



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

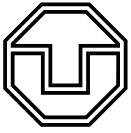
Prof. Dr. rer. nat. Ulrich Schramm

Betreuer

Dr.-Ing. Oliver Knodel

Matthias Werner, M.Sc.

Eingereicht am: 28. November 2019



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

In dieser Arbeit wird ein SYCL-Backend für die Parallelisierungsbibliothek *Alpaka* entwickelt. Besondere Aufmerksamkeit kommt dabei FPGAs zu, die mit ihren inhärenten Hardware- und Parallelisierungseigenschaften eine interessante Plattform für Alpaka- und SYCL-Programme sind. Das entwickelte Backend wird anhand eines Alpaka-basierten Photonenzählers aus dem Umfeld der Laser-Teilchenbeschleunigung evaluiert. Außerdem werden die zur Zeit verfügbaren SYCL-Implementierungen hinsichtlich ihres Entwicklungsstandes und ihrer Kompatibilität mit dem Alpaka-SYCL-Backend untersucht.

Abstract

This thesis shows the development of a SYCL backend for the parallelization library *Alpaka*. Special attention is given to FPGAs which are an interesting platform for both Alpaka and SYCL programs due to their inherent hardware and parallelization properties. The SYCL backend is evaluated against an Alpaka-based photon counter which is used in the field of laser particle acceleration. In addition, the currently available SYCL implementations are investigated with regard to their state of development and their compatibility with Alpaka's SYCL backend.

Inhaltsverzeichnis

Glossar	9
Abkürzungsverzeichnis	11
1. Einleitung	13
1.1. Forschungsstand	15
1.2. Ziel der Arbeit	16
2. FPGAs als Beschleuniger	17
2.1. Überblick	17
2.1.1. Definition	17
2.1.2. Aufbau moderner FPGAs	18
2.1.3. Anwendungsfälle	20
2.2. Entwicklungsprozess	21
2.2.1. Hardware-Beschreibungssprachen	21
2.2.2. High-Level-Synthese und Parallelität	27
3. Der SYCL-Standard	31
3.1. Überblick	31
3.1.1. Der AXPY-Algorithmus als Beispiel	32
3.1.2. Struktur eines SYCL-Programms	32
3.1.3. Beschleunigerwahl und Befehlswarteschlange	33
3.1.4. Speicherreservierung und -initialisierung	34
3.1.5. Kerneldefinition und -ausführung	35
3.1.6. Synchronisierung	36
3.1.7. Zusammenfassung	36
3.2. Weiterführende Konzepte	38
3.2.1. Hardware-Abstraktion	38
3.2.2. Kernel-Start	39
3.2.3. Abhängigkeiten zwischen Kernen	40
3.2.4. Fehlerbehandlung	41
3.2.5. Profiling	41
3.2.6. Referenz-Semantik	41
3.3. Implementierungen	42
3.3.1. ComputeCpp	42
3.3.2. Intel	42
3.3.3. triSYCL	42

3.3.4.	hipSYCL	42
3.3.5.	sycl-gtx	43
3.3.6.	Zusammenfassung	43
3.4.	Erweiterungen für FPGAs	43
4.	Die Alpaka-Bibliothek	47
4.1.	Überblick	47
4.1.1.	Alpaka am Beispiel des AXPY-Algorithmus	48
4.2.	Weiterführende Konzepte	55
4.2.1.	Hardware-Abstraktion	55
4.2.2.	Speicherhierarchie	58
4.2.3.	Abhängigkeiten zwischen Kernen	59
4.2.4.	Fehlerbehandlung	59
4.2.5.	Profiling	59
4.2.6.	Zusammenfassung	60
5.	Implementierung des SYCL-Backends der Alpaka-Bibliothek	63
5.1.	Device-Verwaltung	63
5.1.1.	Fehlende Device-Informationen	64
5.1.2.	Fehlende Device-Funktionen	64
5.2.	Queues und Events	65
5.2.1.	Event-Inkompatibilitäten	65
5.3.	Tasks	65
5.3.1.	Kernel-Tasks	66
5.3.2.	Kopier- und memset-Tasks	70
5.4.	Work-Division	71
5.5.	Geteilter Speicher	71
5.5.1.	Dynamischer, geteilter Speicher	72
5.5.2.	Statischer, geteilter Speicher	72
5.6.	Globale Variablen	72
5.7.	Mathematikfunktionen	73
5.8.	FPGA-Erweiterungen	73
5.9.	Zufallszahlen und Zeit	73
5.10.	Zusammenfassung	73
6.	Ergebnisse	75
6.1.	Nutzbarkeit der SYCL-Implementierungen	75
6.1.1.	ComputeCpp	75
6.1.2.	Intel	76
6.1.3.	Xilinx	76
6.1.4.	Block RAM und Pipelining	77
6.1.5.	Zusammenfassung	78
6.2.	Vergleich zwischen Alpaka und SYCL	78
6.3.	Verifizierung des SYCL-Alpaka-Backends	79
6.3.1.	Der <i>jungfrau-photoncounter</i>	79
6.3.2.	Verifizierung und Performanz	79
6.3.3.	Nutzbarkeit von FPGAs	80
6.4.	Box-Filter	80
6.4.1.	Algorithmus	82
6.4.2.	Messergebnisse	82
7.	Fazit	87

Literatur	89
A. Quelltexte	99
A.1. VHDL-Quelltexte	99
A.2. C++-Quelltexte	101
A.2.1. Verwendung von SYCL-Ausnahmefehlern	101
A.2.2. Verwendung des SYCL-Profilings	102
A.2.3. Implementierung des TaskKernelSycl-Konstruktors	102
A.2.4. Implementierung des SYCL-Puffer-Wrappers	103
A.2.5. Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta- Programmierung	104
A.2.6. Box-Filter-Kernel	105
B. Online-Diskussionen	107
B.1. Diskussionen mit dem SYCL-Spezifikationskomitee	107
B.1.1. Implicit accessor-to-pointer casts	107
B.1.2. Why is there no way to allocate local memory inside a ND-kernel (paral- lel_for)?	108
B.1.3. How to extract address space from raw pointers?	110
B.2. Diskussionen mit Xilinx-Angestellten	112
Danksagung	113
Selbstständigkeitserklärung	115

Glossar

Alpaka C++-Abstraktionsbibliothek für die Beschleunigerprogrammierung

Architecture Verhaltensbeschreibung einer VHDL-Komponente

BLAS *Basic Linear Algebra Subprograms*, Fortran-Bibliothek für Operationen der linearen Algebra

CUDA GPGPU-Sprache der Firma NVIDIA

Device Alpaka- und SYCL-Bezeichnung für einen Beschleuniger

Entity Schnittstellenbeschreibung einer VHDL-Komponente

Flip-Flop Elektronische Schaltung mit zwei Zuständen des Ausgangssignals, die ein Bit beliebig lange speichern kann. Die Zustandsänderung erfolgt durch Ereignisse, nicht durch die Zeit.

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

ISO *International Organization for Standardization*. Kurzname von griechisch ίσος („gleich“). Internationale Vereinigung von Normungsorganisationen, umfasst unter anderem das Deutsche Institut für Normung (DIN)

Kernel Programm, das auf einem Beschleuniger ausgeführt wird.

Latch Sonderform des Flip-Flops. Kann den Zustand von der Anfangs- bis zur Endflanke des Taktschrittes ändern.

OpenCL *Open Computing Language*, standardisierte C-Schnittstelle des Industriekonsortiums *Khronos* für die Programmierung von Beschleunigern

OpenMP *Open Multi-Processing*, standardisierte C, C++- und Fortran-Schnittstelle des Industriekonsortiums *OpenMP Architecture Review Board* für die parallele Programmierung auf Thread-Ebene

SPIR *Standard Portable Intermediate Language*, standardisierte Zwischencode-Sprache des Industriekonsortiums *Khronos*

SYCL standardisierte C++-Schnittstelle des Industriekonsortiums *Khronos* für die Programmierung von Beschleunigern

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

Verilog Hardware-Beschreibungssprache

VHDL *VHSIC Hardware Description Language*, Hardware-Beschreibungssprache

Akronyme

ALU *arithmetic logic unit*

API *application programming interface*

APU *accelerated processing unit*

ASIC *application-specific integrated circuit*

CLB *configurable logic block*

CMT *clock management tile*

CPU *central processing unit*

CU *compute unit*

DRAM *dynamic random-access memory*

DSP *digital signal processor*

FIFO *first in, first out*

FPGA *field programmable gate array*

GPGPU *general-purpose computing on graphics processing units*

GPU *graphics processing unit*

HLS *High-Level-Synthese*

HPC *high-performance computing*

I/O *input/output*

IC *integrated circuit*

II *initiation interval*

IOB *input/output block*

LUT *Lookup-Tabelle*

Abkürzungsverzeichnis

PE *processing element*

RAM *random-access memory*

SIMD *single instruction, multiple data*

SLR *super logic region*

VHSIC *very high speed integrated circuit*

1. Einleitung

Die Hardware-Ausstattungen moderner Computersysteme werden zunehmend heterogen. Mobiltelefone und PCs verfügen nicht nur über mehrkernige, leistungsstarke Prozessoren (*central processing unit* (CPU)), sondern auch über dedizierte Chips zur Grafikdarstellung (*graphics processing unit* (GPU)). Währenddessen verschwimmt bei Spielekonsolen und Laptops die Unterscheidung zwischen Haupt- und Grafikprozessoren dank integrierter Chips, die beide Funktionalitäten performant abdecken können (*accelerated processing unit* (APU)). GPUs unterscheiden sich von CPUs vor allem durch einen spezifischeren Einsatzzweck: Während CPUs vielfältige verschiedene Anwendungen abdecken und diese parallel ausführen können (Taskparallelität), wurden GPUs vornehmlich für die möglichst schnelle und parallele Berechnung einer hohen Zahl von Bildpunkten konzipiert. Datenparallele Algorithmen – also Algorithmen, die parallel auf viele gleichartige Daten angewendet werden – sind also gut dafür geeignet, auf GPUs ausgelagert und von diesen berechnet zu werden. Anders ausgedrückt lassen sich GPUs als *Beschleuniger* für datenparallele Algorithmen verwenden. Im Bereich des *high-performance computing* (HPC) sind verschiedene Beschleunigertypen seit jeher im Einsatz, seien es CPUs mit besonders vielen Rechenkernen, von den GPUs abstammende Beschleuniger ohne grafische Ausgabe wie NVIDIAs Tesla-Reihe, oder Hybride wie Intels Xeon-Phi-Produkte.

Allen gemeinsam ist die Idee der Beschleunigung eines Algorithmus durch dessen Parallelisierung, also die in der Regel parallele Ausführung einzelner Schleifeniterationen auf den in den Beschleunigern vorhandenen Rechenkernen. 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.

Einen Chip, der für einen speziellen Algorithmus entworfen wurde, bezeichnet man als *application-specific integrated circuit* (ASIC). ASICs erfordern durch ihre Produktionskosten einen hohen finanziellen Aufwand und sind daher erst ab hohen Stückzahlen interessant. Darüber hinaus bergen sie das Risiko, geänderten Einsatzzwecken – z.B. durch beim Schaltungsentwurf nicht bedachte Aspekte oder später geänderte Anforderungen – nicht mehr zu genügen, was wiederum eine aufwendige Neuentwicklung und -produktion nötig macht. Im schlimmsten Fall ist der zu tauschende ASIC für einen Menschen physisch nicht mehr zu erreichen, wie etwa in der Raumfahrt oder bei Detektoren innerhalb eines in Betrieb befindlichen Teilchenbeschleunigers.

Durch dynamisch rekonfigurierbare bzw. programmierbare Hardware lassen sich die beschriebenen Herausforderungen besser meistern – wenn auch mit dem Nachteil, gegenüber spezialisierten Chips deutlich langsamer zu sein. Auf diesem Gebiet ist vor allem der Hardware-Typ *field programmable gate array* (FPGA) zu nennen. Gegenüber den oben genannten Beschleunigern und spezialisierten Chips haben FPGAs den Vorteil, dass die einzelnen Chip-

Bestandteile – also Logikfunktionen, On-Chip-Speicher usw. – weitestgehend frei verschaltbar sind. 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 Werkzeuge für die *High-Level-Synthese* (HLS), die Algorithmen in Schaltungen umwandeln können, wurde der Entwicklungsprozess in jüngerer Zeit aber deutlich vereinfacht.

Dadurch sind FPGAs auch für Anwendungen abseits des klassischen Schaltungsentwurfs interessant. Insbesondere latenzkritische Anwendungen eignen sich gut für den Einsatz von FPGAs, da der Zeitpunkt der Ergebnisausgabe taktgenau vorhersagbar ist. Die Inferenz neuronaler Netzwerke oder die Verarbeitung großer Datenströme im HPC-Bereich sind Beispiele für *stream*-artige Probleme, die vom Einsatz eines FPGAs profitieren können.

Darüber hinaus sind FPGAs gegenüber GPUs noch in anderen Punkten flexibler. Da Schaltungen weitgehend frei entworfen werden können, lassen sich Schnittstellen zu externer Hardware mit deutlich kürzeren Kommunikationswegen entwerfen, als dies für GPUs möglich wäre. Kürzere Kommunikationswege bedeuten auch eine geringere Störanfälligkeit gegenüber Strahlung, was FPGAs z.B. für den Einsatz in Experimenten der Teilchenphysik interessant macht. Auch der bei GPUs durch das Betriebssystem und die Hardware-Treiber vorhandene Overhead lässt sich bei FPGAs vermeiden – sofern keine Abhängigkeit des Algorithmus oder weiterer Hardware-Komponenten zu Software besteht – da die Schaltung selbst von Software unabhängig ist.

Da die Synthese einer Schaltung je nach Komplexität des Algorithmus einige Stunden bis Tage benötigen kann, ist für das *Prototyping* ratsam, schnellere Methoden für die inkrementelle Entwicklung zu wählen – also z.B. Beschleuniger. Wünschenswert ist hier vor allem Portabilität zwischen den verschiedenen Hardware-Plattformen.

Das ist durchaus nicht unproblematisch: So waren in den Anfangsjahren des *general-purpose computing on graphics processing units* (GPGPU) nur die Produkte des Herstellers NVIDIA ohne Umwege über Grafikschnittstellen programmierbar, erforderten dafür aber die Nutzung des erstmals 2006 erschienenen NVIDIA-eigenen CUDA-API. Dieses war (und ist) jedoch nicht mit Konkurrenzprodukten kompatibel. Für CPUs etablierte sich schnell die seit 1997 entwickelte Schnittstelle *Open Multi-Processing* (OpenMP) als Mittel der Wahl für die Nutzung mehrerer Rechenkerne. Die Vektorisierung erfordert dagegen entweder befehlssatzspezifische Erweiterungen – die auch innerhalb derselben Befehlssatzfamilie mitunter inkompatibel sind – oder sehr gut optimierende Compiler. Die seit 2011 verfügbaren HLS-Werkzeuge des FPGA-Herstellers Xilinx ermöglichten zwar die Nutzung der Hochsprachen, reicherten diese aber mit zahlreichen Erweiterungen an, was die Übertragung zwischen den Plattformen erschwert.

Um die Entwicklung portabler Programme zu ermöglichen, wurden früh verschiedene Ansätze entwickelt. Das Khronos-Industriekonsortium gab 2008 die Spezifikation der *Open Computing Language* (OpenCL) 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, Altera und Xilinx), flachte jedoch nach wenigen Jahren wieder ab und konnte sich auf GPUs 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 z.B. in einen äquivalenten CUDA-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 ein Backend 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 C++-Programmierschnittstelle. Der neue Standard wird unter anderem von den FPGA-Herstellern Xilinx und Intel vorangetrieben und wäre damit eine interessante Backend-Variante 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 NVIDIA-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 – gegenüber der Entwicklung auf Registerebene – durch eine Hochsprache 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 neuronaler Netzwerke, die durch automatische Synthese in Schaltungen für Xilinx-FPGAs umgewandelt werden. In diesem Projekt kommt allerdings nicht OpenCL zum Einsatz. Stattdessen werden die Xilinx-Erweiterungen für die Programmiersprache C++ verwendet. [vgl. Dua+18]

Die SYCL-Spezifikation wurde in der Literatur in den ersten Jahren ihres Bestehens vornehmlich als einfacher Überbau 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 Problemen sowie Unzulänglichkeiten des C++-Standards zuteil. [vgl. Won+16]

Aliaga *et al.* veröffentlichten 2017 eine in C++ und SYCL geschriebene Implementierung der BLAS-Schnittstelle – ein Quasistandard für Rechenoperationen der linearen Algebra –, 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 neuronalen 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]

1. Einleitung

Hinsichtlich der Verwendung von SYCL auf FPGAs ist in der Literatur – neben einigen untereinander recht ähnlichen 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 bislang 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 wird der SYCL-Standard hinsichtlich der verfügbaren Implementierungen und deren Nutzbarkeit untersucht. Dies geschieht vor allem im Hinblick auf FPGAs, die durch ihre frei veränderbare Hardware-Konfiguration in Verbindung mit einer modernen C++-Programmierschnittstelle eine vielversprechende Hardware-Plattform darstellen. Dabei wird ein experimentelles SYCL-Backend für die Alpaka-Bibliothek implementiert. Die während des Entwicklungsprozesses aufgetretenen Schwierigkeiten und Unzulänglichkeiten werden analysiert, sowie erste Einschätzungen der Leistungsfähigkeit vorgenommen.

2. FPGAs als Beschleuniger

Konzeption und Aufbau der FPGAs sowie der zugehörige Entwicklungsprozess werden in diesem Kapitel geschildert. Dabei wird auf die Vorteile der High-Level-Synthese gegenüber dem hardwarenahen Schaltungsentwurf auf der Register-Transfer-Ebene ebenso eingegangen wie auf die bei FPGAs möglichen Parallelisierungskonzepte.

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 – unter Inkaufnahme geringerer erreichbarer Taktraten und schlechterer Energieeffizienz – 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. FPGAs wurden traditionell vornehmlich im Schaltungsentwurf eingesetzt, finden mittlerweile jedoch auch in anderen Gebieten zunehmend Verwendung (siehe Abschnitt 2.1.3). [vgl. KB13, S. 208]

2. FPGAs als Beschleuniger

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 dediziertes Verbindungsnetz Logikgatter und Speicher. Dabei lassen sich vier zentrale Strukturen unterscheiden:

- Konfigurierbare Logikblöcke,
- programmierbare Verbindungen,
- Puffer für die Ein- und Ausgabe (engl. *input/output*, I/O) und
- weitere Elemente (Speicher, arithmetische Einheiten, Taktnetzwerke, usw.).

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

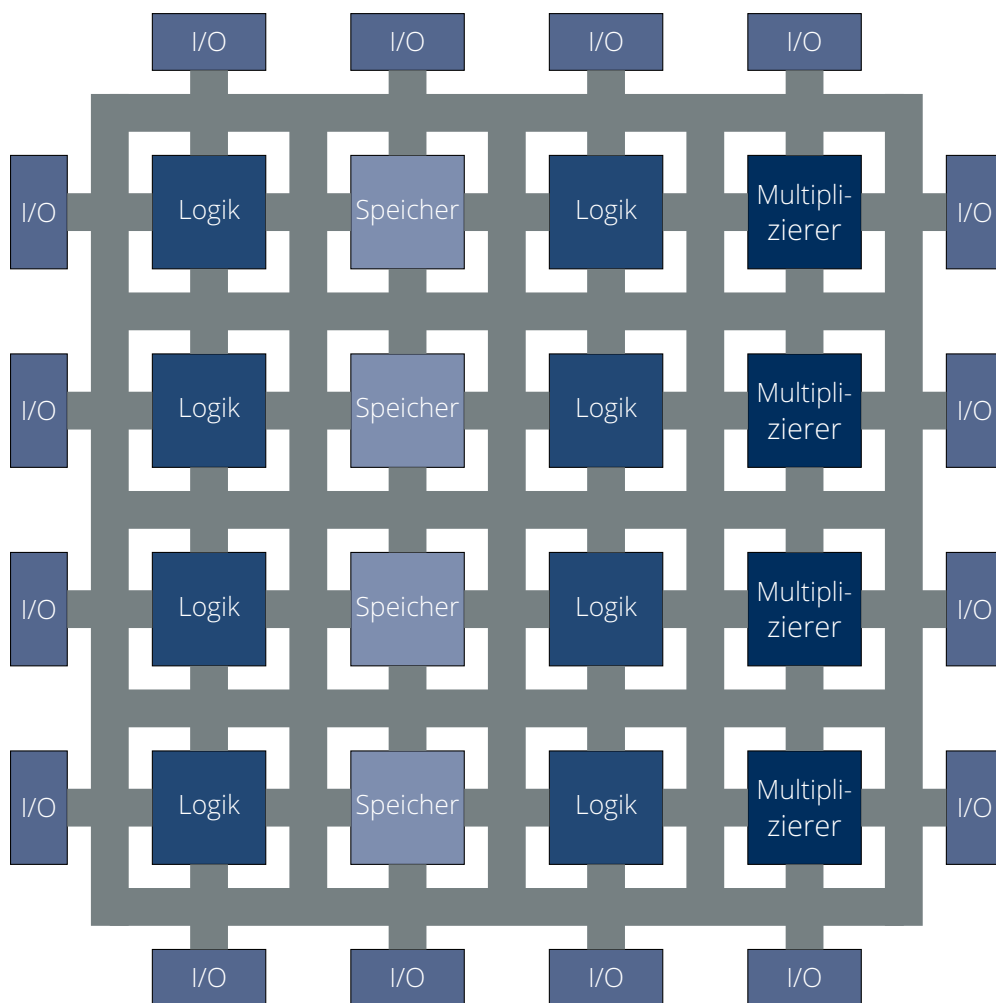


Abbildung 2.1.: abstrakter FPGA-Aufbau [nach HS10, S. 10–14]

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 Flip-Flop 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 arithmetische 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]

Sollen FPGAs als Beschleuniger zum Einsatz kommen, ist die Verwendung weiterer Hardware-Komponenten sinnvoll. Diese befinden sich zwar auf derselben Platine wie der eigentliche FPGA, aber nicht auf demselben Chip. Dies trifft z.B. auf den als DRAM bezeichneten Off-Chip-Speicher zu, 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]. Weitere typische Bestandteile eines Beschleunigers sind z.B. eine Schnittstelle zur CPU und dem Hauptspeicher des Gesamtsystems (etwa über PCI Express) oder Komponenten für die Energieverwaltung.

