INAUGURAL - DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht - Karls - Universität
Heidelberg

vorgelegt von

M.Sc. Markus Gipp
aus Mannheim

Tag der mündlichen Prüfung: 7. Mai 2012

Online- und Offline-Prozessierung von biologischen Zellbildern auf FPGAs und GPUs

Betreuer: Prof. Reinhard Männer
Prof. Holger Fröning

Für meine liebe Sabine

Abstract

This work is about images from a high throughput microscopy. Because of the huge amount of images, the analysis has to be processed in an automatic way. There are two approaches: the offline processing, image processing on a computer cluster, and the online processing, image processing of the streaming data from the sensors.
To cope with the image data in the offline processing this work uses graphics cards as accelerators and shows an CUDA implementation of the Haralick Texture Features. The accelerated version achieves a speed up of around 1000 against a CPU solution. This offers the biologist the opportunity to do more tests and leads to a faster gain of knowledge.
The online processing uses FPGAs which are easy to connect to the sensors. The biologists have the constraint to adapt the algorithm for their future needs. This work presents a developed OpenCL to FPGA compiler prototype. The algorithm can be written in OpenCL and compiled for the FPGA without any knowledge of any hardware description language. Furthermore, OpenCL is a portable language between several computing architectures. If an algorithm written in OpenCL is too complex for the FPGA compiler due to the existing restrictions, then a compilation for the GPUs in the offline processing environment is still possible.

Keywords: **CUDA**, **Co-Processor**, **Compiler**, **DPR**, **FPGA**, **GPGPU**, **haralick texture features**, **hardware syntheses**, **highthoughtput microscopy**, **HPC**, **LLVM**, **OpenCL**, **pipeline generator**, **reconfigurable hardware**, **VHDL**

Kurzbeschreibung

Wenn Bilder von einem Mikroskop mit hohem Datendurchsatz aufgenommen werden, müssen sie wegen der großen Bildmenge in einer automatischen Analyse prozessiert werden. Es gibt zwei Ansätze: die Offlineprozessierung, die Verarbeitung der Bilder auf einem Cluster, und die Onlineprozessierung, die Verarbeitung des Pixelstroms direkt von den Sensoren.
Für die Bewältigung der Bilddaten in der Offlineprozessierung setzt diese Arbeit auf Grafikkarten und demonstriert eine Implementierung der Haralick-Bildmerkmalerkennung in CUDA. Dabei wird der Algorithmus um den Faktor 1000, gegenüber einer CPU-Lösung, beschleunigt. Dies ermöglicht den Biologen weitere Tests und einen schnelleren Erkenntnisgewinn.
Die Onlineprozessierung setzt auf FPGAs, die sich mit den Sensoren elektrisch verbinden lassen. Dabei soll sich der Algorithmus dem Bedarf der Biologen entsprechend verändern lassen. Diese Arbeit zeigt die Entwicklung eines OpenCL-FPGA-Kompilierer-Prototyps. Die Biologen können Algorithmen in OpenCL schreiben und in ein Hardwaredesign für den FPGA übersetzen, was in einer Hardwarebeschreibungssprache für sie zu komplex wäre. Neben der Einfachheit hat die parallele Sprache OpenCL den Vorteil der Portierbarkeit auf andere Architekturen. Falls der FPGA-Kompilierer wegen existierender Einschränkungen den Algorithmus nicht übersetzen kann, lässt sich das OpenCL-Programm auch für die GPUs in der Offlineprozessierung übersetzen.

