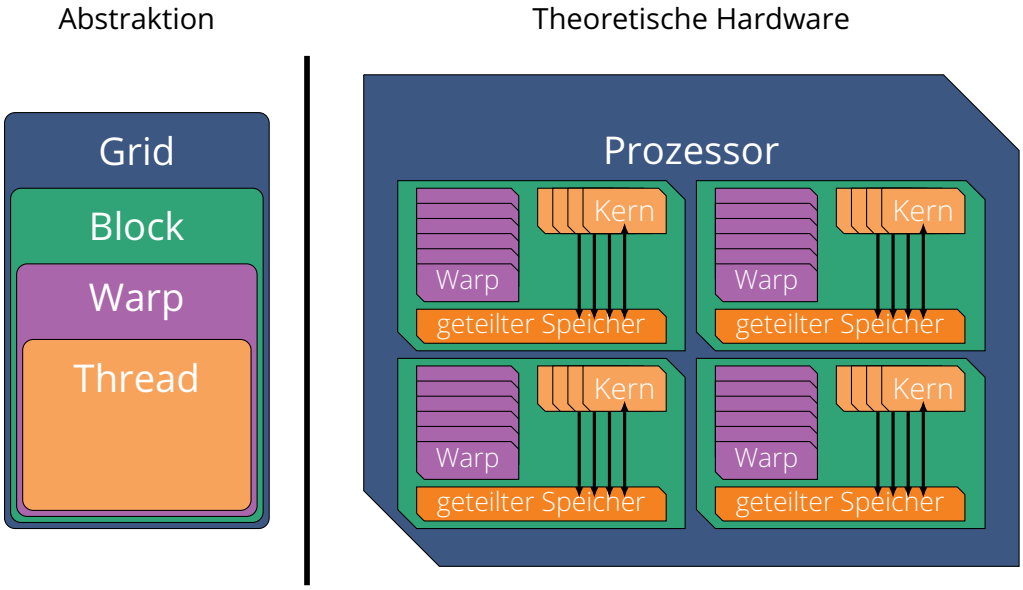

Abbildung 4.3.: Links: Abstraktionshierarchie mit einem *Grid*, das aus zu *Blocks* gruppierten *Warps* besteht. Letzere sind wiederum aus mehreren *Threads* zusammengesetzt. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von *Threads* auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der *Blocks* ermöglicht. Durch die taktgenaue gemeinsame Ausführung der *Threads* in Form von *Warps* kann Chipfläche eingespart werden. [nach Wor15, S. 20]

Warps

Auf einigen Hardware-Plattformen können *Thread*-Gruppen innerhalb eines *Blocks* taktgenau gemeinsam ausgeführt werden. Diese Gruppen werden in der Alpaka-Hierarchie mit dem CUDA-Begriff *Warp* bezeichnet. Das *Warp*-Konzept findet sich beispielsweise auf GPUs, deren Multiprozessoren *Threads* in *Warp*-Form ausführen. Dadurch können sich *Threads* Hardware-Ressourcen teilen, was die benötigte Chipfläche verkleinert. Die Abbildung 4.3 zeigt einen theoretischen Prozessor, der in der Lage ist, das Abstraktionskonzept aus *Grid*, *Blocks*, *Warps* und *Threads* auszuführen.

Die konkrete *Warp*-Größe unterscheidet sich zwischen verschiedenen Plattformen und Herstellern. Aufgrund ihrer Hardware-Nähe steht diese Information in OpenCL und SYCL zur Laufzeit zur Verfügung, ist dort jedoch kein eigener Teil des Abstraktionskonzepts. Alpaka stellt diese Information dagegen nicht zur Verfügung, obwohl *Warps* im theoretischen Konzept vorhanden sind.

Elements

Mit dem *Element*-Konzept ermöglicht Alpaka dem Programmierer die genauere Anpassung auf die Ziel-Hardware. Die Idee, jeden *Thread* genau ein Datum bearbeiten zu lassen, ist nicht immer der optimale Weg. So kann es beispielsweise sinnvoll sein, auf CPU-Beschleunigern mehrere Daten pro *Thread* innerhalb kurzer Schleifen zu verarbeiten. Dadurch können gute Compiler selbstständig von Vektorregistern der CPU Gebrauch machen, alternativ kann der Programmierer dies auch explizit selbst tun.

Die *Element*-Ebene ist die unterste Ebene der Alpaka-Abstraktionshierarchie. Dadurch ergibt sich die Gesamthierarchie, wie sie in Abbildung 4.4 dargestellt ist.

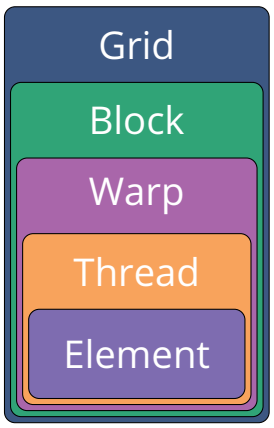

Abbildung 4.4.: Vollständiges Alpaka-Abstraktionskonzept: Mehrere *Elements* werden von einem *Thread* verarbeitet. Mehrere *Threads* werden innerhalb eines *Warps* taktgenau gemeinsam ausgeführt, mehrere *Warps* bilden wiederum einen *Block*. Die Menge aller *Blocks* ergibt das *Grid*. [nach Wor15, S. 22]