Ein XCU200-FPGA verteilt die oben genannten programmierbaren On-Chip-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.

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*

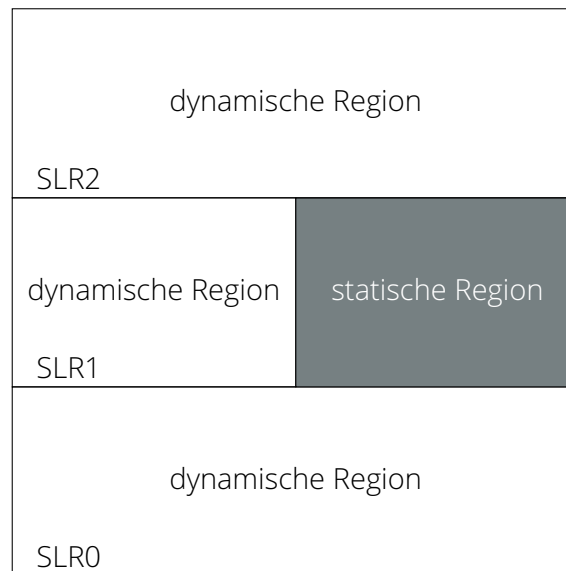


Abbildung 2.2.: Aufbau eines XCU200-FPGAs [nach Xil19a, S. 5]

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]

region bezeichnete Struktur. Zusammengefasst ergibt sich dadurch eine spaltenorientierte Gitterstruktur, wie sie in Abbildung 2.4 zu sehen ist. [vgl. 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 mittlerer Stückzahl 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]. Der Cloud-Anbieter Amazon bietet spezielle FPGA-Instanzen an, in denen Xilinx-FPGAs als Beschleuniger genutzt werden können [vgl. Ama], wovon in der Literatur bereits Gebrauch gemacht wurde [vgl. etwa Di +17].



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

2.2. Entwicklungsprozess

Es sind bei der Software-Entwicklung für FPGAs zwei Vorgehensweisen voneinander abzugrenzen: Einerseits die Entwicklung durch Hardware-Beschreibungssprachen (in dieser Arbeit nur kurz skizziert) und andererseits die High-Level-Synthese, die auf in Hochsprachen implementierten Algorithmen basiert. Die Ansätze unterscheiden sich durch ihre Abstraktion der zugrundeliegenden Hardware (die verschiedenen Abstraktionsebenen wurden von Gajski in seinem Y-Diagramm zusammengefasst, siehe Abbildung 2.5). Die Hardware-Beschreibungssprachen befinden sich mit der Register-Transfer-Ebene auf einem hardware-nahen Level und Detaillierungsgrad, 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 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]

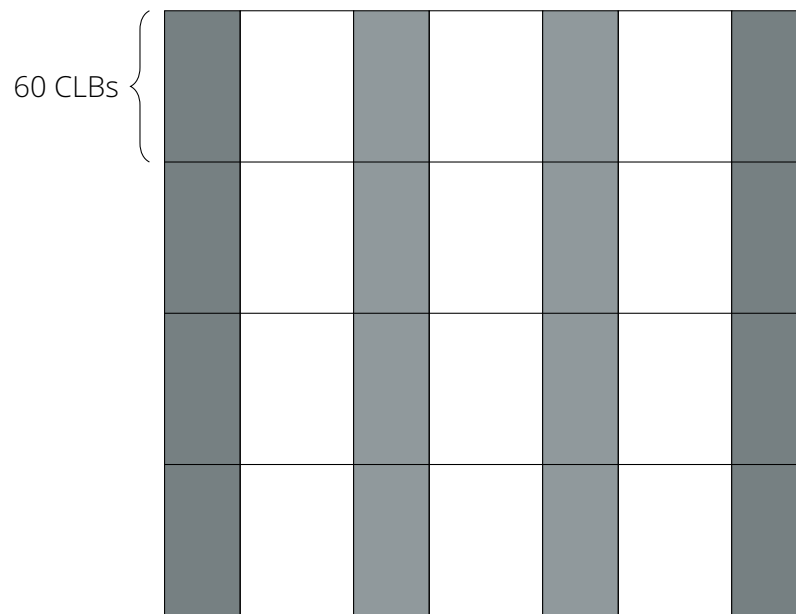


Abbildung 2.4.: Aufteilung der FPGA-Ressourcen auf *clock regions* [nach Xil19e, S. 22]

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 f berechnet durch die Kombination logischer Operatoren (UND, ODER, NICHT, ...) aus der Eingangsvariablen X und der Zustandsvariablen Z des Registers den neuen, im nächsten Takt zu übernehmenden Registerwert $Z+$. Durch die Kombination logischer Operatoren ist f eine Boolesche Funktion¹:

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

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

$$Y = g(Z)$$

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]

¹ auch logische Funktion genannt

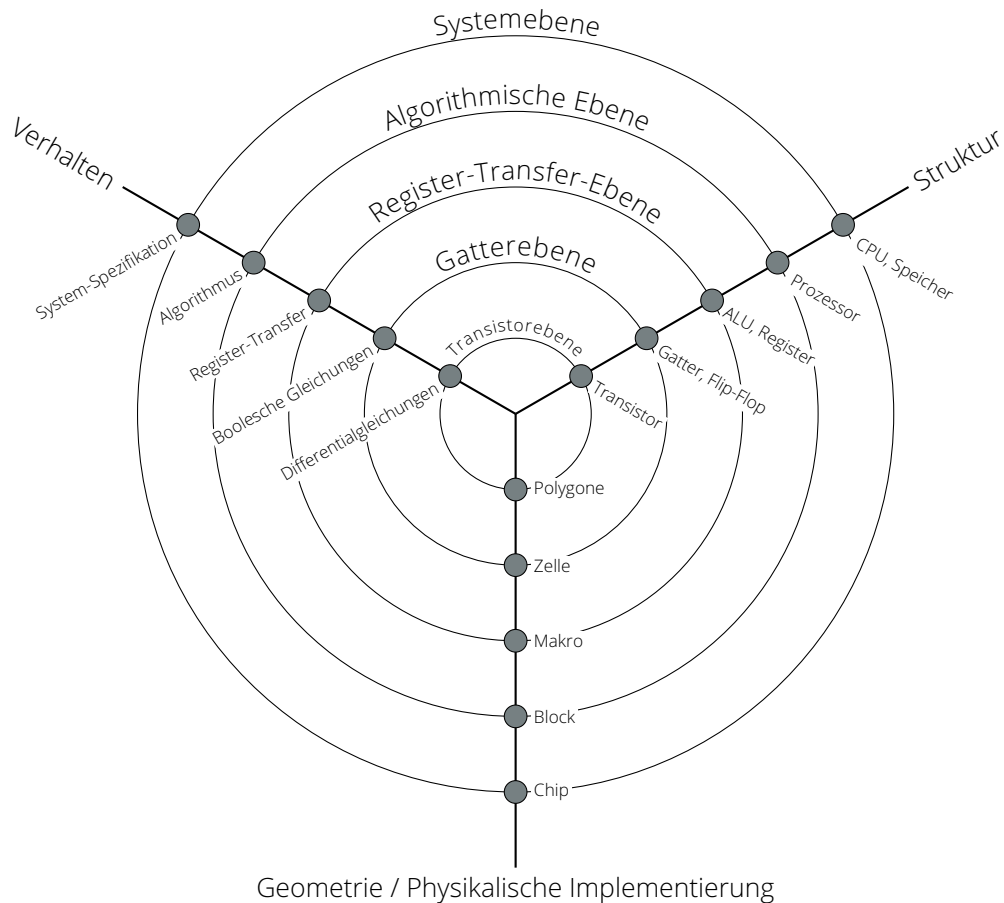


Abbildung 2.5.: Y-Diagramm nach Gajski [nach KB13, S. 10]

```

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 Teilkomponenten. 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]

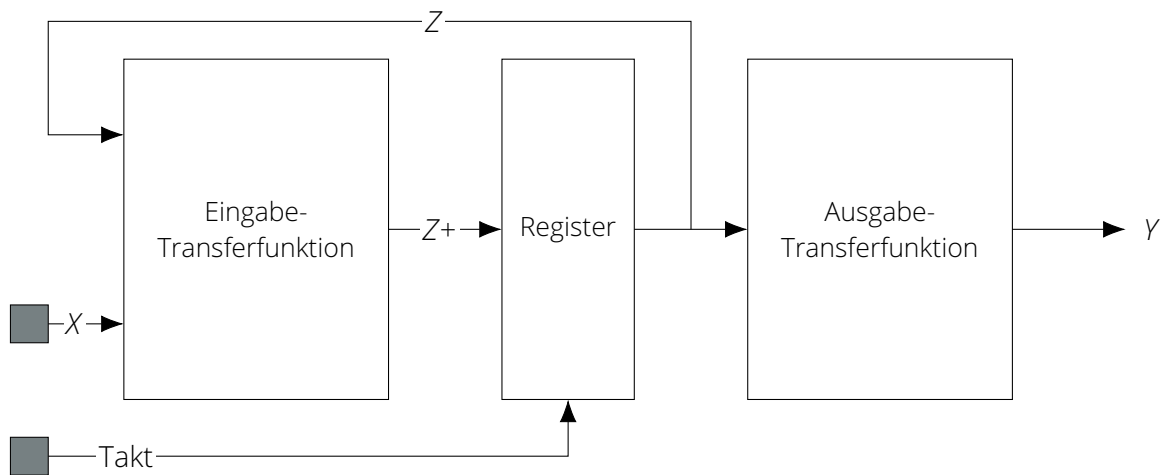


Abbildung 2.6.: Modell eines endlichen Automaten (Moore-Schaltwerk) [nach KB13, S. 35]

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. Dieses 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]

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 ($\text{clk}'\text{event} \text{ AND } \text{clk} = '1'$), werden die Werte von *q0_ns* und *q1_ns* übernommen. Das Verhalten des Registers ist anhand einiger konkreter Werte für die einzelnen Signale in Abbildung 2.7 veranschaulicht. *res* entspricht damit dem *Register* des in Abbildung 2.6 dargestellten Moore-Schaltwerks. [vgl. KB13, S. 30–31]

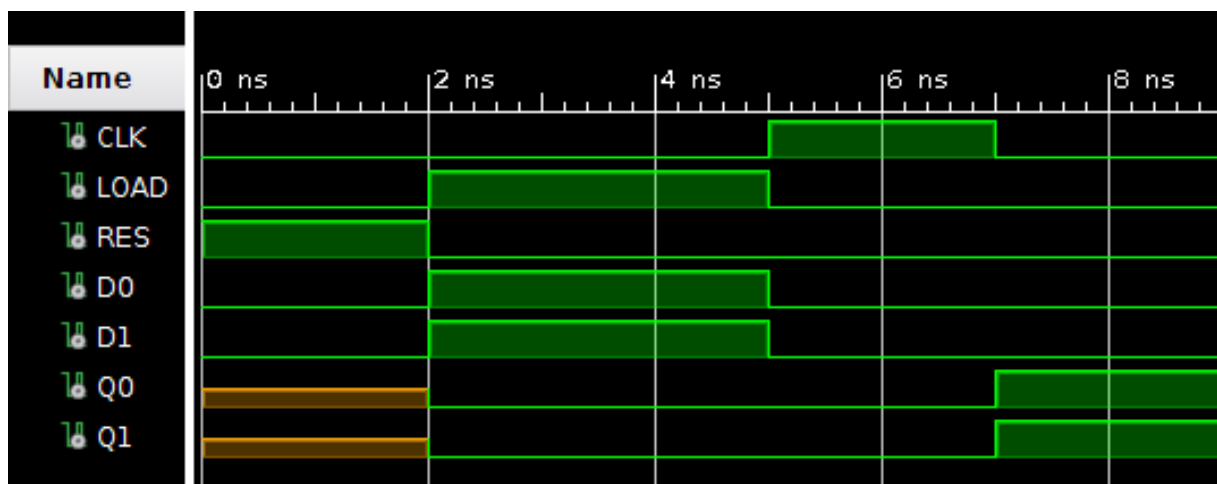


Abbildung 2.7.: Zu Beginn wird das Register über das *res*-Signal zurückgesetzt. Dadurch werden zunächst die internen Signale *q0_s* und *q1_s*, sowie 2 ns später die Ausgangssignale *q0* und *q1* von ihrem undefinierten Zustand auf den Wert „0“ gesetzt. Dann wird *res* auf „0“ gesetzt, während *load* sowie die Eingänge *d0* und *d1* mit „1“ belegt werden. Dadurch werden nach weiteren 3 ns die internen Signale *q0_ns* und *q1_ns* ebenfalls auf „1“ gesetzt. Durch das Anlegen der Taktflanke ($\text{clk} = '1'$) nach insgesamt 5 ns übernehmen *q0_s* und *q1_s* die Werte von *q0_ns* und *q1_ns*, was sich 2 ns später an den Ausgangssignalen zeigt.

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-Flip-Flops 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 Flip-Flops, während die Multiplexer den `mux`-Prozess implementieren. [vgl. KB13, S. 31]

```

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]

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 Flip-Flops zusammensetzen (siehe Quelltext 2.3²).

²Die Verhaltensbeschreibungen der Komponenten `mux2` und `ff2` sind in Anhang A.1 zu finden.

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

```
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]

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 und 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 die Digitaltechnik verlangt. 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 festlegen, die die HLS steuern, wie z.B. den angestrebten Ressourcenbedarf, den gewünschten Datendurchsatz, *Pipelining* oder die Konfiguration des BlockRAMs. 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;
```

wird wie folgt transformiert:

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

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 implementiert³. 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-Prinzip

Das von Prozessoren bekannte Prinzip des *Pipelining*s 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 2.8 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 durch Flip-Flops 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

³Eine LUT entspricht einer Wahrheitstabelle für ein Bit.

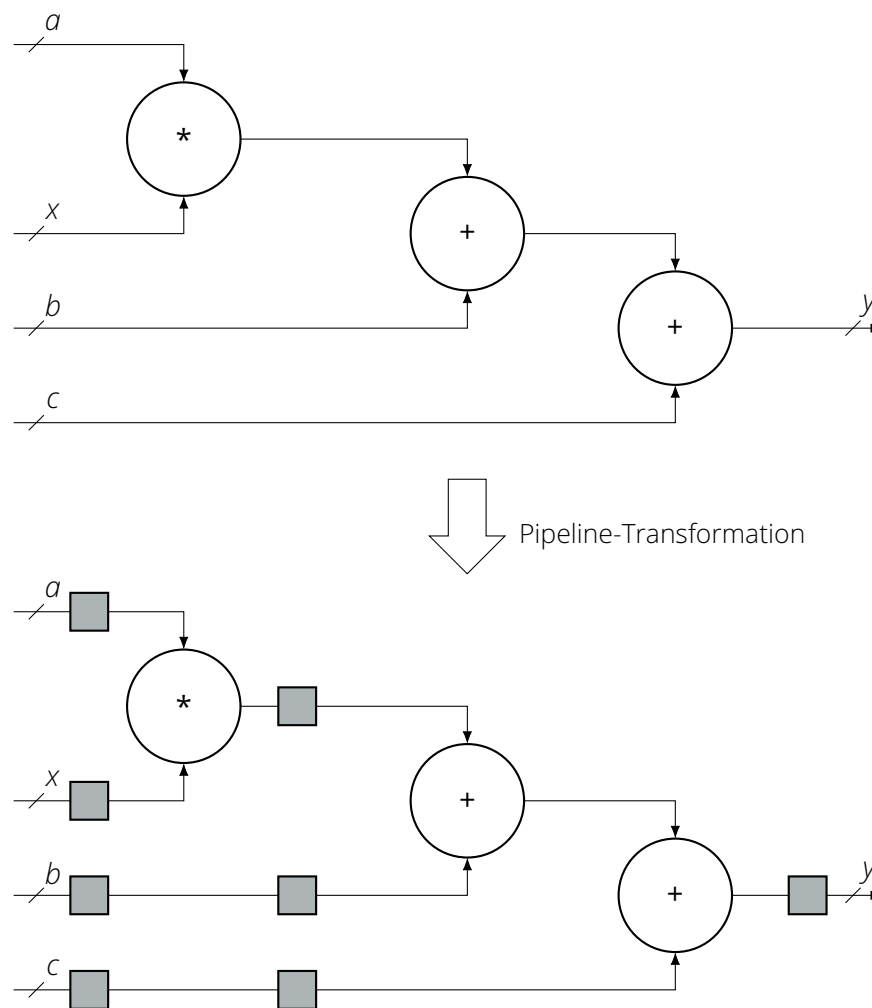


Abbildung 2.8.: FPGA-Implementierung einer mehrstufigen Berechnung [nach Xil19b, S. 21]

in jedem weiteren Taktzyklus einen neuen Wert für y . Dieser Vorgang wird in Abbildung 2.9 illustriert. Je nach verwendetem Algorithmus ist es möglich, dass die Pipeline nicht in jedem Taktzyklus eine neue Schleifeniteration verarbeiten kann (z.B. aufgrund bestehender Abhängigkeiten zwischen einzelnen Iterationen). Die Anzahl der Zyklen, die zwischen dem Beginn zweier aufeinanderfolgender Iterationen liegen, werden als *initiation interval* (II) bezeichnet. Wenn $II = 1$ gilt, beginnt die Pipeline mit jedem Takt eine neue Schleifeniteration. Bei $II = 2$ schafft sie dies nur alle zwei Takte, bei $II = 4$ mit jedem vierten Takt, usw. [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-Prinzip

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-Werkzeuge die Ein- und Ausgangsparameter der im Algorithmus vorhandenen Funktionen. Im einfachsten Fall gibt es keine gemeinsam bearbeiteten Daten, wodurch alle Funktionen parallel

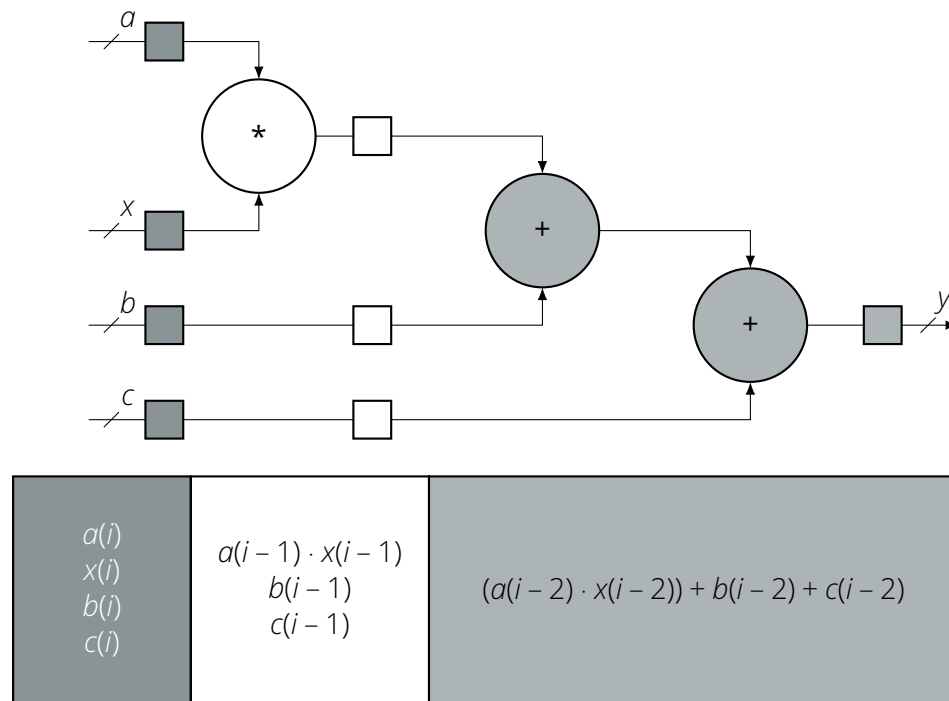


Abbildung 2.9.: FPGA-Pipeline-Architektur [nach Xil19b, S. 22]

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, die ebenfalls als *II* bezeichnet wird, 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

Durch die Xilinx-Implementierung der vor einigen Jahren veröffentlichten SYCL-Spezifikation gibt es eine weitere Möglichkeit, ein Problem auf algorithmischer Ebene zu beschreiben und 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-Standard¹ [vgl. KRH19] verfolgt die herausgebende Khronos-Gruppe das Ziel eines abstrakten C++-Programmiermodells für OpenCL, das die 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, was zu einer besseren Analyse des den *Kernel* umgebenden Kontexts führt. Außerdem ergibt sich der weitere Vorteil, dass *Host* und *Device* Quelltext teilen können, wie z.B. beidseitig verwendete Hilfsfunktionen. Der *Kernel* wird dabei vom *Device*-Compiler extrahiert und in eine

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

Form umgewandelt, 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 Vektorregister. [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-CPU und -GPU ist bereits eine SYCL-Implementierung verfügbar (der Abschnitt 3.3 befasst sich mit allen verfügbaren Implementierungen).