Schlüsselworte:
CUDA, **Compiler**, **Co-Prozessor**, **DPR**, **FPGA**, **GPGPU**, **LLVM**, **OpenCL**, **Pipelinegenerator**, **VHDL**, **rekonfigurierbare Logik**

Abkürzungen

ALU	*Arithmetic Logical Unit*, arithmetische Recheneinheit
AST	*Abstract Syntax Tree*, abstrakter Syntaxbaum
CLB	*Configurable Logic Block*, konfigurierbare Logikzelle
CMP	*Compare*, Vergleich
CPU	*Central Processing Unit*, Computerprozessor
CU	*Compute Unit (OpenCL)*, Recheneinheit in OpenCL
CUDA	*Compute Unified Device Architecture*, Programmiersprache für Grafikkarten
DDR3	*Double Data Rate 3*, Speicherzugriffstechnik
DMA	*Direct Memory Access*, direkter Speicherzugriff
DPR	*Dynamic Partial Reconfiguration*, dynamische partielle Rekonfiguration
FIFO	*First in First out*, Stapelspeicher
FPGA	*Field Programmable Gate Array*, feldprogrammierbare Gatteranordnung
FPU	*Floating Point Unit*, Fließkomma-Recheneinheit
GPGPU	*General Purpose GPU*, GPU für allgemeine Zwecke
GPU	*Grafics Processing Unit*, Grafikprozessor
HPC	*High Performance Computing*, beschleunigtes Rechnen
IR	*Intermediate Representation*, Zwischensprache
LLVM	*Low Level Virtual Machine*, Kompiliererframework
MEM	*Memory*, Speicher
PCIe	*Peripheral Component Interconnect Express*, Computerschnittstelle
PE	*Processing Element (OpenCL)*, prozessierende Einheit in OpenCL
SIMD	*Single Instruction Multiple Data*, Spezifikation einer Rechnerarchitektur
SSA	*Single Static Assignment*, einmalige statische Zuweisung
TCL	*Tool Command Language*, Skriptsprache
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*, Hardwarebeschreibungssprache

Danksagung

Herrn Professor Reinhard Männer danke ich sehr, mich an seinem Lehrstuhl aufgenommen und mir eine Promotion ermöglicht zu haben. Seine Unterstützung mit seiner menschlichen und freudigen Art machten ihn zu einem idealen Doktorvater.
Diese Arbeit wäre auch nicht, wie sie ist, wenn mir Guillermo Marcus mit seinem Scharfsinn für Details keine nützlichen Tipps für die Umsetzung gegeben hätte. Weiter danke ich ihm und Wenxue Gau, deren ausgereifte Entwicklungen (PCI-Treiber und DMA-Logik) ich verwenden durfte, ohne die meine Arbeit konzeptionell nicht vollständig wäre.
Neben den beiden war auch der Austausch mit Andreas Kugel, Thomas Gerlach und Nicolai Schnör stets hilfreich, wenn es, wie häufig in der Hardwareentwicklung, mit den Entwicklungswerkzeugen nicht weiterging.
Ich habe mich am Lehrstuhl mit all seinen freundlichen Mitarbeitern und der entspannten als auch kreativen Atmosphäre sehr wohl gefühlt.
Danke auch an die Korrekturleser: Heike Hildenbrand und Thomas Haas, sie machten meine Sätze dieser Arbeit lesbarer und verständlicher.
Besonderer Dank geht an meine Frau und unsere Familien, die mir in der schwierigen Zeit durch die Promotion stets Halt gaben.

Inhalt

1. Einführung und Ziele des Viroquant-Projekts

1.1. Orientierung im Viroquant-Projekt

Weltweit sterben jährlich drei Millionen Menschen an einer HIV-Infektion. Da sich 60 Millionen Menschen im selben Zeitraum infizieren, wird die Zahl an HIV-Toten ohne Heilmittel stark ansteigen. An chronischen Hepatitis-Infektionen sterben jährlich zwei Millionen Menschen, erschreckende 500 Millionen erkranken. Diese Zahlen stammen aus dem Projektantrag für das VIROQUANT-Projekt und sie alleine stellen eine ausreichende Motivation dar, um aussichtsreiche Forschungen zu fördern, die entweder Infizierten helfen können oder Neuinfektionen reduzieren sollen.