4.2.2. Abhängigkeiten zwischen Kerneln

Im Gegensatz zu SYCL verfügt Alpaka nicht über ein System, um automatisch Abhängigkeiten zwischen Kerneln zu verwalten. Stattdessen ist es Aufgabe des Programmierers, Alpaka-*Events*zur Synchronisierung zwischen Kerneln zu verwenden. Dies ist nötig, da Operationen (Kopien, Kernel, …) in derselben *Queue* zwar sequentiell in der Reihenfolge ihrer Einreihung abgearbeitet werden, in verschiedenen *Queues* dagegen zueinander asynchron laufen.

```
auto queue1 = QueueCudaRtNonBlocking{/*...*/};
auto queue2 = QueueCudaRtNonBlocking{/*...*/};

// A ist Vorbedingung für B und C
auto eventA = EventCudaRt{/* ... */};
// führe A aus und markiere anschließendes Event
alpaka::queue::enqueue(queue1, /* A */);
alpaka::queue::enqueue(queue1, eventA);

// B befindet sich in derselben Queue hinter A
alpaka::queue::enqueue(queue1, /* B */);

// C befindet sich in einer anderen Queue und muss auf eventA warten
alpaka::wait::wait(queue2, eventA);
// C ist Vorbedingung für D
auto eventC = EventCudaRt{/* ... */};
// führe C aus und markiere anschließendes Event
alpaka::queue::enqueue(queue2, /* C */);
alpaka::queue::enqueue(queue2, eventC);

// D befindet sich in derselben Queue hinter B, muss aber auf eventC warten
alpaka::wait::wait(queue1, eventC);
alpaka::queue::enqueue(queue1, /* D */);
```

Quelltext 4.15.: Einfacher Alpaka-Aufgabengraph

Events können zwischen *Queue*-Operationen eingefügt werden. Wurde eine Operation vollständig durchgeführt, so wird das nachfolgende *Event* entsprechend markiert. Andere *Queues* sowie der *Host* können dadurch auf dieses *Event* warten, bevor sie ihre eigenen Aufgaben fort-

setzen.

Der im vorigen Abschnitt (Abschnitt 3.2.2) Beispiel-Aufgabengraph erfordert demnach mehr Aufwand als das SYCL-Äquivalent, wie der Quelltext 4.15 zeigt.

4.2.3. Fehlerbehandlung

Ähnlich wie SYCL verwendet Alpaka C++-Exceptions für die Fehlerbehandlung zur Laufzeit. Allerdings wurde darauf verzichtet, spezielle Exceptions für Alpaka zu entwerfen. Stattdessen wird die Klasse `runtime_error` der C++-Standardbibliothek verwendet.

4.2.4. Profiling

Im Gegensatz zu SYCL bringt Alpaka keine eigenen Werkzeuge für das Profiling mit. Da Alpaka zur Compile-Zeit auf die herstellerspezifischen Schnittstellen abgebildet wird, ist dies auch nicht nötig – dem Programmierer stehen so die vom jeweiligen Hardware-Hersteller mitgelieferten Profiling-Werkzeuge zur Verfügung. So lässt sich beispielsweise der CUDA-Profiler `nvprof` für das Profiling eines mit Alpaka auf NVIDIA-Hardware portierten Programms verwenden, was mit den für NVIDIA-GPUs existierenden SYCL-Implementierungen nicht möglich ist.

5. Implementierung des SYCL-Backends der Alpaka-Bibliothek

Mit Ausnahme der in den folgenden Abschnitten aufgeführten Besonderheiten (Abschnitt 5.1) bzw. Probleme (Abschnitt 5.2) konnte die Implementierung des SYCL-Backends für Alpaka recht einfach durchgeführt werden. Als Vorlage für dessen Aufbau dienten die bereits vorhandenen Backends, wobei hier besonders das CUDA-Backend hervorzuheben ist. Die Abbildung der in Alpaka vorhandenen Datenstrukturen bzw. Funktionen auf SYCL-Funktionalität ist in Tabelle 5.1 dargestellt.

5.1. Besonderheiten des SYCL-Backends

5.1.1. Queue-Verwaltung

5.1.2. Zeiger und *accessors*

Die Übergabe von Datenfeldern an Device- und Kernel-Funktionen erfolgt in Alpaka über die von C oder älterem C++ bekannten Zeiger. Pro Datenfeld erhält die Funktion üblicherweise einen Zeiger, z.B. vom Typ `float*`, und einen ganzzahligen Parameter (meist vom Typ `size_t`), der die Länge des Speicherbereichs angibt. Der device-seitige Zeiger wird über die Funktion `getPtrNative()` aus einem Alpaka-Puffer extrahiert und an den Kernel übergeben.

Diese Vorgehensweise war mit allen bisher in Alpaka vorhandenen Backends unproblematisch, wenngleich etwas altmodisch. SYCL verlangt als Parameter für die Kernel-Funktionen jedoch die eigenen `accessor`-Datenstrukturen, eine Übergabe von Zeigern an den Kernel wird in der Spezifikation explizit verboten [vgl. KRH19, S. 192]. Zwar gibt die Spezifikation an, dass sich die `accessor`-Typen implizit in reine Zeiger umwandeln lassen [vgl. KRH19, S. 27], was jedoch von keiner der verfügbaren Implementierungen unterstützt wird. Im Rahmen dieser Arbeit wurde dieser Umstand an das SYCL-Spezifikationskomitee gemeldet und infolgedessen als Fehler in der Spezifikation anerkannt, der in zukünftigen Versionen behoben sein wird (siehe die Diskussion in Anhang C.1.1).

Innerhalb eines Kernels lassen sich die `accessor`-Typen in spezielle Zeigertypen (`multi_ptr`) umwandeln, die wiederum in „reine“ Zeiger konvertierbar sind. Dadurch lässt sich die geschilderte Problematik in mehreren Stufen lösen:

Auf der Host-Seite wird beim Aufruf der Funktion `getPtrNative()` eine Datenstruktur angelegt, die den SYCL-Puffer kapselt und sich aus Sicht des Programmierers weitestgehend wie ein Zeiger verhält. Daneben enthält diese Struktur ein spezielles Attribut mit dem Namen `is_alpaka_sycl_buffer_wrapper`. (Der Quelltext dieser Struktur ist in Anhang A.2.1 zu finden.)