3.1.1. Der AXPY-Algorithmus als Beispiel

Ein im Bereich der parallelen Programmierung häufig verwendetes einführendes Beispiel ist der sogenannte AXPY-Algorithmus, der ursprünglich aus der Bibliothek BLAS stammt [vgl. Law+79]. Dieser führt die Berechnung

$$\vec{y} = a \cdot \vec{x} + \vec{y} \quad (3.1)$$

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

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

3.1.2. Struktur eines SYCL-Programms

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

3.1.3. Beschleunigerwahl und Befehlswarteschlange

Von OpenCL erbt SYCL die Plattformunabhängigkeit. Es wird das Vorhandensein mindestens einer OpenCL-Plattform auf dem System angenommen², 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 *Queue* 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 *Queue* für ein 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 *Queue*

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-*Queue*. In jedem der genannten Schritte stehen dem Programmierer zahlreiche Informationen über die jeweilige Klasse zur Verfügung (Hersteller der Plattform oder des Beschleunigers, Hardware-Eigenschaften, verfügbare Erweiterungen, usw.), die eine Verfeinerung der Auswahl erlauben.

²Diese wird aber nicht vorausgesetzt! Jede SYCL-Implementierung muss auch ohne eine OpenCL-Plattform auf dem Host lauffähig sein.

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 *Queue*-Konstruktion

3.1.4. Speicherreservierung und -initialisierung

Für die Vektoren \vec{x} und \vec{y} der Formel 3.1 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 Programmiers 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 3.2.3).

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

3.1.5. 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 SIMD³-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. Durch `parallel_for` kann auf einem FPGA ebenfalls eine SIMD-Schaltung synthetisiert werden.

³Das ist *single instruction, multiple data* (SIMD), eine von Flynn definierte Klasse innerhalb seiner Taxonomie [vgl. Fly66].

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 mittels `parallel_for` aufgerufen wird (siehe Quelltext 3.6). 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.). Das jeweilige *work-item* wird als Parameter an die Kernel-Funktion übergeben. Es kapselt unter anderem 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

3.1.6. Synchronisierung

Nachdem der Kernel an die *Queue* 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 eingereichten 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

3.1.7. Zusammenfassung

Das gesamte SYCL-AXPY-Beispiel findet sich in Quelltext 3.8, einschließlich einiger unwesentlicher 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();

    // Zugriff auf h_acc_x und h_acc_y ab hier wieder möglich

    return EXIT_SUCCESS;
}

```

Quelltext 3.8.: AXPY – vollständiges SYCL-Beispiel

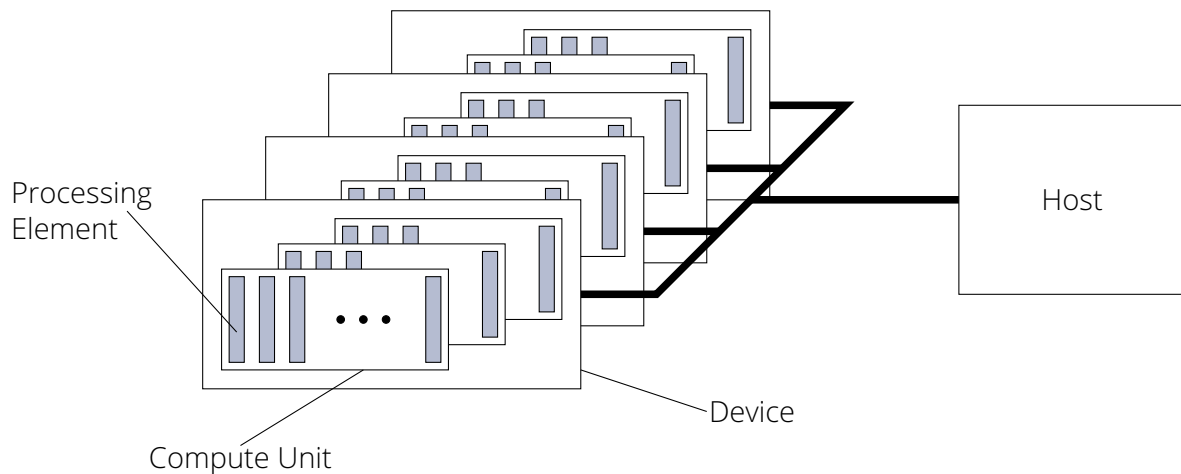


Abbildung 3.1.: SYCLs Plattform-Modell [nach Mun12, S. 23]

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 Kernen. 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 erforderlich, 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 jedoch über Funktionen der gemeinsamen *work-group* synchronisieren. Es ist jedoch nicht mög-

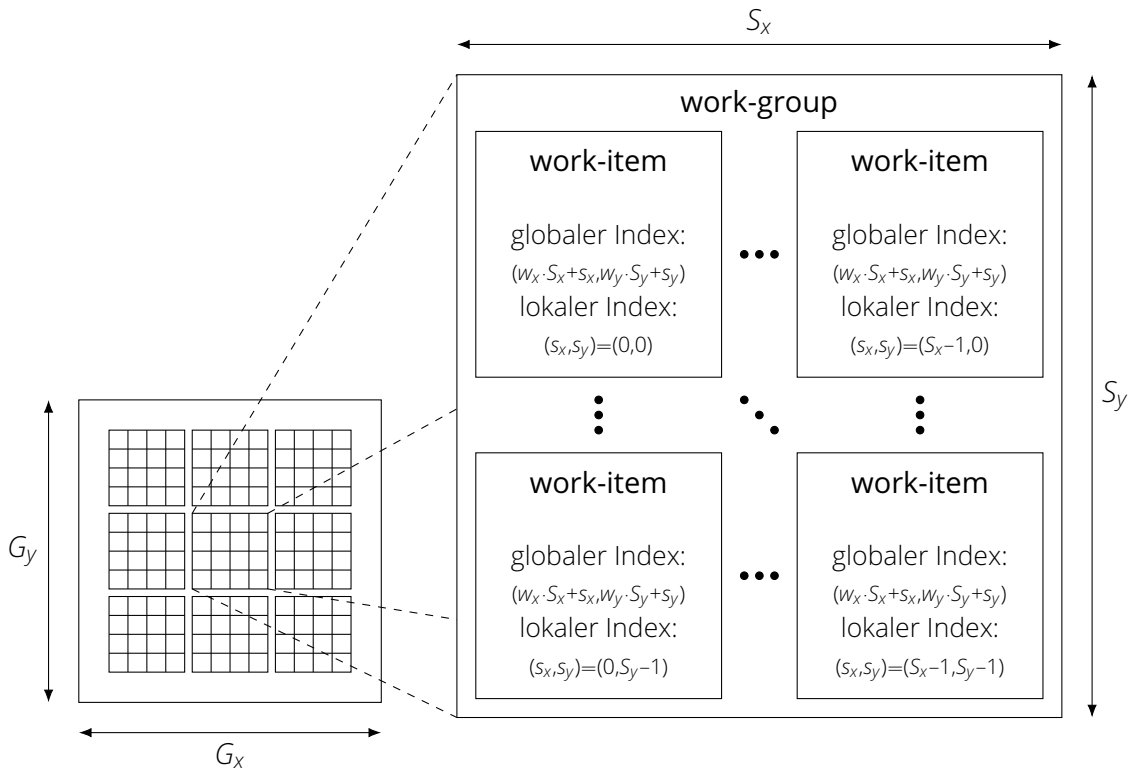


Abbildung 3.2.: SYCLs Indexraum [nach Mun12, S. 25]

lich, *work-items* verschiedener *work-groups* direkt über das SYCL-Interface zu synchronisieren. Diese können nur über den globalen Speicher oder über atomare Funktionen miteinander kommunizieren.

Durch diese Hardware-Abstraktion wird aus Sicht der Programmiers 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.

3.2.2. Kernel-Start

SYCL unterstützt verschiedene Möglichkeiten, den Kernel unter Einbeziehung der im vorigen Abschnitt geschilderten Hierarchie aus *NDRange*, *work-groups* und *work-items* zu starten.

Der Aufruf eines Kernels über die Funktion `parallel_for()` erzeugt eine *NDRange* nach dem oben beschriebenen Muster. Der Programmierer muss immer die Anzahl der *work-items* innerhalb der *NDRange* angeben, während die Angabe der Zahl der *work-items* pro *work-group* optional ist. Sofern der Programmierer darauf verzichtet, die Größe der *work-groups* selbst festzulegen, ist es Aufgabe der SYCL-Implementierung, die Zahl der nötigen *work-groups* zu bestimmen und die Zuordnung der *work-items* durchzuführen. Mit dieser Entscheidung verliert der Programmierer jedoch die Möglichkeit, gruppenweite Operationen durchzuführen, wie z.B. die Synchronisierung innerhalb der *work-group*.

Durch den Aufruf über die Funktion `parallel_for_work_group()` hat der Programmierer die Möglichkeit, seinen Kernel hierarchisch zu gliedern. Der Kernel führt seine Anweisungen zunächst auf der Ebene der *work-group* aus, das heißt, dass jede Anweisung von einem einzigen *work-item* durchgeführt und das Ergebnis der gesamten *work-group* bekannt gemacht wird. Innerhalb des Kernels kann durch die Funktion `parallel_for_work_item()` die Parallelität auf der *work-item*-Ebene ausgedrückt werden: Anweisungen innerhalb dieses Kernel-Teils werden individuell von einer Menge von *work-items* ausgeführt. Diese kann alle *work-items* der *work-group* umfassen oder nur einen Teil der *work-group*. Am Ende des *work-item*-Teils erfolgt eine implizite Synchronisierung der *work-items*.

Sequentielle Kernel lassen sich über die Funktion `single_task()` starten. Die aufgespannte *NDRange* umfasst dann genau eine *work-group* mit genau einem *work-item*. Durch den wiederholten Aufruf von `single_task()` lassen sich komplexe Ketten aufeinander folgender sequentieller Aufgaben einfach modellieren.

3.2.3. Abhängigkeiten zwischen Kernen

Alle *command groups*, die der Programmierer in eine SYCL-Queue einreicht, werden grundsätzlich asynchron ausgeführt, sofern keine Abhängigkeiten zueinander bestehen. Vorhandene Abhängigkeiten werden über die von den Kernen 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 *Queues* nur asynchrone *Queues* 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 3.3 gezeigt. 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

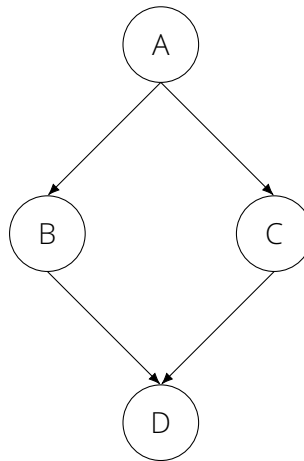


Abbildung 3.3.: Einfacher Aufgabengraph

3.2.4. 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 umwandelt, die dann vom Programmierer weiter verarbeitet werden können (siehe Anhang A.2.1).

3.2.5. 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 Anhang A.2.2).

3.2.6. 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 Kopiervorgang existieren damit zwei voneinander verschiedene Objekte, die getrennte Speicherbe-

reiche 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 konkrete Implementierungen verschiedener Anbieter. 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-Repositorys 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. Die meisten SYCL-Features werden unterstützt, jedoch sind einige SYCL-Klassen noch nicht vollständig implementiert. [vgl. Ker+]

Daneben existiert unter dem Dach des triSYCL-Projekts ein von der Intel-Implementierung abgeleitetes Compiler-Projekt, das sich vornehmlich der besseren Unterstützung von Xilinx-FPGAs anzunehmen scheint. Dieser Compiler wird im Folgenden der Einfachheit halber als Xilinx-Implementierung bezeichnet. [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 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 in der Einleitung 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.]

3.3.6. Zusammenfassung

In der Tabelle 3.1 sind die einzelnen Implementierungen mit ihren unterstützten Hardware-Plattformen und dem jeweiligen Implementierungsstand zusammengefasst.

Implementierung	Hardwareunterstützung	Featurestatus
ComputeCpp	Automotive, Embedded, Intel (CPU, GPU), NVIDIA-GPU (experimentell)	vollständig
Intel	Intel-CPU, Intel-GPU, Intel-FPGA (langfristig)	vollständig
Xilinx	wie Intel, dazu Xilinx-FPGA	vollständig
triSYCL	CPU, SPIR-fähige OpenCL-Hardware	unvollständig
hipSYCL	AMD-GPU, NVIDIA-GPU	unvollständig
sycl-gtx	OpenCL-1.2-fähige Hardware	unvollständig

Tabelle 3.1.: Übersicht der verfügbaren SYCL-Implementierungen

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 `ocl_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 3.10).

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

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

Quelltext 3.10.: 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 `ocl_pipeline_loop`, mit dem Schleifen markiert werden. In SYCL ist sie unter dem Namen `pipeline` verfügbar und wird auf Funktionen angewendet, deren innere Schleifen dieser Optimierung unterzogen werden (siehe Quelltext 3.11).

```

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

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

```

Quelltext 3.11.: 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*. Diese Strategien sind in Abbildung 3.4 grafisch dargestellt. [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 3.12 zeigt die Anwendung dieser Erweiterung.

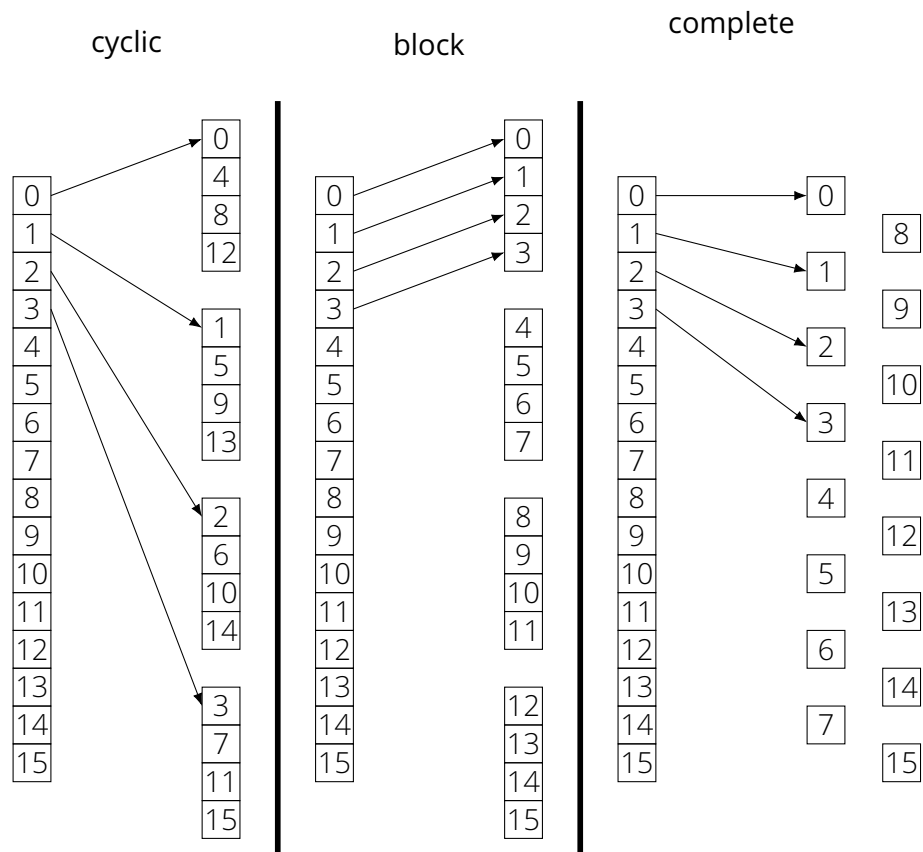


Abbildung 3.4.: Darstellung der Feldpartitionierungsstrategien

```
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 3.12.: 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 bereits die Masterarbeit betreut hatte.

Die Alpaka-Bibliothek definiert eine abstrakte C++-Schnittstelle, mit deren Hilfe parallele Programme geschrieben werden können. Die in Alpaka enthaltenen Abstraktionen ähneln dabei konzeptionell stark dem Aufbau der CUDA-Schnittstelle. Wie im Folgenden gezeigt wird, erstreckt sich diese Ähnlichkeit nicht nur auf das Verhalten der Bibliothek (z.B. bei der Definition von Task-Graphen, siehe Abschnitt 4.2.3), Alpaka übernimmt auch direkt CUDA-Begriffe (z.B. *Grid*, *Block* oder *Thread*, siehe Abschnitt 4.2.1). Im Hintergrund wird Alpaka auf hersteller- oder hardware-spezifische Schnittstellen, wie CUDA oder OpenMP, – im Folgenden als *Backend* bezeichnet – abgebildet. Schematisch ist dies in Abbildung 4.1 dargestellt. 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 sequentiell aus. Während der Kernel-Ausführung existiert also keinerlei Parallelität durch CPU-Threads¹.

¹Der Compiler kann den verwendeten Algorithmus natürlich trotzdem vektorisieren, zudem existieren noch weitere Möglichkeiten der Parallelisierung durch Instruktions-Level-Parallelität oder Pipelining.

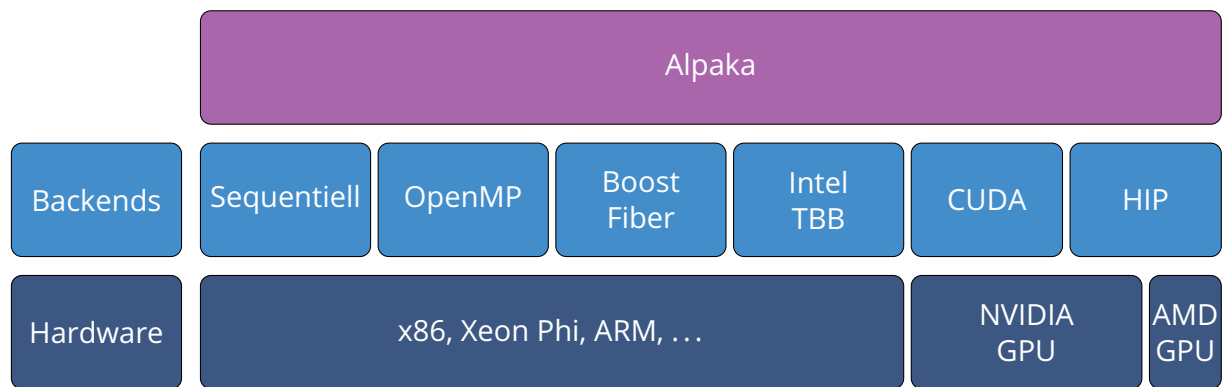


Abbildung 4.1.: Schematischer Aufbau der Alpaka-Bibliothek

4.1.1. Alpaka am Beispiel des AXPY-Algorithmus

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 (siehe Abschnitt 3.1.1 und Formel 3.1). 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 Instanzierung 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 3.1.4 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 reservieren. 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 eingereicht.

```
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 valide Aufteilung vornehmen kann, indem sie die maximal mögliche *Block*-Größe bestimmt. Der Programmierer kann auch 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* 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 AxyKernel
{
    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 gridThreadId = alpaka::idx::getId<
            alpaka::Grid, alpaka::Threads>(acc)[0u];
        auto threadElemExtent = alpaka::workdiv::getWorkDiv<
            alpaka::Thread, alpaka::Elems>(acc)[0u];
        auto threadFirstElemIdx = gridThreadId * 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,
    AxyKernel{},
    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.

```

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

struct AxyKernel
{
    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, AxyKernel{}, 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 Kernen. Für den Entwicklungsprozess mit Alpaka sind außerdem die Methoden für die Fehlerbehandlung und das Profiling relevant.

4.2.1. Hardware-Abstraktion

Alpakas Hardware-Abstraktion 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} der Formel 3.1 durch. Die im Folgenden erläuterten Konzepte wurden stark durch die in CUDA und OpenCL vorhandenen Abstraktionen beeinflusst [vgl. Wor15, S. 17].

Die Konzepte sind hierarchisch und redundant aufgebaut: Sofern eine Ebene der Abstraktionshierarchie von der konkreten Hardware bzw. dem zugehörigen Programmier-Interface nicht unterstützt wird, lässt sich die betreffende Ebene emulieren oder sogar ganz ignorieren. Dadurch ist es möglich, einen Alpaka-Kernel optimal auf die von Alpaka unterstützten Beschleuniger abzubilden. Daher wird Alpakas Hardware-Abstraktion als *Redundant Hierarchical Parallelism* bezeichnet. [vgl. Wor15, S. 22]

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 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 4.2 zeigt eine theoretische Hardware, auf die das *Thread-Grid*-Konzept ideal anwendbar wäre.

Blocks

Eine wie in Abbildung 4.2 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 *Blocks* 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

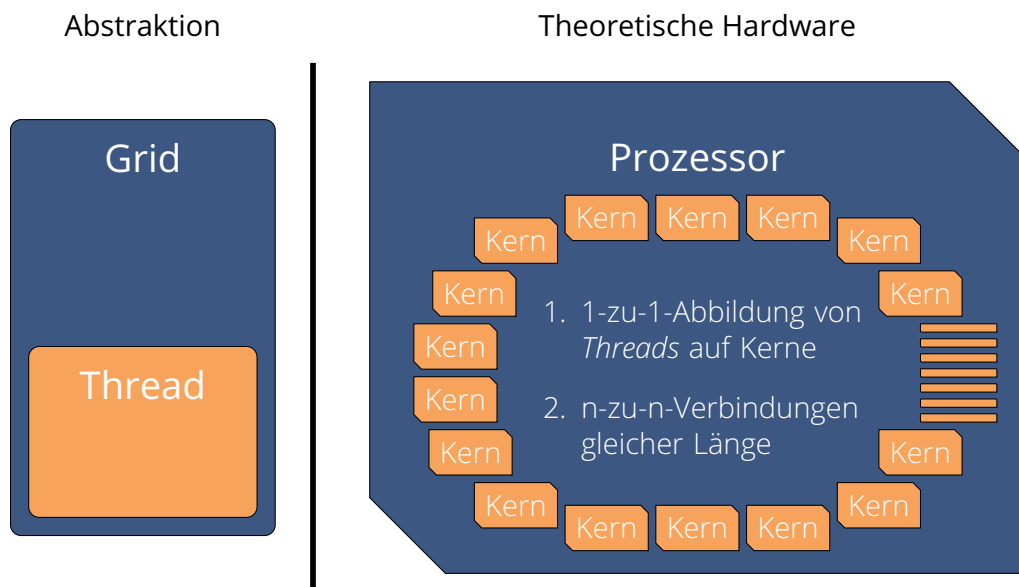


Abbildung 4.2.: Links: Abstraktionshierarchie mit einem aus *Threads* zusammengesetzten *Grid*. Rechts: Ein hypothetischer Prozessor, der datenparallele Anwendungen mit diesem Abstraktionsschema ideal ausführen könnte. [nach Wor15, S. 18]

mehr über das gesamte *Grid*. Eine für dieses Abstraktionsschema ideale theoretische Hardware ist in Abbildung 4.3 dargestellt. Diese erlaubt die *block*-interne Kommunikation, muss jedoch keine Fähigkeiten für die Kommunikation bzw. Synchronisation zwischen *Blocks* besitzen.