Im VIROQUANT-Projekt wird an den Interaktionen zwischen Viren mit Zellen geforscht.
Bereits in der Vergangenheit wurden Zellen mit Viren infiziert und es wurde beobachtet, wie die Viren in die Zelle eindringen, sich vermehren und wieder austreten. Die Untersuchung konzentrierte sich auf einzelne Proteine oder Gene, was auch der Grund dafür ist, nur wenig allgemeine Aussagen treffen zu können.
Ein systembiologischer Ansatz verspricht, allgemeinere und zielgerichtete Aussagen treffen zu können. Die Systembiologie ist eine junge Disziplin, in der die Zelle als biologisches System mit allen vorhandenen Zellbausteinen, biologischen Prozessen und Interaktionsmöglichkeiten in einem mathematischen Modell betrachtet wird. Lebensprozesse mit mathematischen Formeln zu beschreiben, ist keine leichte Aufgabe, aber der Erkenntnisgewinn wird groß sein. Wirkungsvolle Medikamente mit möglichst wenigen Nebenwirkungen herstellen zu können, ist ein Zukunftsziel dieser Forschung.

An einem vereinfachten Beispiel soll verdeutlicht werden, wie eine systembiologische Anwendung funktioniert:
Zellen besitzen nach heutigem Kenntnisstand ca. 20–25 Tausend Gene, in denen alle vererbbaren Informationen gespeichert sind. Einzelne Gene, oder auch Kombinationen davon, sind für die Proteinproduktion (Enzyme) verantwortlich. Die Enzyme werden für chemische Reaktionen benötigt, beispielsweis einer Zellteilung, aber auch Viren brauchen Enzyme als Nahrung.

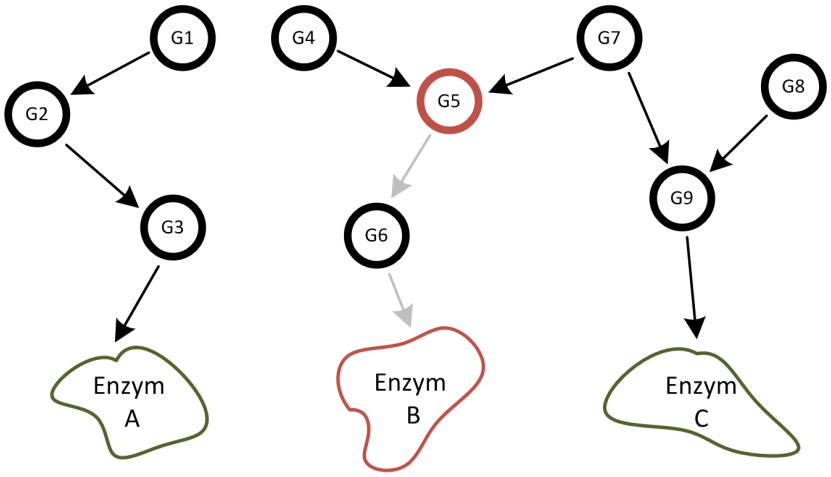


Bild 1.1.: Vereinfachtes Schaubild von Signalwegen zwischen Genen, die Enzyme produzieren. Gen fünf (G5) ist inaktiv, der Signalweg zu Enzym B ist unterbrochen, somit wird es nicht produziert.

Zum Zweck einer Untersuchung gibt es die Möglichkeit, Gene in lebenden Zellen mit einem speziellen Verfahren zu inaktivieren. Die Folge ist, dass diese Zellen bestimmte Enzyme nicht mehr produzieren. Folglich können die Viren, die vom speziellen Enzym abhängig sind, sich nicht mehr reproduzieren oder weiterleben und die infizierte Zelle nimmt keinen weiteren Schaden. Benötigen die Viren dieses Enzym nicht, wird die Zelle konsumiert und stirbt ab.
Ein Forschungsgebiet liegt darin, den Zusammenhang zwischen Genen und der Enzymproduktion zu verstehen. Meist sind mehrere Gene für die Produktion einzelner Enzyme verantwortlich. Umgekehrt sind aber auch einzelne Gene für die Produktion mehrerer Enzyme zuständig. Um den Zusammenhang zwischen Genen und Enzymen besser verstehen zu können, werden Netzwerke bestehend aus Signalwegen konstruiert, die das Wissen veranschaulichen, welche Gene für welche Enzyme zur Produktion benötigt werden. Das Bild 1.1 zeigt eine Unterbrechung des für die Enzymproduktion wichtigen Zweigs.