Dieses Attribut wird bei der Nutzung des Kernel-Objekts verwendet, um die Puffer-Strukturen von den restlichen Parametern abzugrenzen. Strukturen, die dieses Attribut besitzen, werden an eine spezielle Funktion weitergereicht, die den Zusammenhang zwischen SYCL-Puffer und SYCL-accessor herstellt. Alle anderen Datentypen werden unverändert zurückgegeben. Da Techniken der Template-Meta-Programmierung verwendet werden, finden diese Transformationen zur Compile-Zeit statt. Der gesamte Vorgang ist in Quelltext 5.1 dargestellt.

```
namespace alpaka {
    namespace kernel {
        namespace sycl {
            namespace detail {
                struct general {};
                struct special : general{};
                template <typename> struct acc_t { using type = int; };

                // spezieller Fall: Puffer-Struktur
                template <typename TDim, typename TBuf,
                    typename acc_t<
                        typename TBuf::is_alpaka_sycl_buffer_wrapper>::type = 0>
                inline auto get_access(cl::sycl::handler& cgh, TBuf buf,
                                       special)
                {
                    return buf.buf.template get_access<
                            cl::sycl::access::mode::read_write,
                            cl::sycl::access::target::global_buffer>(cgh);
                }

                // allgemeiner Fall: kein Puffer
                template <typename TDim, typename TBuf>
                inline auto get_access(cl::sycl::handler& cgh, TBuf buf,
                                       general)
                {
                    return buf;
                }

                template <typename TDim, typename... TArgs, std::size_t... Is>
                constexpr auto bind_buffers(cl::sycl::handler& cgh,
                                            std::tuple<TArgs...> args,
                                            std::index_sequence<Is...>)
                {
                    return std::make_tuple(get_access<TDim>(cgh,
                                                            std::get<Is>(args),
                                                            special{})...);
                }
            }
        }
    }
}
```

Quelltext 5.1.: Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Programmierung

Der Quelltext 5.2 zeigt die Nutzung der Template-Meta-Funktionen innerhalb der Alpaka-Kernel-Struktur: die in einem `std::tuple` gespeicherten Kernel-Parameter werden in ein SYCL-kompatibles Format umgewandelt.

```
namespace alpaka
{
    namespace kernel
    {
        template <typename TDim, typename TIdx,
                  typename TKernelFnObj, typename... TArgs>
        class TaskKernelSycl
        {
            /* weitere Attribute, Konstruktoren, etc. */

            TKernelFnObj m_kernelFnObj;
            std::tuple<TArgs...> m_args;

        public:
            inline operator()(cl::sycl::handler& cgh)
            {
                // transformiere alle Puffer-Parameter in Accessor-Parameter
                auto accessor_args = sycl::detail::bind_buffers<TDim>(
                            cgh, m_args,
                            std::make_index_sequence<sizeof...(TArgs)>{});

                /* weitere Operationen */
            }
        };
    }
}
```

Quelltext 5.2.: Nutzung der Template-Meta-Funktionen zur Umwandlung der Puffer-Wrapper in SYCL-`accessor`-Typen

Innerhalb des Kernels werden die `accessor`-Typen (über SYCLs `multi_ptr`) nach einem ähnlichen Prinzip in reine Zeiger transformiert. Die Template-Meta-Funktionen sind in Quelltext 5.3 gezeigt, deren Nutzung in Quelltext 5.4.

```
namespace alpaka {
    namespace kernel {
        namespace sycl {
            namespace detail {
                struct general {};
                struct special : general{};
                template <typename> struct acc_t { using type = int; };

                // spezieller Fall: Accessor
                template <typename TAccessor,
                    typename acc_t<decltype(
                        std::declval<TAccessor>().get_pointer())>::type = 0>
                inline auto get_pointer(TAccessor accessor, special)
                {
                    return static_cast<typename TAccessor::value_type*>(
                                                    accessor.get_pointer());
                }

                // allgemeiner Fall: kein Accessor
                template <typename TAccessor>
                inline auto get_pointer(TAccessor accessor, general)
                {
                    return accessor;
                }

                template <typename... TArgs, std::size_t... Is>
                constexpr auto transform(std::tuple<TArgs...> args,
                                         std::index_sequence<Is...>)
                {
                    return std::make_tuple(get_pointer(std::get<Is>(args),
                                                       special{})...);
                }
            }
        }
    }
}
```

Quelltext 5.3.: Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Programmierung

```
namespace alpaka
{
    namespace kernel
    {
        template <typename TDim, typename TIdx,
                  typename TKernelFnObj, typename... TArgs>
        class TaskKernelSycl
        {
            /* weitere Attribute, Konstruktoren, etc. */

        public:
            inline operator()(cl::sycl::handler& cgh)
            {
                /* vorherige Operationen */

                cgh.parallel_for<sycl::detail::kernel<TKernelFnObj>>(
                    cl::sycl::nd_range<TDim::value> {
                        global_size, local_size
                    },
                [=](cl::sycl::nd_item<TDim::value> work_item)
                {
                    auto pointer_args = transform(
                                accessor_args,
                                std::make_index_sequence<sizeof...(TArgs)>{});

                    /* weitere Operationen und Alpaka-Kernel-Aufruf */
                }
            }
        };
    }
}
```

Quelltext 5.4.: Nutzung der Template-Meta-Funktionen zur Umwandlung der `accessor`-Typen in Zeiger

Letztere werden dann an den vom SYCL-Kernel aufgerufenen Alpaka-Kernel übergeben, sodass der Programmierer in seinem Kernel nur die reinen Zeiger vorfindet. Dadurch lässt sich das SYCL-Backend analog zu den restlichen Implementierungen verwenden.

5.1.3. Dynamischer geteilter Speicher

5.2. Probleme

Während der Implementierung traten einige Probleme auf, die sich vornehmlich auf gravierende konzeptionelle Unterschiede zwischen Alpaka und SYCL zurückführen lassen. Diese werden in den folgenden Abschnitten näher beschrieben.

5.2.1. Globale Variablen

5.2.2. Device-Verwaltung

5.2.3. Speicherverwaltung

5.2.4. Event-System

Alpaka übernimmt viele Konzepte des CUDA-API, darunter auch das Event-System. Durch CUDA-*events* wird dem Programmierer eine weitere Synchronisationsmöglichkeit eröffnet. Diese können in einem *stream* vor oder nach asynchronen Operationen – wie etwa Kopiervorgängen oder dem Starten eines Kernels – einsortiert werden. Der Programmierer kann dann später oder parallel in einem anderen *stream* abfragen, ob das jeweilige *event* bereits erreicht wurde und gegebenenfalls darauf warten. Darüber hinaus ermöglichen *events* ein simples Profiling, da z.B. die Zeitspanne zwischen verschiedenen *events* gemessen werden kann.