Über die *Blocks* lässt sich das CUDA-Prinzip der *transparent scalability* umsetzen. Damit lässt sich dasselbe *Grid*, das aus einer festen Menge von *Blocks* besteht, z.B. auf einer GPU mit zwei Multiprozessoren oder auf einer GPU mit acht Multiprozessoren zur Ausführung bringen: Der Scheduler der GPU verteilt die *Blocks* auf die freien Multiprozessoren, wodurch der Kernel selbst skalierbar wird.

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.

Das *Warp*-Konzept ähnelt prinzipiell dem SIMD-Konzept, unterscheidet sich jedoch stellenweise davon. So können die *Threads* innerhalb eines *Warps* auch verschiedene Pfade einer Verzweigung durchlaufen, was bei einem SIMD-Register einer CPU nicht so einfach möglich ist. Die Abbildung 4.4 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.

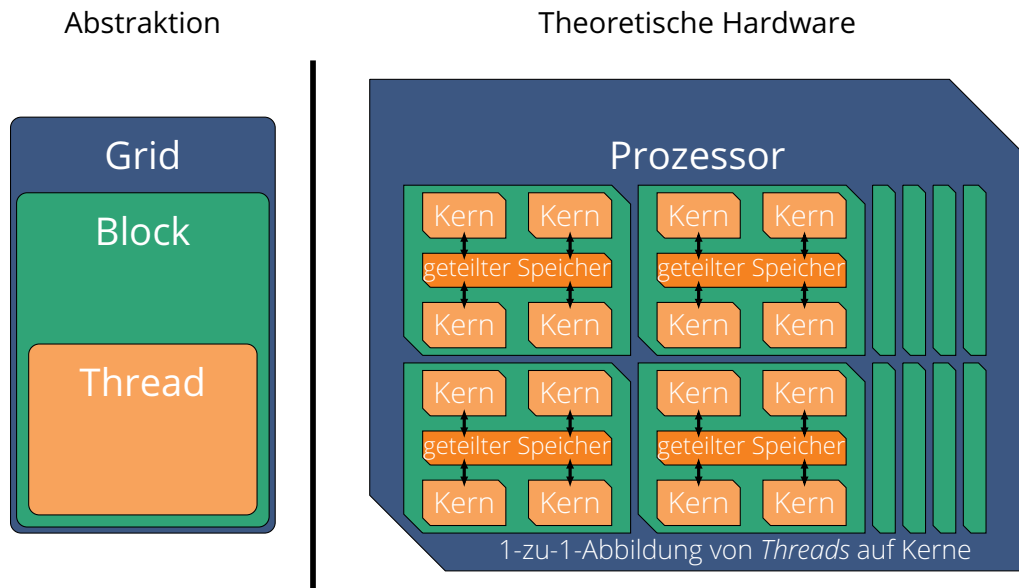


Abbildung 4.3.: Links: Abstraktionshierarchie mit einem *Grid*, das aus zu *Blocks* gruppierten *Threads* besteht. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von *Threads* auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der *Blocks* ermöglicht. [nach Wor15, S. 19]

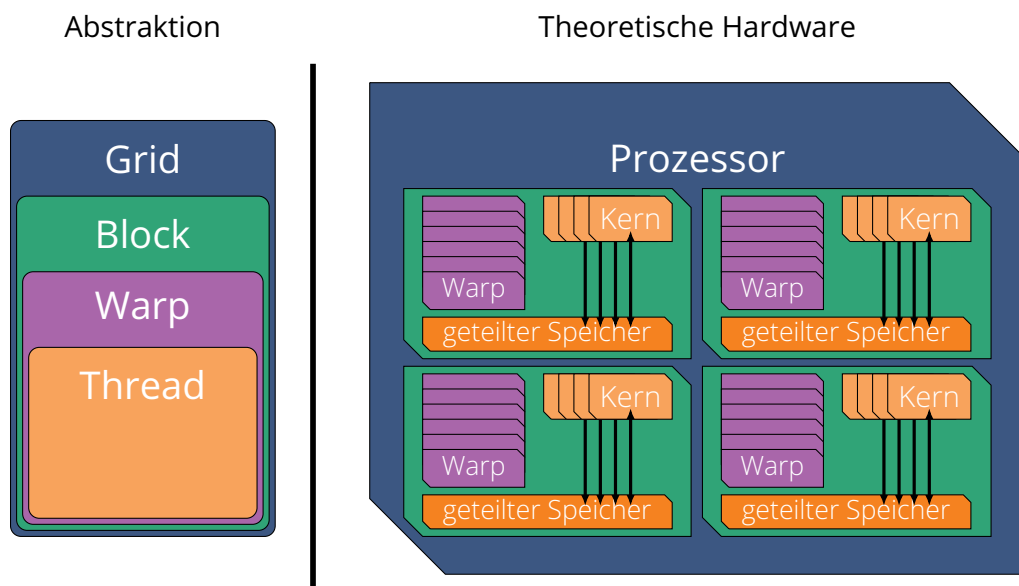


Abbildung 4.4.: Links: Abstraktionshierarchie mit einem *Grid*, das aus zu *Blocks* gruppierten *Warps* besteht. Letzere sind wiederum aus mehreren *Threads* zusammengesetzt. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von *Threads* auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der *Blocks* ermöglicht. Durch die taktgenaue gemeinsame Ausführung der *Threads* in Form von *Warps* kann Chipfläche eingespart werden. [nach Wor15, S. 20]

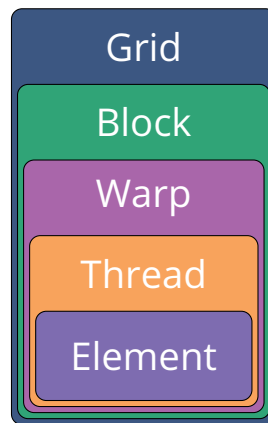


Abbildung 4.5.: Vollständiges Alpaka-Abstraktionskonzept: Mehrere *Elements* werden von einem *Thread* verarbeitet. Mehrere *Threads* werden innerhalb eines *Warps* taktgenau gemeinsam ausgeführt, mehrere *Warps* bilden wiederum einen *Block*. Die Menge aller *Blocks* ergibt das *Grid*. [nach Wor15, S. 22]

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 über alpaka-fremde befehlssatzspezifische Compiler-Intrinsiken explizit selbst tun, gibt dadurch aber die Portabilität auf.

Die *Element*-Ebene ist die unterste Ebene der Alpaka-Abstraktionshierarchie. Dadurch ergibt sich die Gesamthierarchie, wie sie in Abbildung 4.5 dargestellt ist.

4.2.2. Speicherhierarchie

Alpaka übernimmt Teile von CUDAs Speicherhierarchie. Dabei werden aus Beschleunigersicht drei Ebenen innerhalb der Hierarchie unterschieden.

Der globale Speicher (*global memory*) steht allen *Threads* innerhalb des *Grids* zur Verfügung. Im Vergleich zu den restlichen Ebenen der Hierarchie bietet er in der Regel den meisten Speicherplatz, ist allerdings auch deutlich langsamer an den Chip angebunden. Er fungiert außerdem als Bindeglied zum Speicher des *Host*-Systems, da Kopien vom *Host* auf das *Device* über den globalen Speicher durchgeführt werden. Alpakas und SYCLs *global-memory*-Konzepte entsprechen einander.

Auf der nächsten Ebene befindet sich der von den *Threads* eines einzelnen *Blocks* geteilte Speicher (*shared memory*). Jeder *Block* besitzt seinen eigenen *shared memory*, dessen Inhalte verworfen werden, sobald alle *Threads* des *Blocks* ihre Ausführung beendet haben. Gegenüber dem *global memory* zeichnet sich der *shared memory* durch eine höhere Geschwindigkeit bei gleichzeitig deutlich weniger Speicherplatz aus. SYCL kennt ein ähnliches Speicherprinzip, das *local memory* genannt wird.

Jeder *Thread* besitzt außerdem eine Reihe von Registern, auf die nur er selbst Zugriff hat. In diesen werden z.B. *thread*-lokale Variablen gespeichert. SYCLs Äquivalent wird *private memory* genannt.

4.2.3. Abhängigkeiten zwischen Kernen

Im Gegensatz zu SYCL verfügt Alpaka nicht über ein System, um automatisch Abhängigkeiten zwischen Kernen zu verwalten. Stattdessen ist es Aufgabe des Programmierers, Alpaka-*Events* zur Synchronisierung zwischen Kernen 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 fortsetzen.

Der in Abschnitt 3.2.3 eingeführte Beispiel-Aufgabengraph (siehe Abbildung 3.3) erfordert demnach mehr Aufwand als das SYCL-Äquivalent, wie der Quelltext 4.15 zeigt.

4.2.4. 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.

4.2.5. 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. Das ist mit den für NVIDIA-GPUs existierenden SYCL-Implementierungen nicht möglich,

da NVIDIA seit der CUDA-Version 8.0 die Unterstützung für das Profiling OpenCL-basierter Anwendungen eingestellt hat.

Es ist in Alpaka dagegen noch nicht möglich, quelltextseitige Annotationen für Profiler zu verwenden. Diese sind profilerspezifisch und werden von Alpaka noch nicht abstrahiert.

4.2.6. Zusammenfassung

Die Abbildung 4.6 zeigt den konzeptionellen Aufbau der Alpaka-Bibliothek. Die in der Kategorie *Konzept* aufgeführten Bestandteile bilden Schnittstellen nach außen, die von den Backends implementiert werden müssen. In den helleren Farbtönen sind Teilkonzepte dargestellt, die ebenfalls backend-spezifisch sind.

Die Konzepte *Device Manager* und *Device* bilden die *Device*-Verwaltung der Alpaka-Bibliothek. Dabei übernimmt der *Device Manager* die Verwaltung der im System vorhandenen *Devices*, während das *Device* einen konkreten Beschleuniger abstrahiert.

Das *Queue*-Konzept bildet die Verbindung zwischen *Host* und *Device*. In Kombination mit dem *Event*-Konzept wird es auch zur Verwaltung der *Kernel*-Abhängigkeiten verwendet.

Device, *Queue* und *Event* müssen darüber hinaus alle das Teilkonzept *Waitable* erfüllen. Das heißt, dass sie zur *host*-seitigen Synchronisierung geeignet sind.

Das *Buffer*-Konzept abstrahiert den verwendeten Speicher. Über das Teilkonzept *View* und dessen Teilkonzepte lässt sich direkt auf den gekapselten Zeiger zugreifen.

Das *Task*-Konzept ist ein Oberbegriff für *device*-seitige Operationen, die vom *Host* an die *Queue* übergeben werden. Bei einem *Task* handelt es sich um die Verbindung einer Anweisung oder eines Algorithmus (z.B. Kopieren, *memset*, *Kernel*) mit konkreten Daten, auf denen die Anweisung ausgeführt werden soll.

Das Konzept *Accelerator* stellt dem Nutzer eine Reihe *device*-seitiger Funktionen zur Verfügung. Dazu zählen unter anderem Funktionen zur Orientierung innerhalb des *Grid*, mathematische und atomare Funktionen oder Funktionalität für die Generierung von Zufallszahlen.

Das nächste Kapitel wird zeigen, wie diese Konzepte mit SYCL implementiert wurden.

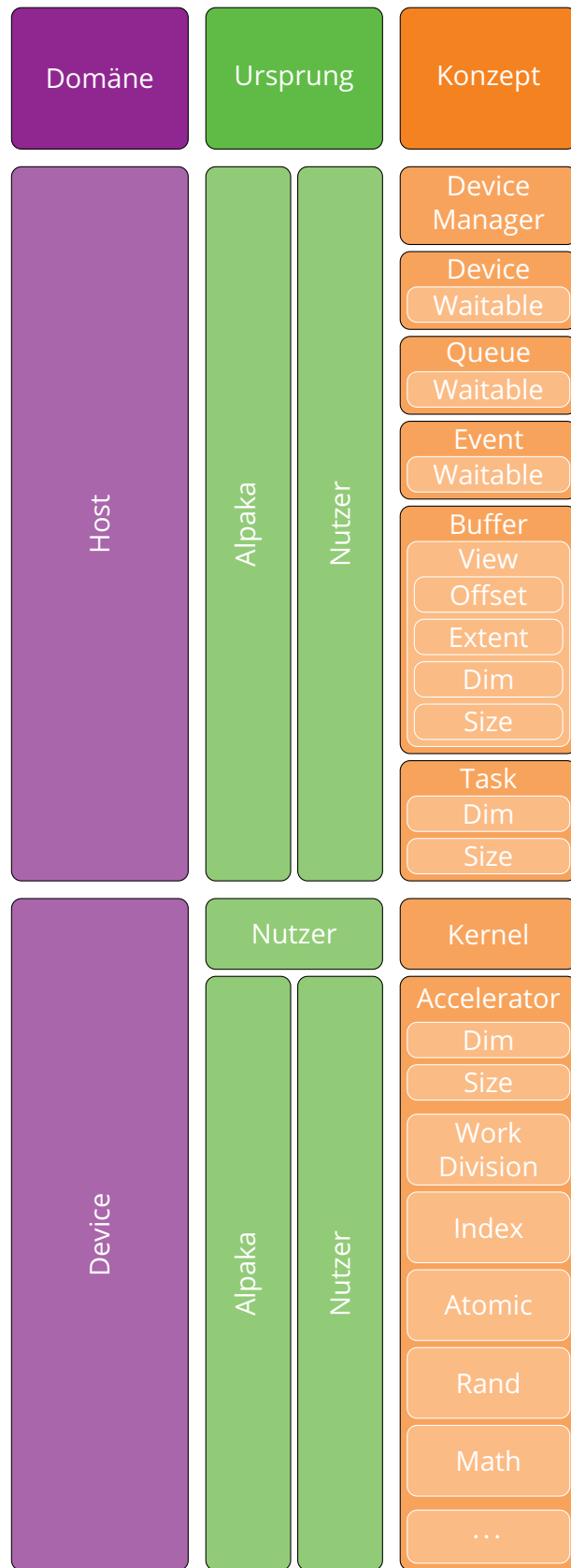


Abbildung 4.6.: Struktureller Aufbau der Alpaka-Bibliothek[nach Wor15, S. 40]

5. Implementierung des SYCL-Backends der Alpaka-Bibliothek

Die Implementierung des Alpaka-SYCL-Backends gestaltete sich in weiten Teilen recht einfach. Stellenweise erforderte sie umfangreiche *Workarounds*, wenn Alpakas Konzepte nicht direkt auf SYCL-Funktionalität abzubilden waren. An anderen Punkten konnte dagegen keine SYCL-Implementierung durchgeführt werden, da Alpaka und SYCL zu widersprüchlich waren.

Die folgenden Abschnitte widmen sich jeweils der Implementierung eines Alpaka-Konzepts und der damit zusammenhängenden Datenstrukturen. Sofern es bei einem Konzept zu den beschriebenen Schwierigkeiten kam, sind diese dort ebenfalls aufgeführt.

Das SYCL-Backend wurde auf Basis des Alpaka-Entwicklungszweigs `develop` entwickelt und hat dessen Neuentwicklungen im Laufe der Arbeit regelmäßig übernommen. Es basiert auf dem Stand vom 23. September 2019 (Commit #82d3294). Alpaka unterstützt mit seinem Interface noch den C++11-Standard. Für das Backend selbst wurde jedoch auf C++17 gesetzt, da dieser Standard von den SYCL-Implementierungen `ComputeCpp`, Intel und Xilinx unterstützt wird und die letzteren beiden C++17 sogar voraussetzen.

5.1. Device-Verwaltung

Am Anfang jedes Alpaka-Programms steht die Auswahl des Beschleunigers. In Alpaka findet diese zur Compile-Zeit statt. Da der Beschleunigertyp bislang durch die Auswahl des Backends impliziert wurde (CUDA → GPU, HIP → GPU, OpenMP → CPU, ...), führte dies nicht zu Problemen.

In SYCL wird diese Wahl erst zur Laufzeit durchgeführt. Der Programmierer gibt zwar einen `device_selector` zur Compile-Zeit an (oder wählt den in Kapitel 3 beschriebenen umfangreicheren Weg zur *Device*-Wahl), dieser wird jedoch erst von der SYCL-Laufzeitumgebung verwendet. Der SYCL-Compiler kennt also noch nicht das zukünftig verwendete *Device*.

Dies führt zu Schwierigkeiten mit Alpaka. Die Annahme, dass das konkrete *Device* erst zur Laufzeit bekannt sein könnte, findet sich nicht in Alpaka. Das auf *type traits* der Template-Meta-Programmierung basierende System zur Auswahl des Beschleunigers stößt hier an seine Grenzen. Wie im Kapitel 4 beschrieben wurde, fungiert eine vom Benutzer gewählte Acc-Datenstruktur als Template-Parameter für viele abstrakte Alpaka-Klassen, die dadurch einen konkreten Code-Pfad wählen können. Die Acc-Datenstruktur selbst wird jedoch nicht auf der *Host*, sondern auf der *Device*-Seite instanziiert und kann deshalb keinen SYCL-`device_selector`

kapseln.

Nun ließe sich einwenden, dass mehrere spezialisierte Acc-Datenstrukturen implementiert werden könnten, die jedes denkbare SYCL-*Device* abbilden. Sieht man von der daraus resultierenden Code-Duplizierung an vielen Stellen im Backend ab, ist damit jedoch noch nicht die Frage der wechselseitigen Beziehungen zwischen Acc und anderen Backend-Klassen gelöst. Ist z.B. nur ein Alpaka-Task gegeben, lässt sich der zugehörige Acc über Alpakas *type traits* ermitteln. Welche der spezialisierten Acc-Klassen soll aber gewählt werden, wenn nur ein Task ohne weitere Informationen gegeben ist?

Die Frage der *Device*-Wahl ließ sich im Rahmen der Arbeit nicht befriedigend lösen. Darüber hinaus ist dies kein SYCL-spezifisches Problem, da neuere OpenMP-Standards ebenfalls mehrere Beschleunigertypen kennen. Hier ist eine Alpaka-weite Lösung erforderlich. Denkbar wäre beispielsweise nicht nur die Angabe der gewünschten Backend-Plattform (z.B. Syc1), sondern auch die zusätzliche Wahl eines konkreten *Devices* zur Compile-Zeit (z.B. XilinxFpga) anstelle der Acc-Datenstruktur. Sowohl das Backend als auch das gewählte *Device* sollten dabei ausschließlich auf der *Host*-Seite instanziiert werden, da es ansonsten wieder zu den Problemen kommt, die momentan bei der Verwendung der Acc-Struktur auftreten.

Die Angabe generischer *Devices* zur Compile-Zeit und die Auswahl eines konkreten *Devices* zur Laufzeit dürfte nicht ausreichen, da es z.B. zwischen FPGAs verschiedener Hersteller durch unterschiedliche Erweiterungen auch zu herstellerabhängigen Code-Pfaden kommen kann.

Der SYCL-Backend-Prototyp erfordert aus den genannten Gründen eine manuelle Auswahl des gewünschten *device_selectors* innerhalb der *Pltf*-Schicht.

Wünscht der Nutzer das SYCL-Backend, steht ihm die Klasse *AccSyc1* zur Verfügung. Diese dient als Parameter für die Klasse *PltfSyc1*, welche zur Laufzeit die *Device*-Auswahl trifft. *PltfSyc1* erzeugt dann ein Objekt der Klasse *DevSyc1*, welches das von der SYCL-Laufzeitumgebung erzeugte SYCL-device kapselt und Alpakas *Device*-Verwaltungsfunktionen auf die äquivalenten SYCL-Funktionen abbildet.

5.1.1. Fehlende *Device*-Informationen

Alpaka bietet dem Nutzer die Möglichkeit, auf Laufzeit-Informationen über das verwendete *Device* zuzugreifen. Darunter fällt auch die Angabe, wie viel globaler Speicher noch unbelegt ist (`dev : : getFreeMemBytes()`). Da es in SYCL keine Möglichkeit gibt, diese Information abzufragen, konnte diese Funktion nicht implementiert werden.

5.1.2. Fehlende *Device*-Funktionen

In CUDA und deshalb auch in Alpaka gibt es die Möglichkeit, ein *Device* in seinen Ursprungszustand zurückzusetzen. Alpakas Funktion `dev : : reset()` unterbricht laufende Kernel und gibt alle Speicherbereiche frei.

In SYCL spielt die *device*-Struktur eine weitaus weniger wichtige Rolle als seine CUDA- und Alpaka-Äquivalente. Ein Äquivalent zu `dev : : reset()` existiert deswegen nicht, und die Dokumentation zum Destruktor der *device*-Klasse enthält keine Angaben darüber, wie sich der Aufruf auf den Zustand des tatsächlichen *Devices* auswirkt [vgl. KRH19, S. 33, 41, 58].

Eine befriedigende Implementierung der `dev : : reset()`-Funktion konnte daher nicht gefunden werden. Das ist problematisch: CUDA-fähige Beschleuniger nutzen diese Funktion, um während des Profilings gepufferte Profiling-Informationen auf der Festplatte zu speichern. Wird das Programm vorher beendet, gehen diese Informationen möglicherweise verloren. Aufgrund des Zustands der zur Verfügung stehenden SYCL-Implementierungen (siehe Abschnitt 6.1) und der Tatsache, dass NVIDIA ohnehin kein Profiling für OpenCL-basierte Programme mehr ermöglicht, kann noch keine Aussage darüber getroffen werden, inwieweit sich das Fehlen der `dev : : reset()`-Funktion für SYCL-Backends zukünftig bemerkbar machen wird.