SYCL kennt ebenfalls ein *event*-Konzept, das sich jedoch von CUDAs bzw. Alpakas System unterscheidet. Auch SYCL-*events* lassen sich für einfaches Profiling nutzen, ermöglichen dem Programmierer jedoch nicht, eine weitere SYCL-*queue* (dem Gegenstück zu CUDA-*streams*) auf ein bestimmtes *event* zu warten.

Dies ist für die Implementierung der Alpaka-*events* ein Problem, da hier CUDAs Verhalten simuliert wird. Zur Zeit ist der Befehl `alpaka::wait::waiterWaitFor()` für die *event*-basierte Synchronisation in Alpaka nicht implementiert, wenn der *waiter* eine Alpaka-*queue* ist.

5.2.5. Statischer geteilter Speicher

Einige Plattformen der parallelen Programmierung, wie etwa CUDA und OpenCL, kennen das Konzept eines Speichers auf Multiprozessor-Ebene, der ungefähr einem programmierbaren L1-Cache entspricht. Dieser Speicher nennt sich im CUDA-Umfeld *shared memory*, während er bei OpenCL (und SYCL) *local memory* heißt. Alpaka übernimmt für dieses Speicherkonzept die CUDA-Terminologie. *Shared memory* steht allen *threads* auf der *block*-Ebene zur Verfügung und bietet deutlich schnellere Zugriffszeiten als der globale Speicher.

Es gibt zwei mögliche Arten, Speicher dieses Typs zu reservieren: *dynamisch*, das heißt außerhalb des Kernel-Codes und zur Laufzeit, sowie *statisch*, das heißt innerhalb des Kernel-Codes und mit einer zur Compile-Zeit feststehenden Größe.

Alpaka stellt für beide Varianten eine Schnittstelle bereit. Für den dynamischen Fall muss der Programmierer das *type trait* `alpaka::kernel::traits::BlockSharedMemDynSizeBytes` auf der Host-Seite für seine Anwendung implementieren. Innerhalb des Kernels kann er dann über die Alpaka-Funktion `alpaka::block::shared::dyn::getMem()` auf den Zeiger zum so reservierten geteilten Speicher zugreifen. Dieser Fall lässt sich auch für SYCL implementieren, indem man die in Abschnitt 5.1.2 vorgestellten Umwandlungen von *accessor*-Typen in *Zeiger* anwendet.

Für den statischen Fall existiert die Funktion `alpaka::block::shared::st::allocVar()`, die innerhalb des Kernels aufgerufen wird und eine beliebige Variable im geteilten Speicher ablegt. Diese Funktion kann für SYCL nicht implementiert werden, da die SYCL-Spezifikation dies (im Gegensatz zu OpenCL) bis auf einen bestimmten Sonderfall [siehe KRH19, Abschnitt 4.8.5.3] nicht vorsieht.

Diese Einschränkung erklärt sich dadurch, dass SYCL mit dem Anspruch entworfen wurde, von jedem beliebigen modernen C++-Compiler übersetzt werden zu können, auch wenn dieser keine Unterstützung für OpenCL- und/oder SYCL-Konzepte mit sich bringt. In diesem Fall generiert der Compiler wie bei jedem anderen C++-Programm normalen CPU-Maschinencode. Der Umfang des statischen geteilten Speichers steht zwar bereits zur Compile-Zeit fest und

kann daher schon vor dem Aufruf des Kernels alloziert werden. Der Zeiger auf diesen Speicherbereich kann durch einen C++-Compiler ohne SYCL-Unterstützung dem betreffenden Kernel vor dessen Ausführung jedoch gar nicht zugeordnet werden, da für diese Funktion noch kein Stapelrahmen existiert. Vom SYCL-Spezifikationskomitee wird jedoch zur Zeit untersucht, inwieweit diese Funktionalität trotzdem verfügbar gemacht werden kann (vgl. die GitHub-Diskussion mit Mitgliedern des SYCL-Spezifikationskomitees im Anhang C.1.2).

5.2.6. Atomare Funktionen

Durch die Nutzung von Zeigern kommt es zu gravierenden Problemen mit atomaren Funktionen. SYCLs vorhandene atomare Funktionen erfordern zwingend die Nutzung der `multi_ptr`-Strukturen, welche wiederum mit ihrem Adressraum verknüpft werden – das heißt, sie werden eindeutig dem globalen, lokalen oder konstanten Adressraum zugeordnet. Da in Alpaka jedoch nur C-Zeiger verwendet werden, ist diese Information innerhalb der Alpaka-Hierarchie nicht mehr vorhanden. Dadurch kann in der Implementierungsschicht der Alpaka-Funktionen kein lokaler `multi_ptr` angelegt werden.

Erschwerend kommt hinzu, dass auch die vorhandenen SYCL-Compiler nicht in der Lage sind, den Adressraum selbst zu ermitteln. Diese Problematik wurde im Rahmen dem SYCL-Spezifikationskomitee im Rahmen der Arbeit gemeldet, bisher gibt es jedoch noch keinen Lösungsansatz (vgl. die GitHub-Diskussion mit Mitgliedern des SYCL-Spezifikationskomitees im Anhang ??).

Zur Zeit wird in der SYCL-Implementierung der atomaren Funktionen daher angenommen, dass jeder übergebene Zeiger dem globalen Adressraum angehört – atomare Funktionen im lokalen oder konstanten Adressraum werden deshalb nicht unterstützt.

5.2.7. FPGA-Erweiterungen

Wie SYCL-FPGA-Erweiterungen nutzen?