5.2. Queues und Events

Alpaka-Queues bilden die Schnittstelle zwischen *Host* und *Device*. Sie nehmen vom *Host* Tasks entgegen und bringen diese in der Reihenfolge ihres Eingehens zur Ausführung.

Alpaka unterscheidet den *Host* blockierende und nicht-blockierende Queues. Blockierende Queues unterbrechen die Ausführung auf dem *Host* so lange, bis alle eingereichten Tasks abgearbeitet sind. Die nicht-blockierenden Queues führen diese Tasks asynchron zum *Host* aus.

SYCL kennt ausschließlich nicht-blockierende Queues. Da auf diese aber manuell gewartet werden kann, gestaltete sich die Implementierung der Alpaka-Queues sehr einfach. Die Klassen `QueueSyclBlocking` und `QueueSyclNonBlocking` stellen Alpakas Queue-Funktionalität zur Verfügung. Lediglich die Implementierung des Event-Systems bereitete größere Schwierigkeiten.

5.2.1. Event-Inkompatibilitäten

Alpaka übernimmt viele CUDA-Konzepte, darunter auch das Event-System. Alpaka-Events werden wie Tasks in eine Queue eingereiht. Beendet die Queue einen Task, wird das dahinterliegende Event markiert. Dadurch kann der Nutzer das Ende eines Kernels abwarten und z.B. einen Task in einer weiteren Queue starten, was der Organisation der Abhängigkeiten zwischen Kernen sehr dienlich ist.

Dieses Prinzip ist in SYCL so nicht vorhanden. Eine SYCL-Queue führt grundsätzlich alle Tasks asynchron zueinander aus. Sofern Abhängigkeiten bestehen, werden diese von der Queue erkannt und entsprechend sortiert. Es ist also in SYCL nicht notwendig, die Abhängigkeiten zwischen Kernen selbst zu verwalten.

Dadurch existiert in SYCL auch kein vergleichbares Event-System. Zwar erhält der Programmierer beim Einreihen einer *command group* ein SYCL-Event zurück, auf das der Host (und nur der Host) warten kann. Dieses dient aber vorrangig dem Profiling der Anwendung. Es ist insbesondere für die manuelle Abhängigkeitsverwaltung ungeeignet, da es zwar das Ende eines Kernels anzeigen kann, jedoch nicht die Verfügbarkeit der von diesem Kernel verwendeten Puffer. Über die Puffer-Verfügbarkeit wird in SYCL die Abhängigkeitsverwaltung durchgeführt.

Für die Implementierung der Alpaka-Events stellt dies ein nicht zu lösendes Problem dar. So ist es beispielsweise in SYCL völlig unmöglich (und unnötig), eine Queue über Events auf eine weitere Queue warten zu lassen – genau dieser Mechanismus ist in Alpakas Interface aber vorgesehen. Analog verhält es sich bei Alpaka-*Devices*, die auf Events warten können, während SYCL-*Devices* dazu nicht in der Lage sind.

Im Backend-Prototypen führt der Aufruf der entsprechenden Funktionen dazu, dass nicht nur die jeweilige Queue auf das Event wartet, sondern zusätzlich auch der Host. Dadurch wird die eigentliche Intention der Funktionen – also das in Bezug auf den *Host* asynchrone Warten – zwar nicht erfüllt, aber es findet immerhin eine Synchronisation mit anderen Queues statt. Es besteht dabei aber weiterhin das Problem, dass es sich dabei nicht um eine echte Abhängigkeitsverwaltung im Sinne der SYCL-Spezifikation handelt und deswegen zu Problemen führen kann.

In der Klasse `EventSycl` wurde Alpakas Event-System mit den beschriebenen Einschränkungen implementiert.

5.3. Tasks

Wie im vorherigen Kapitel dargestellt wurde, werden *device*-seitige Operationen in Alpaka unter dem Begriff *Task* zusammengefasst. Dazu gehören neben dem Kernel die Kopier- und `memset`-Operationen. Alpaka-Tasks werden in einer Alpaka-Queue eingereiht und von dieser in der

Reihenfolge ihrer Einreihung ausgeführt. In diesem Abschnitt wird die Implementierung dieses fundamentalen Alpaka-Konzepts beschrieben.

5.3.1. Kernel-Tasks

Ein Alpaka-Kernel-Task (im Folgenden als Kernel-Task bezeichnet) verknüpft die eigentliche Kernel-Funktion mit konkreten Daten. Damit ähnelt ein Kernel-Task stark SYCLs *command groups*, was die Umsetzung dieses Konzepts vereinfacht.

Die Struktur `TaskKernelSycl` enthält die SYCL-Implementierung der Kernel-Tasks. Diese ist aus mehreren Teilen aufgebaut (siehe Quelltext 5.1):

- Die Struktur wird mit den vier Template-Parametern `TDim`, `TIdx`, `TKernelFnObj` und `TArgs` parametrisiert. `TDim` bezeichnet den Datentyp, der später die Dimensionalität des Algorithmus beschreibt, während `TIdx` den Datentyp des bearbeiteten Indexraums angibt. Diese Parameter werden innerhalb des `Work=Division=Konzepts` – gekennzeichnet durch die Elternklasse `workDivMembers` – verwendet, um innerhalb des Kernels die Position des jeweiligen *Blocks* oder *Threads* bestimmen zu können. `TKernelFnObj` ist der Typ bzw. die Signatur der vom Nutzer implementierten Kernel-Funktion. Der variadische Parameter `TArgs` umfasst die Typen der Parameter der Kernel-Funktion.
- Der Konstruktor der Klasse wird während der Instanziierung der Klasse aufgerufen und nimmt konkrete Werte für die gewünschte Arbeitsaufteilung, den auszuführenden Kernel und die zugehörigen Daten an. Die Implementierung des Konstruktors ist im Anhang A.2.3 zu finden.
- Den Variablen `m_kernelFnObj` und `m_args` werden vom Konstruktor die konkrete Kernelfunktion bzw. die zugehörigen Daten zugewiesen.
- Der Operator `()` bildet die Schnittstelle zur intern verwendeten SYCL-Queue. Durch den Parameter wird der Operator als Einstiegspunkt in eine `command queue` gekennzeichnet, wodurch die gesamte Klasse `TaskKernelSycl` ohne weitere Transformationen in eine SYCL-Queue eingereiht werden kann.

```
template <typename TDim, typename TIdx,
          typename TKernelFnObj, typename... TArgs>
class TaskKernelSycl final : public workdiv::workDivMembers<TDim, TIdx>
{
    template <typename TWorkDiv>
    TaskKernelSycl(TWorkDiv&& workDiv,
                  const TKernelFnObj& kernelFnObj,
                  const TArgs&... args);

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

    auto operator()(cl::sycl::handler& cgh);
};
```

Quelltext 5.1.: Aufbau des SYCL-Kernel-Tasks

Vom SYCL- zum Alpaka-Kernel

In der *command group* wird der eigentliche SYCL-Kernel gestartet, der wiederum die vom Nutzer geschriebene Alpaka-Kernel-Funktion mit ihren Parametern aufruft. SYCL stellt drei ver-

schiedene Möglichkeiten für den Kernel-Start zur Verfügung (siehe Abschnitt 3.2.2: `single_task`, `parallel_for` und `parallel_for_work_group`).

`single_task` stellt keine Parallelität durch *work-items* zur Verfügung und ist in seiner Funktion mit einem einzelnen CPU-Thread vergleichbar, der auf genau einem CPU-Kern ausgeführt wird. In Kombination mit den in Kapitel 3 aufgeführten FPGA-Erweiterungen ist dieses Verhalten für FPGAs durchaus sinnvoll, jedoch für CPU- und GPU-Beschleuniger wenig nützlich bis kontraproduktiv. Im Interesse eines portablen Backends scheidet diese Variante aus.

`parallel_for` ist für datenparallele Algorithmen gedacht. Der Programmierer hat die Möglichkeit, die Zahl der *work-groups* und ihrer *work-items* genau zu spezifizieren. Diese lassen sich einfach auf die Alpaka-Äquivalente *Block* und *Thread* Work-Division-Konzept abbilden, weshalb `parallel_for` eine gute Wahl für den SYCL-Kernel-Start darstellt.

`parallel_for_work_group` bietet dieselbe Funktionalität wie `parallel_for`, exponiert diese jedoch mit einem anderen Schwerpunkt. Mit diesem Ansatz wird der Kernel in zwei separate Bereiche aufgeteilt: Einmal auf Ebene der *work-groups* und einmal auf Ebene der *work-items*, wobei die letztere der ersteren untergeordnet ist bzw. aus ihr heraus aufgerufen wird. Befehle auf der *work-group*-Ebene werden genau einmal für die gesamte *work-group* ausgeführt, Befehle auf der Ebene der *work-items* genau einmal pro *work-item*. Dadurch hat der Programmierer die Möglichkeit, seine Algorithmen hierarchisch zu gliedern. Dieses Konzept findet sich nicht in Alpakas Work-Division-Konzept, weshalb in dieser Implementierung `parallel_for` als Kernel-Start-Funktion gewählt wurde.

Für die Nutzung der `parallel_for`-Funktion muss die Gesamtanzahl der *work-items* sowie die Anzahl der *work-items* pro *work-group* angegeben werden.

Da Alpaka diese Informationen in etwas anderem Format (Anzahl der *Blocks* und Anzahl der *Threads* pro *Block*) sowie in eigenen Datenstrukturen enthält, ist eine Konvertierung erforderlich. Die Funktion `get_global_size()` berechnet aus der Zahl der *Blocks* und der *Threads* pro *Block* die Gesamtanzahl der *Threads* (was der Gesamtanzahl der SYCL-*work-items* entspricht) und führt die Konvertierung in ein SYCL-kompatibles Format durch. Die Funktion `get_local_size()` nimmt die Zahl der *Threads* und konvertiert diese in eine SYCL-Datenstruktur, die die Zahl der *work-items* enthält.

```

namespace sycl
{
    namespace detail
    {
        template <typename TName>
        struct kernel {};
    }

    template <typename TDim, typename TIdx,
              typename TKernelFnObj, typename... Args>
    class TaskKernelSycl final : public workdiv::WorkDivMembers<TDim, TIdx>
    {
        auto operator()(cl::sycl::handler& cgh)
        {
            /* Speicherzuweisung */

            // Work Division
            auto blocks = workdiv::WorkDivMembers<TDim, TIdx>::m_gridBlockExtent;
            auto threads = workdiv::WorkDivMembers<TDim, TIdx>::m_blockThreadExtent;
            auto global_size = get_global_size(blocks, threads);
            auto local_size = get_local_size(threads);

            cgh.parallel_for<sy::detail::kernel<TKernelFnObj>>(
                cl::sycl::nd_range<TDim::value>{global_size, local_size},
                [=](cl::sycl::nd_item<TDim::value> work_item)
                {
                    /* Aufruf des Alpaka-Kernels */
                });
        }
    }
}

```

Quelltext 5.2.: Aufruf der Alpaka-Kernel-Funktion im SYCL-Kernel

Wichtig ist ferner der Name des SYCL-Kernels, der allen Kernel-Start-Varianten übergeben werden muss. In SYCL ist der Kernel-Name ein eigener Datentyp, der der Startfunktion als Template-Parameter übergeben wird, sich von anderen SYCL-Kerneln unterscheiden und global sichtbar sein muss. Daher wurde für das SYCL-Backend die Klasse `sy::detail::kernel` eingeführt, welche den Typ der Alpaka-Kernel-Funktion (`TKernelFnObj`) als Template-Parameter entgegennimmt.

Im Quelltext 5.2 wird der Vorgang des Kernel-Starts gezeigt, jedoch noch ohne accessors – diese werden im nächsten Abschnitt behandelt.

Zeiger und *accessors*

Innerhalb der *command group* ist eine Transformation der Datenfeld-Parameter nötig. In Alpaka erhält die Kernel-Funktion pro Datenfeld 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 während der Instanziierung des Kernel-Tasks 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 wird¹.

```

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

Innerhalb eines Kernels lassen sich die accessor-Typen in spezielle SYCL-eigene 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 Programmiers weitestgehend wie ein normaler 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.2.4 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 5.3 dargestellt.

¹Im Rahmen dieser Arbeit wurde der Konflikt in dieser Frage zwischen der Spezifikation und den vorhandenen Implementierungen an das SYCL-Spezifikationskomitee gemeldet. Infolgedessen wurde der Wortlaut der Spezifikation als Fehler anerkannt, der in zukünftigen Fassungen der Spezifikation behoben sein wird (siehe die Diskussion in Anhang B.1.1).

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

Der Quelltext 5.4 zeigt die Nutzung der Template-Meta-Funktionen innerhalb der *command group*: die in einem `std::tuple` gespeicherten Kernel-Parameter werden in ein SYCL-kompatibles Format umgewandelt.

Innerhalb des SYCL-Kernels werden die accessor-Typen (über SYCLs `multi_ptr`) nach einem ähnlichen Prinzip in reine Zeiger transformiert. Die Template-Meta-Funktionen sind dem oben gezeigten Prinzip sehr ähnlich und im Anhang A.2.5 gezeigt, deren Nutzung in Quelltext 5.5.

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

Die extrahierten Zeiger werden dann an den vom SYCL-Kernel aufgerufenen Alpaka-Kernel übergeben, sodass der Programmierer diese verwenden kann. Dadurch lässt sich das SYCL-Backend analog zu den restlichen Implementierungen verwenden.

5.3.2. Kopier- und memset-Tasks

Im Vergleich zu Kernel-Tasks gestaltete sich die Implementierung der Kopier-Tasks in Form der Struktur `TaskCopySycl` vergleichsweise einfach. In ihrem Aufbau sind diese den Kernel-Tasks recht ähnlich, wie Quelltext 5.6 zeigt.

```

template <typename TElem, typename TDim>
struct TaskCopySycl<TElem, TDim, copy_type::host_to_device>
{
    const TElem * const src_ptr;
    cl::sycl::buffer<TElem, TDim::value> dst_buf;
    cl::sycl::range<TDim::value> range;

    auto operator()(cl::sycl::handler& cgh)
    {
        auto dst_acc = dst_buf.template get_access<
                                cl::sycl::access::mode::write>(cgh, range);
        cgh.copy(src_ptr, dst_acc);
    }
};

```

Quelltext 5.6.: Aufbau eines SYCL-Kopier-Tasks

Der Template-Parameter `TElem` definiert den zu kopierenden Datentyp, während `TDim` den Typ der Dimensionalität des zu kopierenden Puffers angibt.

Der `operator()` wird durch seinen Parameter eine SYCL-*command-group*. Der `TaskCopySycl` kann somit von einer SYCL-Queue ausgeführt werden.

Die Struktur enthält ansonsten nur die Operanden der Kopier-Operation und die Anzahl der zu kopierenden Elemente. Aufgrund der Reihenfolge der Operanden sowie der Unterscheidung zwischen Zeigern auf der *Host*-Seite und SYCL-Puffern auf der *Device*-Seite muss die Task-Struktur für die möglichen Kopierrichtungen (*Host-auf-Device*, *Device-auf-Host*, *Device-auf-Device*) spezialisiert werden.

Der `memset`-Task ähnelt in seinem Aufbau dem Kopier-Task. Für ihn entfällt die Angabe der Richtung (und damit die Notwendigkeit der Spezialisierung), da er immer auf dem *Device* ausgeführt wird. Darüber hinaus findet er nur auf Puffer des Typs `uint8_t`, da es sich hier um eine byte-weise Operation handelt. Puffer anderer Typen werden deshalb als `uint8_t`-Puffer interpretiert. Mit dem Befehl `cgh.fill()` lässt sich der Puffer dann Byte für Byte beschreiben.

5.4. Work-Division

Um auf die Positionen der einzelnen *Threads* und *Blocks* innerhalb des aufgespannten *Grids* zugreifen zu können, stellt Alpaka in seinem Work-Division-Konzept entsprechende Funktionen bereit. Da die Kernel durch SYCLs `parallel_for`-Funktion gestartet werden, ist innerhalb des SYCL-Backends die Datenstruktur `nd_item` vorhanden. Diese stellt äquivalente Funktionalität zur Verfügung, so dass alle Work-Division-Funktionen problemlos implementiert werden konnten.

5.5. Geteilter Speicher

Alpaka stellt einen auf *Block*-Ebene geteilten Speicher zur Verfügung, über den die *Threads* des *Blocks* kommunizieren können. Dieser bietet gegenüber dem globalen Speicher sehr viel schnellere Zugriffe, ist aber auch deutlich kleiner.

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.

5.5.1. Dynamischer, geteilter Speicher

Das *type trait* `kernel::traits::BlockSharedMemDynSizeBytes` muss für den dynamischen Fall vom Programmierer auf der Host-Seite für seine Anwendung implementiert werden. Innerhalb des Kernels kann er dann über die Alpaka-Funktion `block::shared::dyn::getMem()` auf den Zeiger zum so reservierten geteilten Speicher zugreifen.

Dieses Verhalten ließ sich so auch für SYCL implementieren. Dafür musste lediglich innerhalb der `TaskKernelSycl`-Struktur ein SYCL-*accessor* für den lokalen Speicher (dem SYCL-Äquivalent zum geteilten Speicher) angelegt werden, auf den die genannte Funktion zugreifen und den zugehörigen Zeiger zurückgeben kann.

5.5.2. Statischer, geteilter Speicher

Für den statischen Fall existiert die Funktion `block::shared::st::allocVar()`, die innerhalb des Kernels aufgerufen wird und Speicherplatz für eine Variable im geteilten Speicher reserviert. Diese Funktion kann für SYCL nicht implementiert werden, da die SYCL-Spezifikation für den Kernel-Start mit `parallel_for` keine äquivalente Funktionalität bietet.

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-Maschinen-code. Der Umfang des statischen geteilten Speichers steht zwar bereits zur Compile-Zeit fest und kann deshalb schon vor dem Aufruf des Kernels reserviert 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 *stack frame* existiert. Als Reaktion auf eine im Rahmen dieser Arbeit gestellte Rückfrage zur Sinnhaftigkeit dieser Einschränkung wird vom SYCL-Spezifikationskomitee zur Zeit untersucht, inwieweit diese Funktionalität trotzdem verfügbar gemacht werden kann (vgl. die GitHub-Diskussion mit Mitgliedern des SYCL-Spezifikationskomitees im Anhang B.1.2).

5.6. Globale Variablen

Die Makros `ALPAKA_STATIC_ACC_MEM_GLOBAL` und `ALPAKA_STATIC_ACC_MEM_CONSTANT` werden von Alpaka zur Verfügung gestellt, um globale Variablen direkt im globalen bzw. konstanten *Device*-Speicher anlegen zu können. Diese dienen vornehmlich der Abstraktion der CUDA-Schlüsselwörter `__device__` und `__constant__`, die dieselbe Funktion erfüllen. Diese Funktionalität wurde für die restlichen Alpaka-Backends bisher nicht benötigt. Im Zusammenhang mit SYCL kommt es hier jedoch zu Problemen.

Zwar ist das Anlegen globaler *device*-seitiger Variablen prinzipiell auch in SYCL möglich, erfordert hier aber den Weg über Puffer und *accessors*. Die C++-Makro-Mächtigkeit reicht allerdings nicht aus, um aus einer Variablendeklaration

```
ALPAKA_STATIC_ACC_MEM_GLOBAL int x;
```

eine Transformation in einen SYCL-Puffer sowie einen zugehörigen *accessor* durchzuführen:

```
cl::sycl::buffer<int, 1>{cl::sycl::range<1>{1}} x_buf;  
// im Kernel  
auto acc = x_buf.get_access</* ... */>(cgh);
```

Daher konnte die Implementierung der obigen Makros für das SYCL-Backend nicht durchgeführt werden.

5.7. Mathematikfunktionen

Bei der Implementierung der von Alpaka vorgegebenen mathematischen Funktionen kam es zu keinen Schwierigkeiten. SYCL stellt für alle Alpaka-Funktionen ein Äquivalent zur Verfügung, das Datentypen mit halber, einfacher und doppelter Präzision entgegennimmt.

5.8. FPGA-Erweiterungen

Wie in Abschnitt 3.4 dargestellt wurde, existieren einige SYCL-Erweiterungen für FPGAs, die eine bessere Nutzung dieser Hardware ermöglichen sollen. Diese sind naturgemäß nicht auf anderen Hardware-Plattformen nutzbar und auch nicht zwischen verschiedenen FPGAs unterschiedlicher Hersteller kompatibel. Es stellt sich daher die Frage, wie diese herstellerabhängigen Erweiterungen innerhalb der Alpaka-Bibliothek nutzbar gemacht werden können. Für den SYCL-Backend-Prototypen wurde auf Erweiterungen verzichtet, um die Hardware-Unabhängigkeit nicht zu gefährden. Dies ist jedoch kein dauerhaft wünschenswerter Zustand, da die Erweiterungen ganz erhebliche Performanzgewinne zur Folge haben können. Denkbar ist ein ähnliches Erweiterungskonzept für Alpaka, in welchem dem Nutzer abstrahierte Formen der SYCL-Erweiterungen zur Verfügung gestellt werden, die er selbst in seinem Kernel nutzen kann. Diese bedarf jedoch einer vorherigen Diskussion innerhalb der Alpaka-Entwicklergemeinschaft.

5.9. Zufallszahlen und Zeit

Da SYCL erst vor wenigen Jahren erstmals veröffentlicht wurde, 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.

5.10. Zusammenfassung

Der strukturelle Aufbau des SYCL-Backends bzw. die in ihm abgebildeten Alpaka-Konzepte sind in der Abbildung 5.1 zusammengefasst. Dabei zeigt blau eine Implementierung aller Features des zugrunde liegenden Konzepts, gelb die Implementierung einer Teilmenge des Konzepts und lila eine nicht erfolgte Implementierung.

Insgesamt umfasst das Backend 5011 Zeilen C++-Quelltext (ohne Kommentare und Leerzeilen).

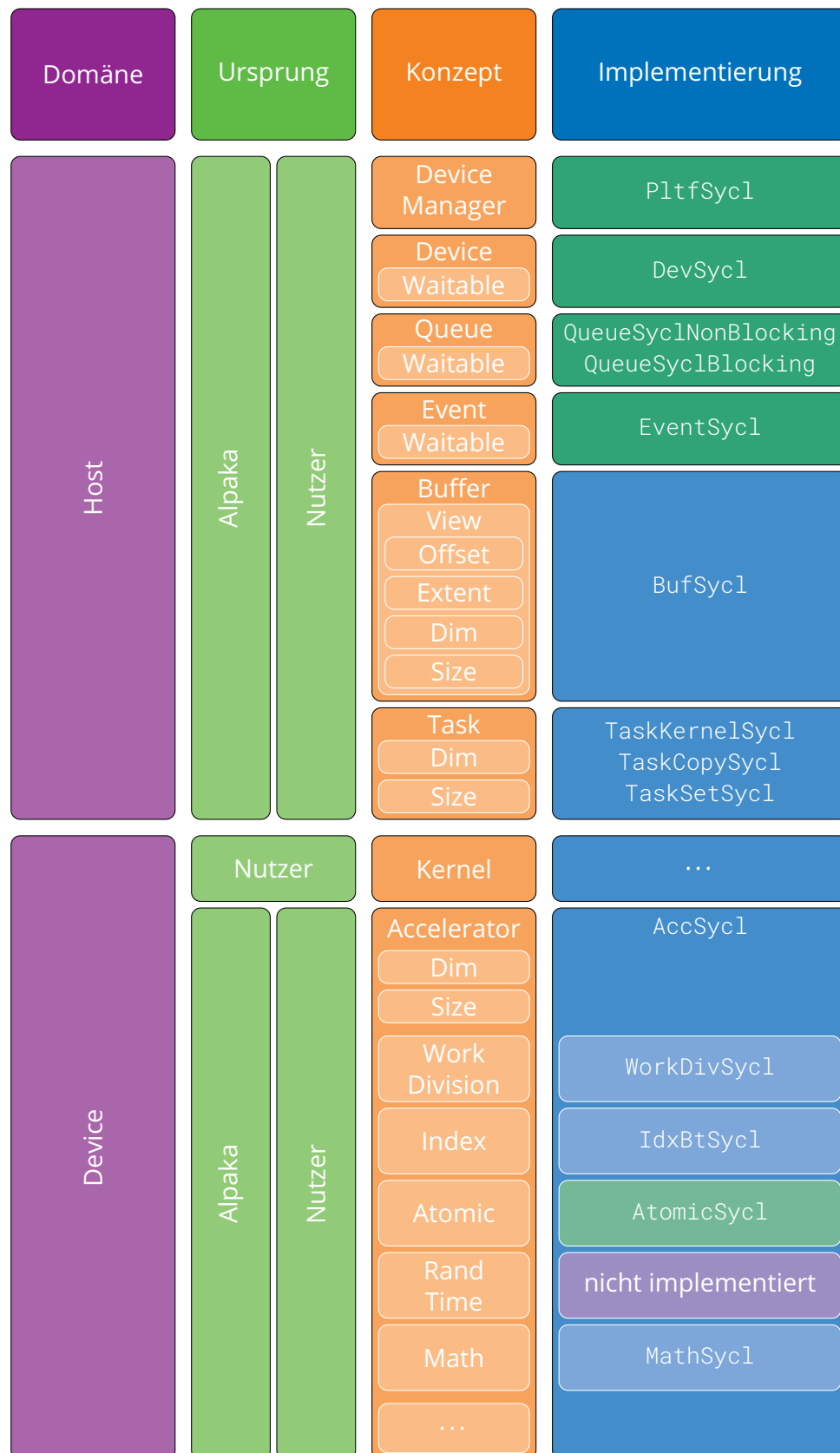


Abbildung 5.1.: Struktureller Aufbau des SYCL-Alpaka-Backends. Blau zeigt eine vollständige Implementierung an, dunkelgrün eine unvollständige Implementierung und lila signalisiert das vollständige Fehlen einer Implementierung.

6. Ergebnisse

6.1. Nutzbarkeit der SYCL-Implementierungen

Im Folgenden wird die Nutzbarkeit von drei öffentlich verfügbaren SYCL-Implementierungen dargelegt. Dabei handelt es sich um ComputeCpp sowie die Implementierungen der Firmen Intel und Xilinx. Die im Kapitel 3 erwähnten Implementierungen hipSYCL und sycl-gtx wurden während dieser Arbeit nicht in Betracht gezogen, da ihnen zu viele kritische Features fehlen und sie daher nicht für eine Nutzung von Alpaka geeignet sind.

6.1.1. ComputeCpp

Die folgenden Ausführungen zu ComputeCpp beziehen sich auf die frei verfügbare *Community Edition* und alle Versionen bis einschließlich Version 1.1.5.

Von den derzeit der Öffentlichkeit zugänglichen SYCL-Implementierungen ist ComputeCpp die einzige, die nicht quelloffen ist. Sie wird vom schottischen Unternehmen Codeplay entwickelt, welches ebenfalls federführend an der Entwicklung des SYCL-Standards beteiligt ist. ComputeCpp unterstützt mehr Hardware-Plattformen als die anderen SYCL-Implementierungen

Im Laufe der Implementierung des SYCL-Backends stellte sich schnell heraus, dass ComputeCpp nicht in der Lage sein würde, das SYCL-Alpaka-Backend zu verwenden. Dafür gibt es zwei Gründe, die nachstehend weiter ausgeführt werden.

Zeiger

Zusätzlich zu der in Kapitel 5 beschriebenen Problematik mit Zeigern kommt es zu weiteren Schwierigkeiten, wenn Zeiger in Verbindung mit ComputeCpp genutzt werden sollen.

ComputeCpp versucht die Information, zu welchem Adressraum ein Zeiger gehört, durch spezielle Zeiger-Attribute nachzuliefern. Ein Zeiger des Typs `int*`, der auf den globalen Adressraum zeigt, wird vom ComputeCpp-Compiler zu `__global int*` transformiert. Das Ergebnis der Transformation wird vom selben Compiler jedoch als ein eigener Typ betrachtet, der nichts mehr `int*` zu tun hat. Dadurch kommt es zu Problemen mit den *type traits* der C++-Standardbibliothek, die einen Zeiger vom Typ `__global int*` nicht mehr als `int`-Zeiger erkennen.

Da es dem Programmierer ebenfalls verboten ist, diese Attribute selbst zu verwenden (ComputeCpp generiert einen Syntax-Fehler), kann eine manuelle Auswahl entsprechender Code-Pfade auch nicht vorgenommen werden.

Darüber hinaus sind diese Zeigertypen nicht mit den SYCL-Klassen kompatibel. So führt die Umwandlung eines entsprechenden Zeigers in einen SYCL-`multi_ptr` dazu, dass letzterer

nicht mehr mit den atomaren Funktionen des SYCL-Standards verwendet werden kann. In diesem Fall meldet ComputeCpp den Fehler, dass die Verwendung von Adressraum-Attributen in Verbindung mit atomaren Funktionen verboten ist.

Fehlerhafte Instruktionen

Auf NVIDIA-GPUs generiert ComputeCpp mitunter Instruktionen, die von NVIDIAs OpenCL- oder CUDA-Umgebung nicht verstanden werden. Dies fällt erst bei der Ausführung des Compilats auf, das entsprechende Fehlermeldungen der NVIDIA-Laufzeitumgebung meldet. Darüber hinaus fehlen ComputeCpp noch einige wichtige Instruktionen, die für ein Funktionieren mit Alpaka nötig sind, darunter viele mathematische Funktionen.

6.1.2. Intel

Eine weitere wichtige Implementierung des SYCL-Standards wird seit Anfang des Jahres 2019 von der Firma Intel herausgegeben. Diese quelloffene Variante ist auf die Nutzung der Intel-OpenCL-Umgebungen für CPUs und GPUs ausgelegt. Es ist aufgrund der parallel zu dieser Arbeit verlaufenen Weiterentwicklung absehbar, dass kurz- bis mittelfristig auch die FPGAs dieses Herstellers unterstützt werden sollen.

Intels Compiler ist die einzige SYCL-Implementierung, die Alpaka-Quelltexte mit aktiviertem SYCL-Backend kompilieren konnte und kam deshalb zur Verifizierung zum Einsatz.

6.1.3. Xilinx

Während der Niederschrift dieser Arbeit wurde der Entwicklungszweig der Xilinx-SYCL-Implementierung verwendet, der dem Hauptzweig einige Wochen voraus und näher an der zugrunde liegenden Intel-Implementierung ist. Die folgenden Ausführungen beziehen sich auf:

- den Commit `#dfb95af` des Zweiges `sycl/unified/next` der Xilinx-Implementierung,
- die Entwicklungsumgebung `SDAccel 2019.1`,
- die OpenCL-Umgebung `XRT 2.2` und
- die Deployment-Plattform `xilinx-u200-xdma` für den Beschleuniger Alveo U200 in der Version `201830.2-2580015` für Ubuntu 18.04 und
- die zugehörigen Entwicklungs-Plattform `xilinx-u200-xdma-dev` in der Version `201830.2-2580015` für Ubuntu 18.04.

Die SYCL-Implementierung der Firma Xilinx hängt eng mit der Entwicklung des Intel-SYCL-Compilers zusammen. Dabei wird in unregelmäßigen Abständen die Code-Basis des Intel-Compilers übernommen und die Xilinx-eigenen Codepfade darin integriert. Das hat zur Folge, dass Fehlerkorrekturen des Intel-Compilers erst mit einiger Verzögerung in die Xilinx-Implementierung Eingang finden.

Darüber hinaus erwies sich die Schnittstelle des SYCL-Compilers zu Xilinx' SDAccel-Plattform, welche die eigentliche Synthese durchführt, sowie zu Xilinx' OpenCL-Treibern im Laufe dieser Arbeit als fehleranfällig oder unvollständig. Aufgrund dieser Probleme war eine Nutzung des Alpaka-SYCL-Backends für Xilinx-FPGAs nicht möglich.

Mathematische Funktionen

Der Compiler generiert aus einigen mathematischen SYCL-Funktionen Instruktionen, die in Xilinx' OpenCL-Treiber nicht vorhanden sind. Dies ist zwar auf ein falsches Benennungsschema innerhalb der OpenCL-Implementierung zurückzuführen, steht einer Nutzbarkeit im Zusammenhang mit Alpaka aber trotzdem im Wege. Andere mathematische Funktionen führen unter ungünstigen Umständen durch ihre Nutzung dazu, dass der Compiler selbst abstürzt.

Strukturen

Zu einem gravierenden Problem kommt es bei der Nutzung von benutzerdefinierten Strukturen. Sofern diese außerhalb eines Kernels definiert und dann innerhalb eines Kernels verwendet werden, kommt es zu einem Absturz des Compilers. Der Grund dafür liegt in einem Fehler des im Hintergrund verwendeten Xilinx-OpenCL-Compilers `xocc`, welcher die Synthese steuert. Nach Aussage der an der SYCL-Implementierung beteiligten Xilinx-Mitarbeiter genießt die Behebung dieses Fehlers niedrige Priorität, weshalb in nächster Zeit nicht mit Besserung zu rechnen ist. Dieser Fehler ist der hauptsächliche Grund, warum das aus vielen Strukturen bestehende Alpaka nicht mit Xilinx' SYCL-Implementierung verwendet werden kann.

6.1.4. Block RAM und Pipelining

In Xilinx' OpenCL-Implementierung wird *local memory* in Block RAM synthetisiert, um den Logikblöcken möglichst schnelle Speicherzugriffe bieten zu können. Xilinx' SYCL-Implementierung verfügt ebenfalls über die notwendigen Klassen und Strukturen, um *local memory* innerhalb eines *Kernels* verwenden zu können. Aufgrund eines Compiler-Fehlers werden diese jedoch nicht als solche erkannt, wodurch Felder im *local memory* tatsächlich im *global memory* angelegt werden.

Alternativ ließe sich BlockRAM über die in Abschnitt 3.4 erwähnten SYCL-Erweiterungen für die Feldpartitionierung verwenden. Diese sind jedoch ebenfalls vom oben genannten Problem mit *kernel*-fremden Strukturen betroffen und führen zu einem Compiler-Absturz.

Zusammengefasst lässt sich Block RAM mit keiner der in SYCL dafür vorgesehenen Funktionalität nutzen. Dies wirkt sich auch auf die Nutzung der Erweiterung für Pipelining aus. Ein häufiges Speicherzugriffsmuster bei FPGAs sind die *burst reads* genannten Speicherzugriffe. In einer Schleife werden aufeinander folgende Daten – z.B. eine Pixelzeile eines Bildes – vom *global memory* in den *local memory* kopiert. Durch die Anwendung des Pipelining-Prinzips auf diese Schleife lässt sich die zur Verfügung stehende Bandbreite besser ausnutzen, als wenn erst im eigentlichen Algorithmus auf die Daten des *global memory* zugegriffen würde.

Da der *local memory* nicht zur Verfügung steht, erfolgen Lese- und Schreibzugriffe ausschließlich auf den *global memory*. Die Zahl der zugehörigen Lese- und Schreib-Ports ist jedoch begrenzt, wodurch die Schleife nicht dem Pipelining-Prinzip unterworfen werden kann.

Kompatibilität mit der SDAccel-Umgebung

Zur Generierung von Profiling-Informationen während der Hardware-Emulation ist es notwendig, die ausführbare Datei mit Debug-Symbolen zu generieren (Compiler-Flag `-g`). Durch einen Compiler-Fehler werden allerdings im *Kernel*-Kompilat inkompatible Debug-Symbole generiert, die von Xilinx' OpenCL-Umgebung nicht verarbeitet werden können. Das macht die Nutzung des mitgelieferten visuellen Profilers `sdx` bzw. die Visualisierung der Profiling-Ergebnisse in Form von Timelines unmöglich.

OpenCL-Treiber

Xilinx' OpenCL-Treiber, auf dem die SYCL-Implementierung aufsetzt, erwies sich im Zusammenspiel mit SYCL als äußerst instabil. Aufgrund seiner internen Struktur ist er nicht in der Lage, einmal reservierten Speicher wieder freizugeben, wenn das reservierende Programm abstürzt. Der Speicher bleibt so lange unzugänglich, bis er neugestartet wird. Da dies nur mit Administrationsrechten funktioniert, ist dies de facto ein Ausschlusskriterium für den Einsatz in Rechenzentren oder Hochleistungs-Clustern.

Sofern der Treiber nicht das oben beschriebene Verhalten zeigt, kann ein fehlerhaftes Programm auch zum Komplettabsturz des Gesamtsystems führen. In diesem Fall ist das System per Fernzugriff nicht mehr erreichbar und muss vom Administrator (oder physisch per Reset-Taste) neugestartet werden. Für den Einsatz in Rechenzentren und vergleichbaren Einrichtungen ist dieses Fehlverhalten denkbar ungeeignet.

6.1.5. Zusammenfassung

Von den beschriebenen SYCL-Implementierungen konnte nur der Intel-Compiler im Zusammenhang mit Alpaka genutzt werden, was als Ziel-Hardware für das SYCL-Backend nur Intel-CPU's und -GPU's zulässt. Insbesondere Xilinx-FPGAs können zum aktuellen Zeitpunkt aufgrund zahlreicher Probleme der SYCL-Implementierung derzeit nicht vom Alpaka-SYCL-Backend verwendet werden.

6.2. Vergleich zwischen Alpaka und SYCL

Im direkten Vergleich erwies sich SYCL gegenüber Alpaka als die modernere, intuitivere und angenehmer zu benutzende Schnittstelle.

Dazu tragen SYCLs Orientierung an modernen C++-Standards (alle untersuchten Implementierungen unterstützen den C++17-Standard, Intel und Xilinx den in Entwicklung befindlichen C++20-Standard) sowie die stilistische Nähe zur C++-Standardbibliothek bei. Dem gegenüber stehen Alpakas Stil-Konventionen, die vom in der C++-Standardbibliothek verwendeten *snake case* (`eine_kleine_funktion()`) zugunsten des *lower camel case* (`eineKleineFunktion()`) abweichen. Darüber hinaus finden sich in Alpaka stilistische Eigentümlichkeiten, die in den meisten C++-Projekten unüblich sind, z.B. die Schreibweise als `char const *` anstelle des weiter verbreiteten `const char *`. Damit steht das C++-Projekt Alpaka auch im Gegensatz zu den vom C++-Standardisierungskomitee veröffentlichten Stilrichtlinien, den *C++ Core Guidelines*.

Die Modellierung von Aufgabengraphen bzw. der Abhängigkeiten zwischen Kernen ist in SYCL deutlich einfacher als in Alpaka. Während dies in SYCL automatisch von der Laufzeitumgebung übernommen wird, muss der Programmierer in Alpaka selbst tätig werden – der Aufwand ist in Alpaka also höher.

Die meisten Konzepte sind in SYCL und Alpaka jedoch recht ähnlich, sodass hinsichtlich der Mächtigkeit keine großen Unterschiede bestehen. Darüber hinaus hat Alpaka gegenüber SYCL den faktischen Vorteil der Hardware-Unterstützung. Während SYCL zur Zeit nur mit Intel-CPU's und -GPU's zufriedenstellend funktioniert (und möglicherweise nicht getesteter Automotive- und Embedded-Hardware), ist Alpaka auf NVIDIA- und AMD-GPU's sowie über OpenMP auf allen CPU's lauffähig. Daher ist Alpaka bereits in einigen produktiven Anwendungen im Einsatz, während sich SYCLs Ökosystem bislang auf die diversen Implementierungen sowie einige von der Firma Codeplay entwickelte Bibliotheken für Mathematik und neuronale Netzwerke beschränkt.

6.3. Verifizierung des SYCL-Alpaka-Backends

Als Integrationstest für das SYCL-Backend diene das in Alpaka geschriebene Programm *jungfrau-photoncounter*, das von der Gruppe *Computergestützte Strahlenphysik* des *Instituts für Strahlenphysik* am *Helmholtz-Zentrum Dresden-Rossendorf* in Zusammenarbeit mit dem schweizerischen *Paul Scherrer Institut*¹ (PSI) entwickelt wird und 2018 in Sebastian Benners Bachelorarbeit beschrieben wurde [vgl. Ben18].

6.3.1. Der *jungfrau-photoncounter*

Der *jungfrau-photoncounter* wertet Daten des Photonendetektor-Typs JUNGFRAU (*adjusting gain detector for the Aramis user station*) aus und zählt die von diesem registrierten Photonen. Der JUNGFRAU-Detektortyp wird für den am PSI befindlichen Freie-Elektronen-Laser *SwissFEL* entwickelt. Ein JUNGFRAU-Detektor verfügt über eine Auflösung von 16 Megapixeln mit 2 Byte pro Pixel und produziert Messergebnisse mit einer Frequenz von derzeit 100 Hz (umgerechnet $3,2 \text{ GB s}^{-1}$). Der Detektortyp befindet sich nach wie vor in Entwicklung und soll langfristig Daten mit einer Frequenz von 2,2 kHz (umgerechnet 74 GB s^{-1}) generieren. Das macht eine dementsprechend schnelle Weiterverarbeitung der Daten durch den *jungfrau-photoncounter* notwendig.

In Rahmen dieser Arbeit wurde nur der Teil des gesamten Funktionsumfangs des *jungfrau-photoncounter* betrachtet, der sich mit dem Zählen der Photonen pro Detektor-Pixel befasst. Der Algorithmus berechnet für jedes Detektorpixel die Formel

$$N_\gamma = \frac{\text{ADC} - \text{Sockel}}{\text{Verstärkung} \cdot E_\gamma} \quad (6.1)$$

Dabei bezeichnet N_γ die Zahl der erkannten Photonen, *ADC* das vom Analogen ins Digitale konvertierte Messergebnis des Pixels, *Sockel* das von Mess- und Umweltbedingungen abhängige Grundrauschen des jeweiligen Pixels (engl. *pedestal*), *Verstärkung* die Signalverstärkung des Pixels (engl. *gain*) und E_γ die Konstante der Photonenenergie.

Um – im Vergleich zur Multiplikation – langsame Divisionen bei der Berechnung zu vermeiden, werden für *Verstärkung* und E_γ vor der Ausführung des Algorithmus die Kehrwerte gebildet. Für E_γ kann dies global erfolgen. Für *Verstärkung* erfolgt die Invertierung mittels eines eigenen *Kernels* (*GainmapInversionKernel*), der den Kehrwert für jedes Pixel bildet.

Der *Sockel*-Wert wird ebenfalls für jedes Pixel bestimmt. Je nach Messbedingungen kann er entweder im Abstand einiger Stunden gemessen (z.B. beim Betrieb des Lasers bei Zimmertemperatur) oder kontinuierlich während der Messung aktualisiert (z.B. beim Betrieb bei Temperaturen unter dem Gefrierpunkt, da hier die Pixel empfindlicher sind) werden. Der zweite Fall wird durch einen weiteren *Kernel* (*CalibrationKernel*) abgedeckt, der anhand der Standardabweichung und des Durchschnitts der zuletzt betrachteten Messergebnisse und *Sockel*-Werte einen neuen *Sockel*-Wert berechnet.

Die Formel 6.1 wird in einem eigenen *Kernel* umgesetzt (*PhotonFinderKernel*), der auf jedes vom Detektor produzierte Messergebnis angewendet wird.

6.3.2. Verifizierung und Performanz

Das SYCL-Alpaka-Backend wurde mit dem Commit #78d9957 des *sycl*-Zweigs des Intel-SYCL-Compilers in Kombination mit der *Intel Graphics Compute Runtime for OpenCL* in der Version 19.46.14807 verifiziert. Dabei wurde eine integrierte GPU des Typs *Intel HD Graphics 520* (auch

¹ Sici! Das PSI schreibt sich ohne Bindestriche.

als *Skylake GT2* bezeichnet) als Hardware-Plattform verwendet, die über 6 GiB Speicher verfügen kann. Diesen muss sie jedoch mit dem *Host*-System teilen. Diese GPU erreicht eine maximale Performanz von 100,8 GFLOPS bei doppelter Präzision². Das *Host*-System selbst verfügt über insgesamt 8 GiB Speicher und ist mit einem Intel-Prozessor des Typs i7-6500U ausgestattet, der eine maximale Taktfrequenz von 3,1 GHz erreichen kann. Als Betriebssystem kam Ubuntu 19.10 zum Einsatz.

Der Quelltext des *jungfrau-photoncounters* wurde dem master-Zweig des Projekts entnommen (Commit #d94f836) und geringfügig an das SYCL-Backend angepasst. Dabei wurden keine Optimierungen der *Kernel* für Intel-Hardware vorgenommen. Von den Entwicklern des *jungfrau-photoncounters* wurde für die Messungen der Datensatz mit der Bezeichnung `px_101016` zur Verfügung gestellt, der 10 000 Messungen mit jeweils 1024×512 Pixeln umfasst. Aufgrund der Speicherlimitierung des Gesamtsystems wurde der Datensatz auf 1000 Messergebnisse reduziert.

Die Abbildung 6.1 zeigt das Ergebnis des oben beschriebenen Algorithmus für den reduzierten Datensatz. Die weißen Pixel zeigen ungültige Messergebnisse an. Diese werden im *jungfrau-photoncounter* durch eine separate Maske eigentlich abgeschaltet. Durch einen Fehler unbekannter Herkunft funktionierte dieser Teil des Programms jedoch nicht mit dem Alpaka-SYCL-Backend. Stattdessen wurden alle Pixel maskiert, was zu einem leeren Ergebnis führte. Aus diesem Grund wurde auf den Einsatz der Maske verzichtet.