5.2.8. Zufallszahlen und Zeit

Da SYCL erst wenige Jahre alt ist, ist das zugehörige Ökosystem noch nicht sonderlich stark ausgeprägt. So fehlt auch Funktionalität, die auf Seiten des *Devices* die Generierung von Zufallszahlen oder Zeitmessung erlauben. Alpaka stellt diese Funktionen jedoch zur Verfügung. Eine Lösung für dieses Problem konnte im Rahmen dieser Arbeit nicht gefunden werden, diese Funktionalität fehlt daher für das SYCL-Backend.

Bereich	Alpaka	SYCL
Attribute	`ALPAKA_FN_HOST`	nicht nötig
	`ALPAKA_FN_ACC`	nicht nötig
	`ALPAKA_FN_HOST_ACC`	nicht nötig
Adressräume	`block::shared::st::allocVar()`	siehe Abschnitt 5.2.5
	`ALPAKA_STATIC_ACC_MEM_CONSTANT`	siehe Abschnitt 5.2.1
	`ALPAKA_STATIC_ACC_MEM_GLOBAL`	siehe Abschnitt 5.2.1
Indizes	`getIdx<Block, Threads>()`	`nd_item::get_local_id()`
	`getIdx<Grid, Blocks>()`	`nd_item::get_group()`
	`getWorkDiv<Block, Threads>()`	`nd_item::get_local_range()`
	`getWorkDiv<Grid, Blocks>()`	`nd_item::get_group_range()`
Typen	`Vec<TDim, TVal>`	`nd_range<TDim>`
Devices	`reset(device)`	siehe Abschnitt 5.2.2
	`wait(device)`	siehe Abschnitt 5.2.2
	`getDevCount()`	siehe Abschnitt 5.2.2
	`getAccDevProps()`	`device::get_info()`
Queues	`QueueBlocking`	`queue`
	`QueueNonBlocking`	`queue`
	`empty(queue)`	siehe Abschnitt 5.1.1
	`wait(queue)`	`queue::wait_and_throw()`
	`wait(queue, event)`	siehe Abschnitt 5.2.4
Events	`Event`	`event`
	`test(event)`	`event::get_info()`
	`enqueue(queue, event)`	siehe Abschnitt 5.2.4
	`wait(event)`	`event::wait_and_throw()`
Speicher	`alloc()`	`buffer`
	`copy()` (synchron)	`handler::copy()`
	`copy()` (asynchron)	`handler::copy()`
	`set()` (synchron)	`handler::fill()`
	`set()` (asynchron)	`handler::fill()`
	`getMemBytes()`	`device::get_info()`
	`getFreeMemBytes()`	siehe Abschnitt 5.2.3
Kernel	`exec(queue, workDiv, kernel, ...)`	`queue.enqueue()`
	`BlockSharedExternMemSizeBytes`	siehe Abschnitt 5.1.3

Tabelle 5.1.: Implementierung der Alpaka-Funktionalität durch SYCL.

6. Messergebnisse

6.1. Methoden

6.1.1. Verwendete Hard- und Software

6.1.2. Beispielalgorithmus

[Ben18]

$$C[i] = A[i] + B[i] \tag{6.1}$$

6.2. Ergebnisse

7. Fazit

7.1. Zusammenfassung

7.2. Ausblick

Literatur

[Alp] Aksel Alpay. *hipSYCL – Implementation of SYCL 1.2.1 over AMD HIP/NVIDIA CUDA*. URL: https://github.com/illuhad/hipSYCL (besucht am 19. 08. 2019).

[ARG17] José Ignacio Aliaga, Ruymán Reyes und Mehdi Goli. „SYCL-BLAS: Leveraging Expression Trees for Linear Algebra". In: *Proceedings of 5th International Workshop on OpenCL*. Mai 2017. DOI: 10.1145/3078155.3078189.

[Bad+] Alexey Bader u. a. *Intel Project for LLVM technology*. URL: https://github.com/intel/llvm (besucht am 19. 08. 2019).

[Ben18] Sebastian Benner. „Parallelisierung des datenintensiven Kalibrierungsalgorithmus für den Röntgenstrahlen-Pixeldetektor ‚Jungfrau'". Bachelorarbeit. Fakultät Informatik, Helmholtzstraße 10, 01069 Dresden: Technische Universität Dresden, 2018.

[Bur+19] Rod Burns u. a. „Accelerated Neural Networks on OpenCL Devices Using SYCL-DNN". In: *Proceedings of the 7th International Workshop on OpenCL*. Mai 2019. DOI: 10.1145/3318170.3318183.

[Chu+18] Eric Chung u. a. „Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* Jahrgang 38.Ausgabe 2 (März 2018), S. 8–20. DOI: 10.1109/MM.2018.022071131.

[CK17] Marcin Copik und Hartmut Kaiser. „Using SYCL as an Implementation Framework for HPX.Compute". In: *Proceedings of the 5th International Workshop on OpenCL*. Mai 2017. DOI: 10.1145/3078155.3078187.

[Coda] Codeplay Software Ltd. *ComputeCpp*. URL: https://www.codeplay.com/products/computesuite/computecpp (besucht am 19. 08. 2019).

[Codb] Codeplay Software Ltd. *sycl.tech*. URL: http://sycl.tech (besucht am 19. 08. 2019).

[DKO17] Anastasios Doumoulakis, Ronan Keryell und Kenneth O'Brien. „SYCL C++ and OpenCL interoperability experimentation with triSYCL". In: *Proceedings of the 5th International Workshop on OpenCL*. Mai 2017. DOI: 10.1145/3078155.3078188.

[Dua+18] Javier Duarte u. a. „Fast inference of deep neural networks in FPGAs for particle physics". In: *Journal of Instrumentation* Jahrgang 13 (Ausgabe 07 Juli 2018). DOI: 10.1088/1748-0221/13/07/p07027.

[Fif+16] Jeff Fifield u. a. „Optimizing OpenCL applications on Xilinx FPGA". In: *Proceedings of the 4th International Workshop on OpenCL*. Apr. 2016. DOI: 10.1145/2909437.2909447.

[Fir+18] Daniel Firestone u. a. „Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation*. Apr. 2018, S. 51–64.

[Fow+18] Jeremy Fowers u. a. „A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. Juni 2018, S. 1–14. DOI: 10.1109/ISCA.2018.00012.