Nach mündlicher Aussage der Entwickler des *jungfrau-photoncounter* ist es nicht möglich, die Ergebnisse des im vorigen Abschnitt beschriebenen Algorithmus anhand synthetischer Daten zu verifizieren. Die Verifizierung realer Daten erfolgt deswegen durch die Anwender des *Paul Scherrer Instituts*, die die Ergebnisse auf ihre Plausibilität prüfen. Die Abbildung 6.1 wurde von einem der Entwickler des *jungfrau-photoncounter*, Jonas Schenke, für plausibel befunden.

Das gezeigte Bild wurde anhand von 1000 Messergebnissen berechnet. Die genannte GPU benötigte für die Verarbeitung der Eingangsdaten (ohne das Laden der Daten von der Festplatte, Kalibrierung und Invertierung) 48,372 s. Damit erreicht sie eine Verarbeitungsfrequenz von 20,6731 Messungen pro Sekunde, was unter den derzeitigen 100 Hz und deutlich unter den avisierten 2,2 kHz des JUNGFRAU-Detektors liegt. Da es sich bei der verwendeten GPU nicht um einen Hochleistungsbeschleuniger handelt und ihr tatsächlicher Einsatz im Umfeld eines JUNGFRAU-Detektors nicht zu erwarten ist, reicht dieses Ergebnis als Beweis für die grundsätzliche Funktionstüchtigkeit des Alpaka-SYCL-Backends aus. In einem realen Anwendungsszenario wären Beschleuniger mit höherer Leistung (insbesondere bei doppelter Präzision) zwingend erforderlich, um die angestrebte Frequenz des JUNGFRAU-Detektors erreichen zu können.

6.3.3. Nutzbarkeit von FPGAs

Da der Detektor mit fester Frequenz Daten produziert, würde sich ein FPGA für den beschriebenen Verarbeitungsalgorithmus sehr gut eignen. Durch die in Abschnitt 6.1.3 geschilderten Schwierigkeiten mit der derzeit zur Verfügung stehenden Xilinx-SYCL-Implementierung konnte das bestehende Alpaka-Programm jedoch nicht auf Xilinx-FPGAs zur Ausführung gebracht werden.

6.4. Box-Filter

Da sich die ursprünglich für den Einsatz von FPGAs vorgesehene Alpaka-Anwendung aufgrund des Entwicklungsstatus der Xilinx-SYCL-Implementierung nicht für Messungen eignete, wurde

²Die doppelte Präzision wird innerhalb des Programms benötigt, da einige Zwischenergebnisse in diesem Format gespeichert werden.

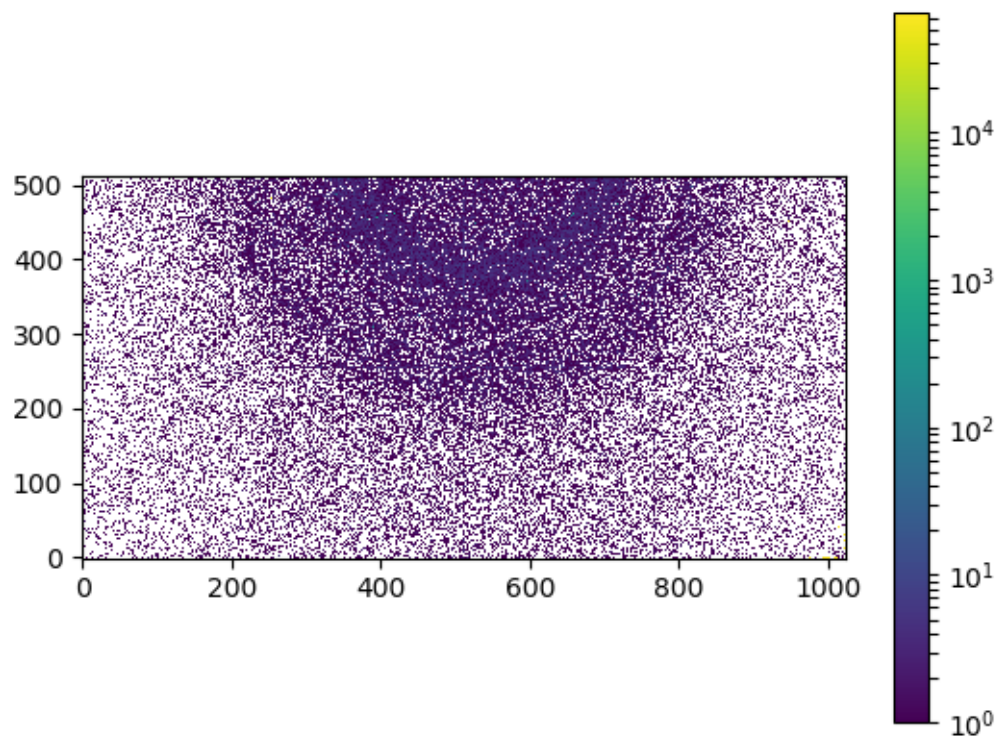


Abbildung 6.1.: Visualisierung eines Ergebnisses des *jungfrau-photoncounters*. Weiße Pixel zeigen ungültige Ergebnisse an, die nicht maskiert werden konnten.

ein Bildverarbeitungsprogramm in SYCL implementiert. Dieses wendet einen einfachen Box-Filter auf eine Menge von Bildern an.

6.4.1. Algorithmus

Die Grundlage des Programms bildet ein einfacher Box-Filter für zweidimensionale Bilder [vgl. NF17]. Dieser berechnet für den Pixelwert p an der Position (x, y) den gefilterten Wert p' an gleicher Stelle. p' ist der Mittelwert des Werts p sowie der direkt an der Position (x, y) horizontal, vertikal und diagonal angrenzenden Pixelwerte:

$$p'(x, y) = \frac{1}{9} \cdot \sum_{j=-1}^1 \sum_{i=-1}^1 p(x + i, y + j) \quad (6.2)$$

Sofern ein Pixel am Bildrand liegt, wird der Wert der Nachbarn, die außerhalb des Bildes lägen, als 0 angenommen.

Der dem Algorithmus entsprechende SYCL-Kernel ist im Anhang A.2.6 zu finden. Obwohl *local memory* und Pipelining zur Zeit von der Xilinx-SYCL-Implementierung nicht unterstützt werden bzw. zu der Nutzung des falschen Speichertyps führen, wurden sie im Kernel-Quelltext zur Illustration der FPGA-SYCL-Programmierung verwendet.

6.4.2. Messergebnisse

Der obige Algorithmus wurde in einer Schleife nacheinander auf 512 Bilder mit jeweils 512×256 Pixeln angewendet. Außerdem wurden separate Messungen für die Pixel-Datentypen `int` und `float` vorgenommen, um die durch einen anderen Datentyp verursachten Veränderungen des Ressourcenverbrauchs beobachten zu können.

Der SYCL-Quelltext des Programms wurde mit dem SYCL-Compiler der Xilinx-Implementierung (Entwicklungszweig `sycl/unified/next`, Commit `#dfb95af`) übersetzt. Die darauf folgende High-Level-Synthese wurde von den Werkzeugen der Entwicklungsumgebung `SDAccel` in der Version 2019.1 durchgeführt.

Die Ausführung der synthetisierten Schaltungen erfolgte auf dem FPGA-Knoten `h002` des am Helmholtz-Zentrum Dresden-Rossendorf befindlichen HPC-Systems *Hemera*. Dieses verfügt über zwei Xilinx-FPGA-Beschleuniger des Typs *Alveo U200*, wovon einer für die Messungen verwendet wurde. Softwareseitig kamen auf diesem Knoten die Module `gcc/9.1.0` und `xilinx/2.2` zum Einsatz. Die SYCL-Implementierung steht nicht als Modul zur Verfügung und musste zunächst in der oben genannten Version lokal im Benutzerverzeichnis auf dem Knoten kompiliert werden.

Compile-Zeiten

Schon während der Synthese der Schaltungen ergaben sich durch den Austausch der Datentypen signifikante zeitliche Unterschiede bei den einzelnen Syntheseschritten (siehe Abbildung 6.2). Während beide Datentypen während der Umwandlung des SYCL-Kernels in ein HLS-taugliches Format noch ungefähr die gleiche Zeit benötigen, benötigt die `int`-Schaltung in allen folgenden Schritten deutlich messbar mehr Zeit als das `float`-Äquivalent. Anscheinend fällt es den Synthesewerkzeugen leichter, optimierte Schaltungen für `float`-Operationen zu generieren.

Ressourcenverbrauch

Beim Blick auf die Auslastung der zur Verfügung stehenden Ressourcen (siehe Abbildung 6.3) fallen zwei Dinge auf:

Zum einen benötigt die `int`-Schaltung nicht nur mehr Zeit, sondern auch mehr Ressourcen als ihr `float`-Gegenstück. Die CLBs und ihre Komponenten werden wesentlich stärker in Anspruch genommen als bei der `float`-Schaltung. Besonders deutlich wird der Unterschied bei der Betrachtung der Netzlisten (siehe Abbildung 6.4). Es ist klar zu sehen, dass die `int`-Schaltung weitaus mehr Fläche auf dem Chip belegt als ihr `float`-Gegenstück. Eine weitere Erkenntnis ist, dass die Synthese zunächst den inneren Bereich des FPGAs zu füllen scheint und erst dann auf die weiter außen liegenden Bereiche ausweicht.

Zum anderen ist das im vorigen Abschnitt beschriebene fehlerhafte Verhalten bezüglich SYCLs *local memory* gut sichtbar. Beide Schaltungen verwenden keinen *UltraRAM* und nur einen sehr kleinen Anteil des zur Verfügung stehenden *Block RAM*.

Laufzeiten

Über die in SYCL vorhandenen Profiling-Funktionen lassen sich Erkenntnisse über die Laufzeit gewinnen. Unglücklicherweise war nur die `float`-Schaltung auf dem FPGA-Knoten lauffähig. Diese benötigte für die Filterung von 512 Bildern etwa 22,26 s, oder ca. 45 ms pro Bild. Das entspricht einer Verarbeitungsfrequenz von rund 23 Bildern pro Sekunde, womit ungefähr die Geschwindigkeit der Intel-GPU bei der Ausführung des *jungfrau-photoncounter* im vorigen Abschnitt erreicht ist. Für einen Beschleuniger, der für den Einsatz in Rechenzentren ausgelegt wurde, ist das deutlich zu wenig. Es ist allerdings zu bedenken, dass die Synthesewerkzeuge aufgrund des Entwicklungsstandes der Xilinx-SYCL-Implementierung keinerlei Optimierungen – wie z.B. die Nutzung des Block RAM oder Pipelining – vornehmen konnte, obwohl diese im Kernel-Quelltext selbst eingefügt wurden. Mit zunehmendem Reifegrad der Xilinx-SYCL-Implementierung dürfte derselbe Quelltext zukünftig deutlich bessere Ergebnisse liefern.

Die `int`-Schaltung führte dagegen – trotz vorheriger Verifizierung durch Software- und Hardware-Emulation – entweder zu einem Absturz des gesamten Knotens oder wurde vom HPC-System nach einer Stunde(!) beendet, ohne ein Ergebnis zu produzieren. Daher ist ein Vergleich der Laufzeiten zwischen den Schaltungen für verschiedene Datentypen zum aktuellen Zeitpunkt nicht möglich.

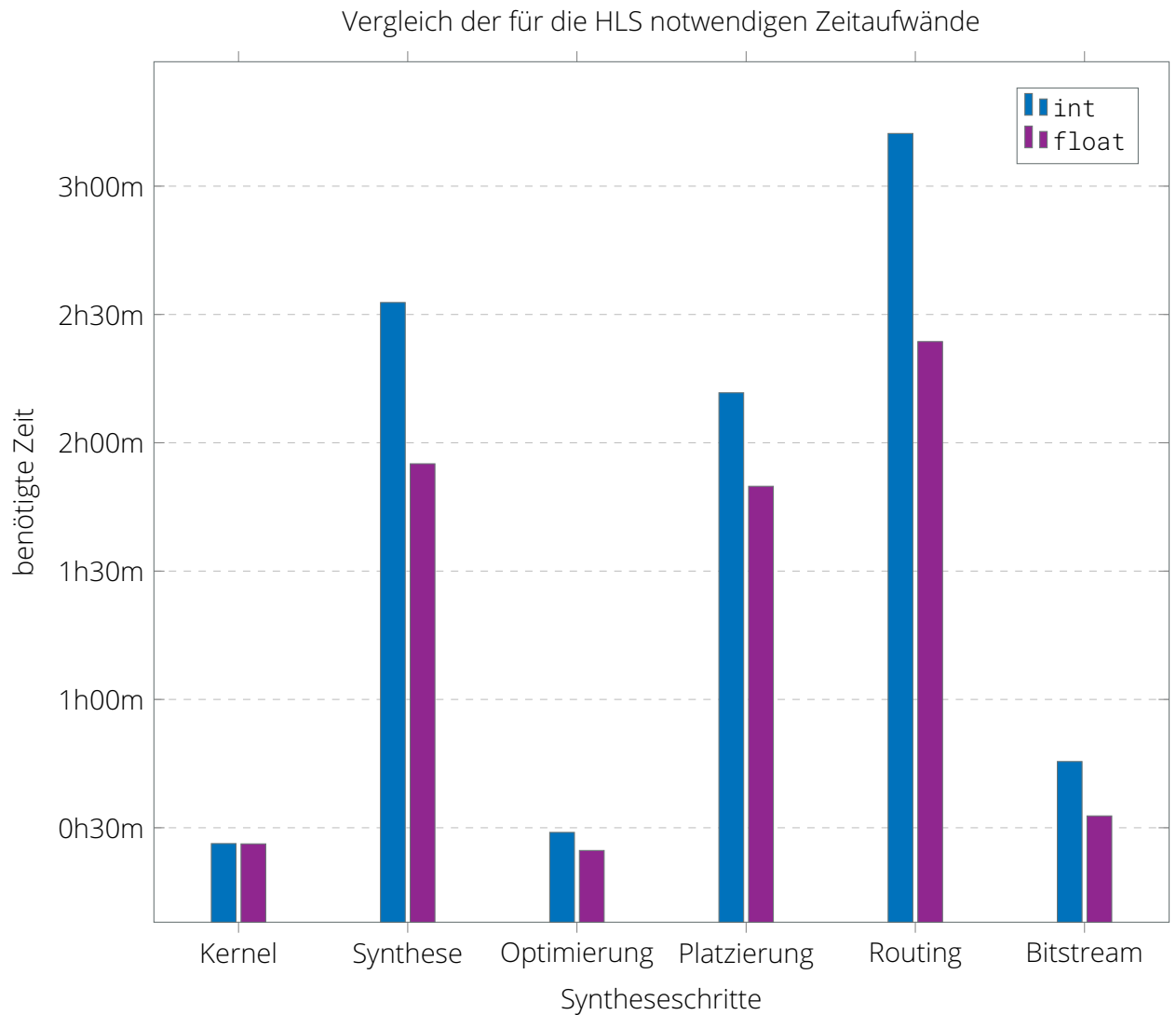


Abbildung 6.2.: Vergleich der für die HLS notwendigen Zeitaufwände bei verschiedenen Datentypen. Die Syntheseschritte werden während der HLS von links nach rechts ausgeführt. Der Schritt *Kernel* bezeichnet die Umwandlung des SYCL-Kernels in ein für die HLS geeignetes Format. Der Schritt *Synthese* meint die Block-Level-Synthese, *Optimierung* die Logikoptimierung, *Platzierung* die Logikplatzierung, *Routing* das Routing und *Bitstream* die Bitstream-Generierung.

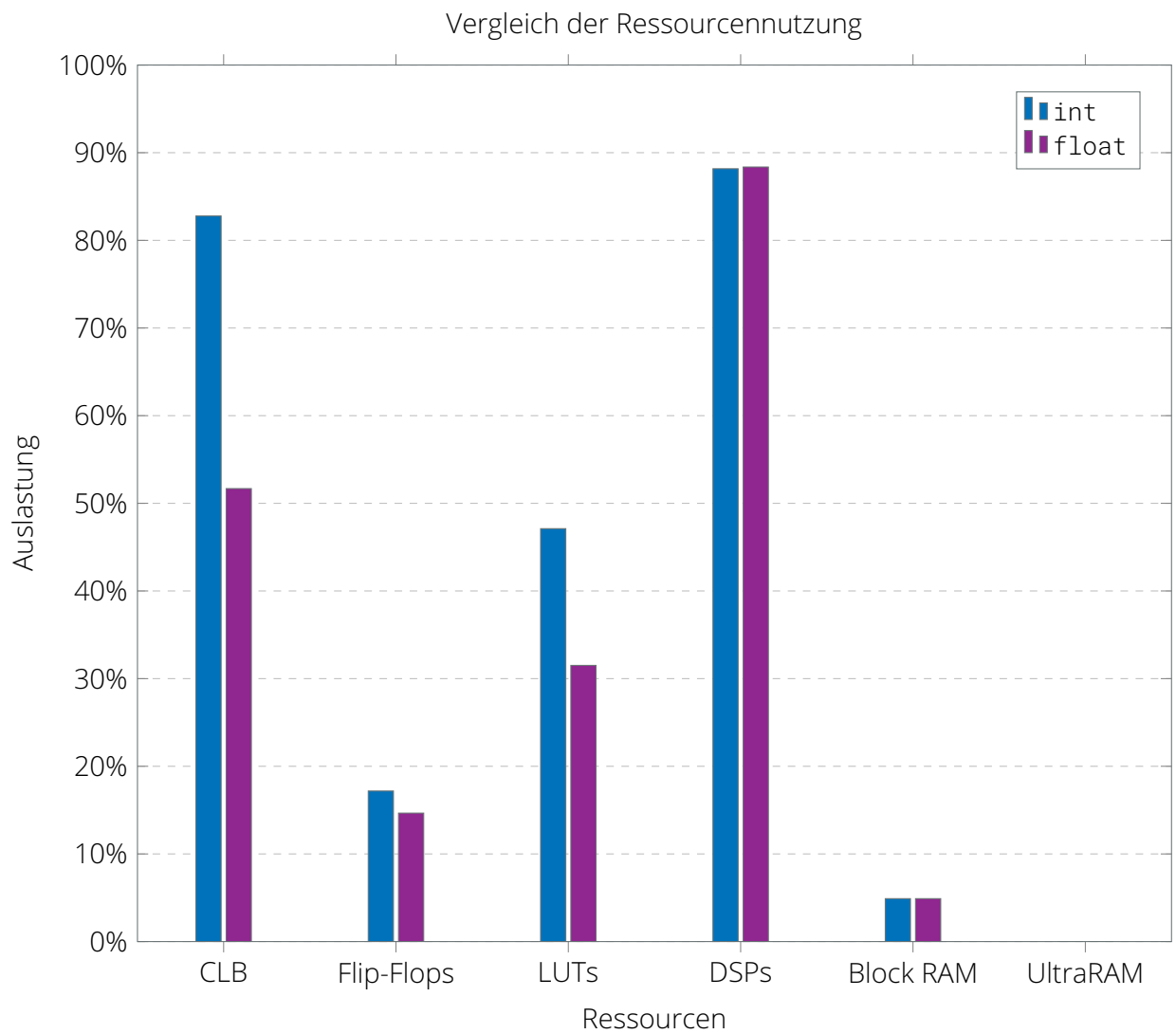
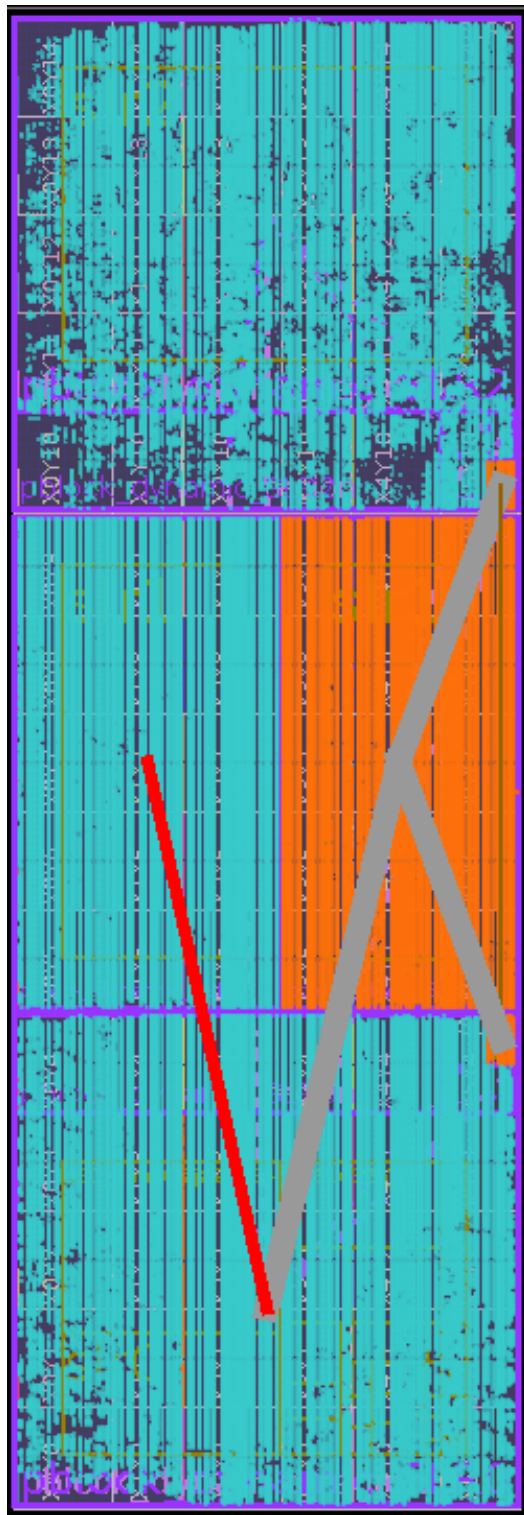
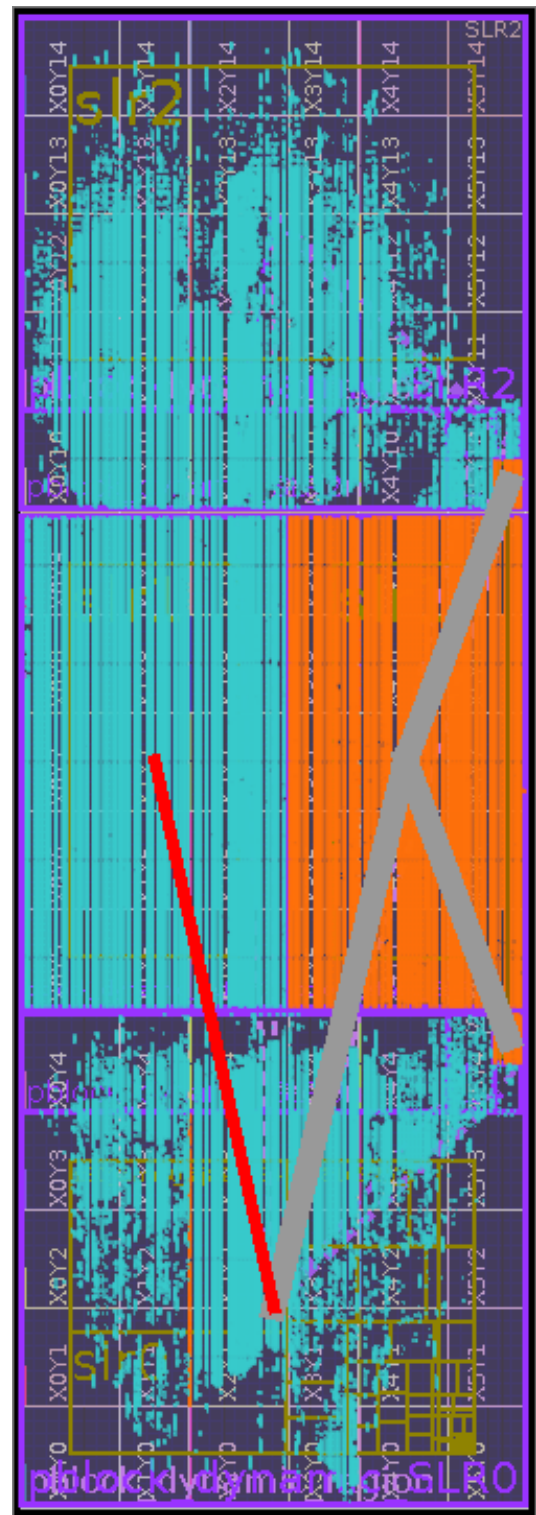


Abbildung 6.3.: Vergleich der Ressourcennutzung bei verschiedenen Datentypen



Netzliste für den Datentyp int



Netzliste für den Datentyp float

Abbildung 6.4.: Netzliste des Box-Filter-Kernels mit verschiedenen Datentypen. Die unterschiedliche Auslastung der Hardware-Ressourcen ist in dieser Ansicht deutlich sichtbar. Blaue Bereiche werden für die Schaltung verwendet, dunkelgrau sind nicht in Gebrauch. Der orange Bereich ist für die Laufzeitumgebung reserviert. Die kräftigen grauen und roten Balken repräsentieren die Zahl der Verbindungen zwischen den einzelnen FPGA-Komponenten.

7. Fazit

Es konnte gezeigt werden, dass SYCL sich als Backend für die Alpaka-Bibliothek grundsätzlich eignet. Da zur Zeit noch gravierende Inkompatibilitäten zwischen SYCL und Alpaka bestehen, ist eine Entwicklung über den Prototypen-Status im Moment nicht durchführbar. Diesbezüglich sind noch einige konzeptionelle Änderungen in Alpaka und/oder SYCL notwendig.

Aufgrund des Entwicklungsstands der verfügbaren SYCL-Implementierungen ist eine Ausführung von Alpaka-Programmen auf den meisten Hardware-Plattformen derzeit nicht möglich. Lediglich Intel-CPU und -GPU sind zur Zeit nutzbar, sofern man die Beschränkungen des Backend-Prototypen in Kauf zu nehmen bereit ist.

Gegenüber Alpaka bietet SYCL das modernere, intuitivere und standardnähere Programmier-Interface. Das fragmentierte Ökosystem sowie der eingeschränkte Hardware-Support sprechen im Moment noch recht deutlich gegen die Nutzung dieses insgesamt vielversprechenden Ansatzes zur parallelen Programmierung. Es bleibt deswegen abzuwarten, ob SYCL in den nächsten Monaten und Jahren weitere Verbreitung erfährt oder in der momentanen relativen Bedeutungslosigkeit verharret. Wichtig wäre hier eine baldige vollständige Unterstützung von NVIDIA-GPUs, die den Bereich des HPC mit weitem Abstand dominieren. Auch für moderne AMD-GPUs existiert bislang keine SYCL-Implementierung, die auf dem HPC-Sektor vorbehaltlos einzusetzen ist. Dadurch kann SYCL in der GPGPU-Programmierung keine Rolle spielen, sofern man von den vergleichsweise leistungsschwachen Intel-Laptop-GPUs absieht. Diesbezüglich wird interessant sein, ob und wie gut Intels zukünftige dedizierte GPU-Plattform mit SYCL genutzt werden kann.

Für die Zukunft der Alpaka-Bibliothek wäre eine weniger starke Anlehnung an CUDA wünschenswert. Diese ist zwar durch die historische Entwicklung der Alpaka-Bibliothek begründet, erschwert aber die Integration konkurrierender Plattformen, wie die Probleme bei der Entwicklung des SYCL-Backends zeigten. Beispielhaft sei hier die Unterstützung unterschiedlicher Beschleunigertypen und/oder Hersteller genannt, die erst zur Laufzeit des Programms bekannt sind. Dieses Problem betrifft nicht nur SYCL, sondern auch neuere Fassungen des OpenMP-Standards. Nicht nur an dieser Stelle ist daher eine Diskussion über das zukünftige Design der Bibliothek innerhalb der Alpaka-Entwicklergemeinschaft notwendig. Trotz alledem ist Alpaka für den Einsatz in der Praxis gut geeignet, wie bereits produktiv genutzte Programme auf Alpaka-Basis beweisen.

Die Programmierbarkeit von FPGAs sowohl durch das Alpaka-Backend als auch durch SYCL selbst ist langfristig eine attraktive Option. Hier dürfte zunächst die Intel-Implementierung des SYCL-Standards von Bedeutung sein, da eine Unterstützung der hauseigenen FPGAs mittelfristig wahrscheinlich ist und die bisherige Implementierung für CPUs und GPUs recht ausgereift wirkt.

Dem gegenüber steht die auf der Intel-Implementierung aufsetzende Xilinx-Implementie-

rung, die sich noch in einem sehr frühen Entwicklungsstadium befindet und unter einer sehr kleinen Entwicklerzahl leidet. Es ist dennoch lohnenswert, das Projekt der FPGA-Unterstützung für beide Hersteller in Alpaka weiter zu verfolgen: Einerseits hätte Alpaka damit ein wichtiges Alleinstellungsmerkmal gegenüber vergleichbaren Projekten wie Kokkos oder HPX.Compute. Andererseits sind die Hardware- und Parallelisierungseigenschaften der FPGAs für viele Algorithmen – die zum Teil schon in einer Alpaka-Fassung vorliegen – sehr interessant. Insbesondere Programme aus dem Umfeld der Laser-Teilchenbeschleunigung könnten von einer physischen Schaltung profitieren, die auf der Basis eines bildverarbeitenden Algorithmus synthetisiert und an einen mit hoher Messfrequenz betriebenen Detektor angeschlossen wurde.

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).
- [Ama] Amazon Web Services, Inc. *Amazon EC2 F1-Instances*. URL: <https://aws.amazon.com/de/ec2/instance-types/f1/> (besucht am 22.11.2019).
- [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).
- [Di +17] Lorenzo Di Tucci u. a. „The Role of CAD Frameworks in Heterogeneous FPGA-Based Cloud Systems“. In: *IEEE 35th International Conference on Computer Design*. Nov. 2017, S. 423–426. DOI: 10.1109/ICCD.2017.74.
- [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.
- [Fly66] Michael J. Flynn. „Some Computer Organizations and Their Effectiveness“. In: *IEEE Transactions on Computers* Jahrgang C-21. Ausgabe 9 (Sep. 1966), S. 948–960. DOI: 10.1109/TC.1972.5009071.
- [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. DOI: 10.1145/355841.355847.
- [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.
- [NF17] Masahiro Nakamura und Norishige Fukushima. „Fast Implementation of Box Filtering“. In: *Proceedings of the International Workshop on Advanced Image Technology (IWAIT)*. Jan. 2017.
- [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

2.1. abstrakter FPGA-Aufbau [nach HS10, S. 10–14]	18
2.2. Aufbau eines XCU200-FPGAs [nach Xil19a, S. 5]	20
2.3. spaltenweise Verteilung der FPGA-Ressourcen [nach Xil19e, S. 22]	21
2.4. Aufteilung der FPGA-Ressourcen auf <i>clock regions</i> [nach Xil19e, S. 22]	22
2.5. Y-Diagramm nach Gajski [nach KB13, S. 10]	23
2.6. Modell eines endlichen Automaten (Moore-Schaltwerk) [nach KB13, S. 35]	24
2.7. Verhalten des 2-Bit-Registers	24
2.8. FPGA-Implementierung einer mehrstufigen Berechnung [nach Xil19b, S. 21]	29
2.9. FPGA-Pipeline-Architektur [nach Xil19b, S. 22]	30
3.1. SYCLs Plattform-Modell [nach Mun12, S. 23]	38
3.2. SYCLs Indexraum [nach Mun12, S. 25]	39
3.3. Einfacher Aufgabengraph	41
3.4. Darstellung der Feldpartitionierungsstrategien	46
4.1. Schematischer Aufbau der Alpaka-Bibliothek	48
4.2. Links: Abstraktionshierarchie mit einem aus <i>Threads</i> zusammengesetzten <i>Grid</i> . Rechts: Ein hypothetischer Prozessor, der datenparallele Anwendungen mit die- sem Abstraktionsschema ideal ausführen könnte. [nach Wor15, S. 18]	56
4.3. Links: Abstraktionshierarchie mit einem <i>Grid</i> , das aus zu <i>Blocks</i> gruppierten <i>Threads</i> besteht. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von <i>Threads</i> auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der <i>Blocks</i> ermöglicht. [nach Wor15, S. 19]	57
4.4. Links: Abstraktionshierarchie mit einem <i>Grid</i> , das aus zu <i>Blocks</i> gruppierten <i>Warps</i> besteht. Letzere sind wiederum aus mehreren <i>Threads</i> zusammengesetzt. Rechts: Ein theoretischer Prozessor, der eine 1-zu-1-Abbildung von <i>Threads</i> auf Kerne sowie schnelle Synchronisation und Kommunikation innerhalb der <i>Blocks</i> er- möglicht. Durch die taktgenaue gemeinsame Ausführung der <i>Threads</i> in Form von <i>Warps</i> kann Chipfläche eingespart werden. [nach Wor15, S. 20]	57
4.5. Vollständiges Alpaka-Abstraktionskonzept: Mehrere <i>Elements</i> werden von einem <i>Thread</i> verarbeitet. Mehrere <i>Threads</i> werden innerhalb eines <i>Warps</i> taktgenau gemeinsam ausgeführt, mehrere <i>Warps</i> bilden wiederum einen <i>Block</i> . Die Men- ge aller <i>Blocks</i> ergibt das <i>Grid</i> . [nach Wor15, S. 22]	58
4.6. Struktureller Aufbau der Alpaka-Bibliothek[nach Wor15, S. 40]	61
5.1. Struktureller Aufbau des SYCL-Alpaka-Backends	74

6.1. Visualisierung eines Ergebnisses des <i>jungfrau-photoncounters</i>	81
6.2. Vergleich der für die HLS notwendigen Zeitaufwände bei verschiedenen Datentypen	84
6.3. Vergleich der Ressourcennutzung bei verschiedenen Datentypen	85
6.4. Netzliste des synthetisierten Box-Filter-Kernels mit verschiedenen Datentypen .	86

Tabellenverzeichnis

2.1. Ressourcen der dynamischen Regionen eines XCU200-FPGAs [siehe Xil19a, S. 5]	20
3.1. Übersicht der verfügbaren SYCL-Implementierungen	43

Quelltextverzeichnis

2.1. <i>Entity</i> eines 2-Bit-Registers [siehe KB13, S. 26]	23
2.2. Verhaltensbeschreibung eines 2-Bit-Registers [siehe KB13, S. 28]	25
2.3. Strukturbeschreibung eines 2-Bit-Registers [siehe KB13, S. 36]	26
3.1. Struktur eines SYCL-Programms	32
3.2. Auswahl eines Xilinx-FPGAs und Erzeugung einer zugehörigen <i>Queue</i>	33
3.3. Ausführliche Beschleunigerwahl und <i>Queue</i> -Konstruktion	34
3.4. Speicherreservierung und -initialisierung in SYCL	35
3.5. Struktur einer <i>command group</i>	35
3.6. Struktur einer <i>command group</i> mit Kernel-Aufruf	36
3.7. Struktur einer <i>command group</i> mit Kernel-Aufruf	36
3.8. AXPY – vollständiges SYCL-Beispiel	37
3.9. Einfacher SYCL-Aufgabengraph	40
3.10. Datenfluss-Erweiterung in SYCL	44
3.11. Pipeline-Erweiterung in SYCL	45
3.12. Feldpartitionierung in SYCL	46
4.1. Struktur eines Alpaka-Programms	48
4.2. Auswahl der in Alpaka vorhandenen NVIDIA-CUDA-Implementierung	49
4.3. Spezialisierung abstrakter Alpaka-Klassen	49
4.4. Auswahl der Alpaka-Befehlswarteschlange	49
4.5. Instanziierung der Alpaka-Datentypen	49
4.6. Definition eines Größenvektors mit Alpaka	50
4.7. Speicherallokation mit Alpaka	50
4.8. Initialisierung eines Alpaka-Puffers	50
4.9. Kopie der initialisierten Daten mit Alpaka	50
4.10. Arbeitsaufteilung durch Alpaka-Schätzfunktion	51
4.11. Kernel-Definition in Alpaka	52
4.12. Task-Definition und -Ausführung in Alpaka	52
4.13. Synchronisation zwischen Host und Device in Alpaka	53
4.14. Vollständiges Alpaka-AXPY-Beispiel	54
4.15. Einfacher Alpaka-Aufgabengraph	59
5.1. Aufbau des SYCL-Kernel-Tasks	66
5.2. Aufruf der Alpaka-Kernel-Funktion im SYCL-Kernel	68
5.3. Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Program- mierung	69

5.4. Nutzung der Template-Meta-Funktionen zur Umwandlung der Puffer-Wrapper in SYCL-accessor-Typen	70
5.5. Nutzung der Template-Meta-Funktionen zur Umwandlung der accessor-Typen in Zeiger	70
5.6. Aufbau eines SYCL-Kopier-Tasks	71
A.1. VHDL-Quelltext eines 2-Bit-Flipflops [siehe KB13, S. 39]	99
A.2. VHDL-Quelltext eines 2-Bit-Multiplexers [siehe KB13, S. 39–40]	100
A.3. Verwendung von SYCL-Ausnahmefehlern	101
A.4. Verwendung des SYCL-Profilings	102
A.5. Implementierung des Konstruktors der TaskKernelSycl-Klasse	102
A.6. Implementierung des SYCL-Puffer-Wrappers	103
A.7. Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Program- mierung	104
A.8. SYCL-Box-Filter-Kernel	105

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]

```
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. Verwendung von SYCL-Ausnahmefehlern

```
#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 A.3.: Verwendung von SYCL-Ausnahmefehlern

A.2.2. Verwendung des SYCL-Profilings

```
#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 A.4.: Verwendung des SYCL-Profilings

A.2.3. Implementierung des TaskKernelSycl-Konstruktors

```
template <typename TWorkDiv>
TaskKernelSycl(TWorkDiv&& workDiv,
               const TKernelFnObj& kernelFnObj,
               const TArgs&... args)
: workdiv::WorkDivMembers<TDim, TIdx>(std::forward<TWorkDiv>(workDiv))
, m_kernelFnObj{kernelFnObj}
, m_args{args...}
{
}
}
```

Quelltext A.5.: Implementierung des Konstruktors der TaskKernelSycl-Klasse

A.2.4. 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.6.: Implementierung des SYCL-Puffer-Wrappers

A.2.5. Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Programmierung

```

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 A.7.: Umwandlung der Puffer in SYCL-accessor-Typen durch Template-Meta-Programmierung

A.2.6. Box-Filter-Kernel

```

cgh.single_task<box_filter>([=]() { for(auto y = 0ul; y < dim_y; ++y) {
    for(auto c = 0ul; c < (dim_x / chunk_size); ++c) {
        const auto cl_id = cl::sycl::id<2>{c * chunk_size - 1ul, y};
        const auto left = (c == 0ul) ? 0 : buffers_in[cl_id];
        /* right analog */

        // lese 3 Zeilen aus globalem in lokalen Speicher
        cl::sycl::xilinx::pipeline([&]() {
            for(auto x = 0ul; x < chunk_size; ++x) {
                if(y == 0ul) {
                    const auto local = cl::sycl::id<2>{x, 0ul};
                    local_in[local] = 0.0;
                } else {
                    const auto global = cl::sycl::id<2>{
                        c * chunk_size + x, y - 1ul};
                    const auto local = cl::sycl::id<2>{x, 0ul};
                    local_in[local] = buffers_in[global];
                }
            }
        }); /* Zeilen 2 und 3 analog */

        cl::sycl::xilinx::pipeline([&]() { // box filter
            for(auto x = 0ul; x < chunk_size; ++x) {
                const auto tl_id = cl::sycl::id<2>{x - 1ul, 0ul};
                const auto tl = (x == 0ul) ? left : local_in[tl_id];
                /* cl und bl analog */
                const auto tc_id = cl::sycl::id<2>{x, 0ul};
                const auto tc = local_in[tc_id];
                /* cc und bc analog */
                const auto tr_id = cl::sycl::id<2>{x + 1ul, 0ul};
                const auto tr = (x == (chunk_size - 1ul)) ?
                    right : local_in[tr_id];
                /* cr und br analog */
                const auto top = tl + tc + tr;
                const auto center = cl + cc + cr;
                const auto bottom = bl + bc + br;

                const auto local = cl::sycl::id<1>{x};
                local_out[local] = (top + center + bottom) / 9;
            }
        });

        // schreibe Ergebnis in globalen Speicher
        cl::sycl::xilinx::pipeline([&]() {
            for(auto x = 0ul; x < chunk_size; ++x) {
                const auto global = cl::sycl::id<2>{c * chunk_size + x, y};
                const auto local = cl::sycl::id<1>{x};
                buffers_out[global] = local_out[local];
            }
        });
    }
});

```

Quelltext A.8.: SYCL-Box-Filter-Kernel

B. Online-Diskussionen

B.1. Diskussionen mit dem SYCL-Spezifikationskomitee

B.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.

B.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.

B.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 ambiguous.

Ronan Keryell (Xilinx) Interestingly, Intel is trying hard to hide what you are asking for: `intel11vm#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

¹Verweis 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.

B.2. Diskussionen mit Xilinx-Angestellten

Danksagung

Für hilfreiche Antworten und Anregungen während der Implementierung des Alpaka-SYCL-Backends bedanke ich mich herzlichst bei Herrn René Widera. Großer Dank gebührt darüber hinaus Herrn Jonas Schenke, der mir während der Arbeit mit dem *jungfrau-photoncounter* noch in den tiefsten Abendstunden mit zahlreichen Ratschlägen zur Seite stand.

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, 28. November 2019

Jan Stephan