[HKB19] Jeff R. Hammond, Michael Kinsner und James Brodman. „A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications". In: *Proceedings of the 7th International Workshop on OpenCL*. Mai 2019. DOI: 10.1145/3318170.3318193.

[HS10] Charles Hawkins und Jaume Segura. *Introduction to Modern Digital Electronics*. Preliminary Edition. SciTech Publishing, Inc., 2010. ISBN: 978-1-891-12107-4.

[KB13] Frank Kesel und Ruben Bartholomä. *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs – Einführung mit VHDL und SystemC*. 3. Auflage. Oldenbourg Verlag, 2013. ISBN: 978-3-486-73181-1.

[Ker+] Ronan Keryell u. a. *triSYCL – Generic system-wide modern C++ for heterogeneous platforms with SYCL from Khronos Group*. URL: https://github.com/triSYCL/triSYCL (besucht am 19. 08. 2019).

[KGL] Ronan Keryell, Andrew Gozillon und Victor Lezaud. *sycl – Experimental fusion of triSYCL with Intel SYCL upstreaming effort into Clang/LLVM*. URL: https://github.com/triSYCL/sycl (besucht am 19. 08. 2019).

[Khr14] Khronos Group. „Khronos Releases SYCL 1.2 Provisional Specification". In: *The Khronos Group Inc* (19. März 2014). URL: https://www.khronos.org/news/press/khronos-releases-sycl-1.2-provisional-specification (besucht am 26. 09. 2019).

[KRH19] Ronan Keryell, Maria Rovatsou und Lee Howes, Hrsg. *SYCL™ Specification*. 9450 SW Gemini Drive #45043, Beaverton, OR 97008-6018, Vereinigte Staaten von Amerika: The Khronos Group, Apr. 2019.

[Law+79] Charles L. Lawson u. a. „Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Transactions on Mathematical Software* Jahrgang 5.Ausgabe 3 (Sep. 1979), S. 308–323.

[Mun12] Aaftab Munshi, Hrsg. *The OpenCL Specification*. Version 1.2, Revision 19. The Khronos Group. 9450 SW Gemini Drive #45043, Beaverton, OR 97009-6018, Vereinigte Staaten von Amerika, Nov. 2012.

[Pap+09] Alexandros Papakonstantinou u. a. „FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs". In: *2009 IEEE 7th Symposium on Application Specific Processors*. Juli 2009, S. 35–42. DOI: 10.1109/SASP.2009.5226333.

[Set13] Sean O. Settle. „High-performance Dynamic Programming on FPGAs with OpenCL". In: *2013 IEEE High Performance Extreme Computing Conference*. Laut Angabe im Artikel Teil des genannten Konferenzbandes, wird jedoch nicht im Inhaltsverzeichnis des besagten Bandes aufgeführt. Sep. 2013.

[Won+16] Michael Wong u. a. *Khronos's OpenCL SYCL to support Heterogeneous Devices for C++*. Vorschlag für das C++-Standardisierungsverfahren. Untergruppe EWG, Studiengruppen SG1 und SG14. Dokumentennummer P0236R0. Codeplay Software Ltd., Feb. 2016.

[Wor15] Benjamin Worpitz. „Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures“. Masterarbeit. Fakultät Informatik, Helmholtzstraße 10, 01069 Dresden: Technische Universität Dresden, Okt. 2015. DOI: 10.5281/zenodo.49768.

[Xil17] Xilinx, Inc. *UltraScale Architecture Configurable Logic Block – User Guide*. UG574 (v1.5). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Feb. 2017.

[Xil18a] Xilinx, Inc. *UltraScale Architecture Clocking Resources – User Guide*. UG572 (v1.8). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Dez. 2018.

[Xil18b] Xilinx, Inc. *Xilinx Alveo – Adaptable Accelerator Cards for Data Center Workloads*. Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, 2018.

[Xil19a] Xilinx, Inc. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. DS962 (v1.1). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Juni 2019.

[Xil19b] Xilinx, Inc. *Introduction to FPGA Design with Vivado High-Level Synthesis*. UG998 (v1.1). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Jan. 2019.

[Xil19c] Xilinx, Inc. *SDAccel Environment Profiling and Optimization Guide*. UG1207 (v2019.1). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Juni 2019.

[Xil19d] Xilinx, Inc. *SDx Pragma Reference Guide*. UG1253 (v2019.1). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Juni 2019.

[Xil19e] Xilinx, Inc. *UltraScale Architecture and Product Data Sheet: Overview*. DS890 (v3.10). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Aug. 2019.

[Xil19f] Xilinx, Inc. *UltraScale Architecture DSP Slice – User Guide*. UG579 (v1.8). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Mai 2019.

[Xil19g] Xilinx, Inc. *UltraScale Architecture Memory Resources – User Guide*. UG573 (v1.10). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Feb. 2019.

[Xil19h] Xilinx, Inc. *UltraScale Architecture SelectIO Resources – User Guide*. UG571 (v1.12). Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124, Vereinigte Staaten von Amerika, Aug. 2019.

[Žuž16] Peter Žužek. „Implementacija knjižnice SYCL za heterogeno računanje“. Masterarbeit. Kongresni trg 12, 1000 Ljubljana, Republik Slowenien: Univerza v Ljubljani, März 2016.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

A. Quelltexte

A.1. VHDL-Quelltexte

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY ff2 IS
    PORT (
        clk : IN    std_logic;
        d0  : IN    std_logic;
        d1  : IN    std_logic;
        res : IN    std_logic;
        q0  : OUT   std_logic;
        q1  : OUT   std_logic
    );
END ff2 ;

ARCHITECTURE beh OF ff2 IS
    SIGNAL q0_s, q1_s : std_logic;
BEGIN

    reg: PROCESS (clk, res)
    BEGIN
        IF res = '1' THEN
            q0_s <= '0';
            q1_s <= '0';
        ELSIF clk'event AND clk = '1' THEN
            q0_s <= d0;
            q1_s <= d1;
        END IF;
    END PROCESS reg;

    q0 <= q0_s AFTER 2 ns;
    q1 <= q1_s AFTER 2 ns;

END beh;
```

Quelltext A.1.: VHDL-Quelltext eines 2-Bit-Flipflops [siehe KB13, S. 39]

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY mux2 IS
    PORT (
        a1  : IN    std_logic;
        a2  : IN    std_logic;
        b1  : IN    std_logic;
        b2  : IN    std_logic;
        sel : IN    std_logic;
        o1  : OUT   std_logic;
        o2  : OUT   std_logic
    );
END mux2 ;

ARCHITECTURE beh OF mux2 IS
BEGIN

    mux: PROCESS (a1, a2, b1, b2, sel)
    BEGIN
        IF sel = '1' THEN
            o1 <= a1 after 3 ns;
            o2 <= a2 after 3 ns;
        ELSE
            o1 <= b1 after 4 ns;
            o2 <= b2 after 4 ns;
        END IF;
    END PROCESS mux;

END beh;
```

Quelltext A.2.: VHDL-Quelltext eines 2-Bit-Multiplexers [siehe KB13, S. 39–40]

A.2. C++-Quelltexte

A.2.1. Implementierung des SYCL-Puffer-Wrappers

```
namespace alpaka
{
    namespace mem
    {
        namespace buf
        {
            namespace sycl
            {
                namespace detail
                {
                    template <typename TBuf>
                    struct buffer_wrapper
                    {
                        using buf_type = TBuf;
                        using value_type = typename buf_type::value_type;
                        using is_alpaka_sycl_buffer_wrapper = bool;

                        buffer_wrapper(TBuf wrapped_buf) noexcept
                        : buf{wrapped_buf}
                        , dummy{std::aligned_alloc(alignof(value_type),
                                                   sizeof(std::size_t)),
                                [](void* ptr) { std::free(ptr); }}
                        {
                        }

                        operator value_type*() noexcept
                        {
                            return reinterpret_cast<value_type*>(dummy.get());
                        }

                        operator const value_type*() const noexcept
                        {
                            return reinterpret_cast<const value_type*>(
                                                                dummy.get());
                        }

                        TBuf buf;
                        std::shared_ptr<void> dummy;
                    };
                }
            }
        }
    }
}
```

Quelltext A.3.: Implementierung des SYCL-Puffer-Wrappers

B. Fehlerberichte und Korrekturen

B.1. Fehlerberichte und Korrekturen für die Xilinx-OpenCL-Laufzeitumgebung

B.2. Fehlerberichte und Korrekturen für den Xilinx-SYCL-Compiler

B.3. Fehlerberichte und Korrekturen für den Intel-SYCL-Compiler

B.4. Fehlerberichte und Korrekturen für den ComputeCpp-SYCL-Compiler

C. Online-Diskussionen

C.1. Diskussionen mit dem SYCL-Spezifikationskomitee

C.1.1. Implicit accessor-to-pointer casts

Original: `https://github.com/KhronosGroup/SYCL-Docs/issues/11`, zuletzt abgerufen am 06. November 2019.

Jan Stephan This is a duplicate of triSYCL/triSYCL#247, moving the discussion here.

The Problem The SYCL specification (3.5.2.1) says the following:

> Within kernels, accessors can be implicitly cast to C++ pointer types. The pointer types will contain a compile-time deduced address space. So, for example, if an accessor to global memory is cast to a C++ pointer, the C++ pointer type will have a global address space attribute attached to it. The address space attribute will be compile-time propagated to other pointer values when one pointer is initialized to another pointer value using a defined mechanism.

This is not reflected in `accessor`'s interface and none of the publicly available implementations support this.

Example

```
void vec_add(const int* a, const int* b, int* c, std::size_t size);

queue.submit([&](cl::sycl::handler& cgh)
{
    auto a = a_d.get_access<cl::sycl::access::mode::read>(cgh);
    auto b = b_d.get_access<cl::sycl::access::mode::read>(cgh);
    auto c = c_d.get_access<cl::sycl::access::mode::discard_write(cgh);

    cgh.single_task<class vector_add>([=]()
    {
        // no known conversion from accessor to const int*
        vec_add(a, b, c, 1024);
    });
});
```

Use case Libraries such as Alpaka or HPX provide abstraction layers over competing compute APIs such as CUDA. Unfortunately their API works with raw pointers in their abstract kernels. In order to implement a SYCL backend the `accessor` has to be explicitly transformed into a pointer right now (by going through `multi_ptr`). Being able to do this implicitly would be a lot easier.

Possible solutions

1. Remove the above wording and rely on `multi_ptr`'s conversion instead.
2. Allow `accessor` to be implicitly cast as well.

Ronan Keryell (Xilinx) The working group is looking at this internally.

Ruymán Reyes Castro (Codeplay) Internal issue tracker: `https://gitlab.khronos.org/sycl/Specification/issues/258`, currently assigned to Codeplay.

Ruymán Reyes Castro Decision has been to go for option 1, MR up in: #44

Gordon Brown (Codeplay) SYCL working group: The above merge request has been approved so we can close this now.

C.1.2. Why is there no way to allocate local memory inside a ND-kernel (parallel_for)?

Original: `https://github.com/KhronosGroup/SYCL-Docs/issues/20`, zuletzt abgerufen am 06. November 2019.

Jan Stephan I know this is possible using the hierarchical `parallel_for` invoke, but we can't do it with the `nd_item` version despite being able to specify the group size. OpenCL allows this (AFAIK), SYCL's competitors do, too, so why not allow it in SYCL? I suppose there must be a reason for leaving it out.

Victor Lomuller (Codeplay) The main reason is the host device. If your compiler is not SYCL aware, you need to be able to preallocate this memory before calling the functor, which is not trivial without compiler support.

Jan Stephan Wouldn't this be solvable by doing the inverse of the hierarchical case? I.e. everything is `private` by default if declared inside the kernel, unless embedded with something like `cl::sycl::local_memory`.

Victor Lomuller It does not address the compiler support problem. You still need to preallocate memory before calling the functor, but without compiler support, you cannot know the amount of memory you need nor where to place the pointer to that memory (the stack frame does not exist yet).

Ronan Keryell (Xilinx) which SYCL competitor can run on CPU without a specific compiler? This allows for example to use HellGrind & ThreadSanitizer with plain GCC or Clang to debug a SYCL program just by running it on my laptop. I find this an amazing feature of SYCL...

Gordon Brown (Codeplay) SYCL working group: We agree this would be a useful feature to have, we are investigating how to potentially support this internally.

C.1.3. How to extract address space from raw pointers?

Original: `https://github.com/KhronosGroup/SYCL-Docs/issues/21`, zuletzt abgerufen am 06. November 2019.

Jan Stephan Imagine a device-side function with the following signature:

```
void foo(int* vec);
```

I don't know if `vec` comes from global, local, constant or private memory. However, inside `foo` I'd like to do something to `vec` which requires me to know the address space of the pointer, e.g. a `cl::sycl::atomic_fetch_add`. How do I tell the `multi_ptr` / `atomic` inside `foo` which address space is needed? Simply using a `global_ptr` will break if `vec` actually resides in local memory. Using `multi_ptr` will fail because the address space template parameter is missing. Creating an `atomic` by passing `vec` to its constructor will fail because `vec` isn't a `multi_ptr`. Using `atomic_fetch_add` on `vec` will fail because `vec` isn't an `atomic` type.
Some implementations (like ComputeCpp) internally use `__global` to annotate the pointer during device compilation. But even if there was a way to write something like `void foo(__global int* vec)` (there isn't as far as I know, ComputeCpp complains if I do this) this would be a bad idea because the address space attributes are implementation-defined.
Why do we need this? Sadly, there are libraries / frameworks out there that pass around raw pointers but where a SYCL backend is planned / worked on.
Edit: I also tried to overload `foo` with `global_ptr`, `local_ptr` etc. directly. This will fail because the call is ambigous.

Ronan Keryell (Xilinx) Interestingly, Intel is trying hard to hide what you are asking for: `intel/llvm#348`[1]. Can you imagine an API that could be added to the standard?

Jan Stephan An easy solution that doesn't require an API change would be to correctly deduce the overloads, i.e. `foo(global_ptr)`, `foo(local_ptr)` and so on. This is not very intuitive, though, and might break user APIs.
From the programmer's point of view it would be preferable to allow `multi_ptr` construction on raw pointers without having to specify the address space. The compiler should be able to figure this out by itself since it knows about the address spaces anyway.
On the other hand it should raise an error if the programmer tries to assign a raw pointer in local space to a `global_ptr`. Currently this doesn't happen, both the Intel and ComputeCpp compiler will happily compile if I pass the same pointer to `global_ptr`'s and `local_ptr`'s constructor.
Admittedly I haven't given this much thought yet (I only encountered the problem on Wednesday), I'll try to think this through on the weekend.

Jan Stephan The weekend has passed... Apart from the solutions above the best I could come up with is something like `cl::sycl::pointer_traits` to be added to the specification. The interface would look something along the lines of

[1] Verweis auf Änderung des Intel-SYCL-Compilers, J.S.

```
template <typename Ptr>
struct pointer_traits
{
    static_assert(is_raw_ptr_type(Ptr), "Ptr needs to be a raw pointer type");
    using pointer_t = /* implementation-defined */ Ptr;
    using address_space = /* implementation-defined */;
    // maybe add other traits here
};
```

Since the compiler needs to figure out the address space on its own anyway (if I understand Section 6.8 correctly), it would fill out the `implementation-defined` parts. A programmer could then use SFINAE or `if constexpr` to adapt to the different address spaces.
This is basically the problem `multi_ptr` tries to solve, it already encapsulates the functionality above. However, `multi_ptr` requires the user to specify the address space before using it. This makes sense because we can request a `multi_ptr` from a buffer accessor, a local accessor, and so on and the `multi_ptr` data structure has to know about its address space. It also renders us unable to construct it from a pointer we don't know the address space of.
So my straight-forward resolution still is to remove the requirement to specify the address space for the `multi_ptr` type. Instead the compiler needs to figure out the correct value for the `address_space` member of `multi_ptr` (or the `Space` template parameter). If this is not an option because of implications I'm not aware of (and I'm sure there are plenty) I'd shoot for the `pointer_traits` option.

Ronan Keryell

> Since the compiler needs to figure out the address space on its own anyway (if I understand Section 6.8 correctly), it would fill out the implementation-defined parts. A programmer could then use SFINAE or `if constexpr` to adapt to the different address spaces.

The problem is that this address space resolution can be done in LLVM or even in the SPIR-V backend or whatever... So you might not have this information inside Clang as a type trait... :-(`multi_ptr` was designed:

- to avoid requiring this kind of address-space inference by avoiding using raw pointers. Of course this means passing around the `multi_ptr` type. But with `auto` nowadays it is easier;
- to provide a way to interoperate with existing OpenCL C kernel code. But since there is no type inference in OpenCL C either, you have to do an explicit dispatch yourself from the `multi_ptr` to call an OpenCL function with different version and different names for each possible address-space...

Jan Stephan

> The problem is that this address space resolution can be done in LLVM or even in the SPIR-V backend or whatever... So you might not have this information inside Clang as a type trait... :-(

I have to admit that my knowledge about compiler construction is a bit limited. But the backends will have to look up this information, too - why can't the frontend do the same?

> to avoid requiring this kind of address-space inference by avoiding using raw pointers. Of course this means passing around the multi_ptr type. But with auto nowadays it is easier;

While I can understand this intent with regard to new code I believe this is an oversight if we consider legacy code bases. If those have a raw pointer API the design of `multi_ptr` or the lack of a feature to otherwise extract the address space becomes a major obstacle.

C.2. Diskussionen mit Xilinx-Angestellten

C.3. Diskussionen mit Codeplay-Angestellten

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Entwicklung eines SYCL-Backends für die Alpaka-Bibliothek und dessen Evaluation mit Schwerpunkt auf FPGAs* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 16. Dezember 2019

Jan Stephan