Um die eben genannte Beispielanwendung durchführen zu können, werden mehrere Experten aus unterschiedlichen Gebieten benötigt. Zu ihnen zählen Biologen, Physiker, Mathematiker, Informatiker und Ingenieure. Im VIROQUANT-Projekt sind die Aufgaben in drei Bereiche unterteilt.

A Biologie. In diesem Bereich werden die Zellkulturen angelegt, mit den Viren infiziert und die Zellbilder ausgewertet.

B Modellierung. Eng in Zusammenarbeit mit Bereich A wertet dieser Bereich die Zellbilder aus und erweitert mathematische Modelle mit den Messergebnissen.

C Technologie. Dieser Bereich arbeitet an der Verbesserung des Mikroskops und an den Bildalgorithmen zur Auswertung der Zellbilder zusammen mit den Bereichen A und B.

In den folgenden Abschnitten beschäftigt sich diese Dissertation weiter mit den Aufgaben aus dem Technologie Bereich C.

1.2. Hochdurchsatzmikroskopie

Für einen sogenannten „Genomweiten Screen“, die Untersuchung der Signalwege bestimmter Gene, sind sehr viele Aufnahmen von Zellbildern nötig. Für diese Menge ist ein Hochdurchsatzmikroskop erforderlich, das sehr schnell die Zellkulturen fotografiert und mit einem Roboterarm die Proben austauscht. Im Rahmen des Projekts werden existierende Mikroskope in ihrer Arbeitsweise beschleunigt, um der Anforderung gerecht zu werden. Bild 1.2 zeigt schematisch alle nötigen Komponenten für die Datenaufnahme.

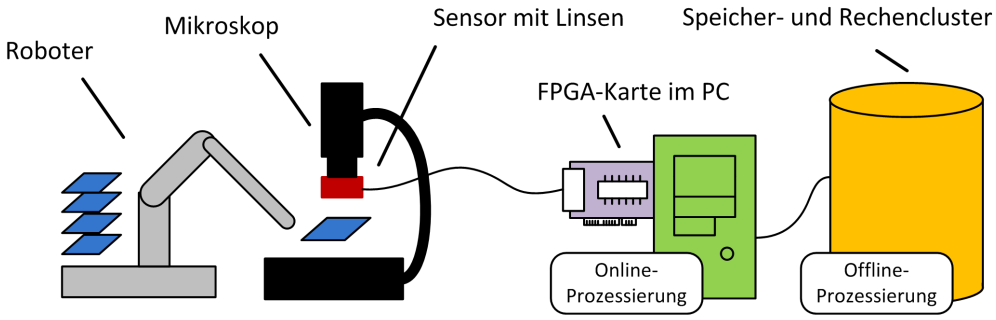


Bild 1.2.: Hochdurchsatzmikroskop mit verbundenen Komponenten.

Eine Konsequenz eines schnell arbeitenden Mikroskops, ist ein höherer Datendurchsatz von mehreren hundert Megabytes pro Sekunde. Die Aufnahmen werden von den Fotosensoren zu einer Ausleseelektronik übertragen. Als Ausleseelektronik wird oft eine FPGA-Karte in einem PC verwendet. Diese hat neben dem Datentransport zum Massenspeicher auch die Aufgabe einer Vorprozessierung der Bilddaten. Typische Aufgaben sind Intensitätskorrekturen der Pixeldaten, Sortierung des Pixelstroms bzw. die Prozessierung einfacher Bildalgorithmen zur Qualitätsanalyse. Dabei existiert die Anforderung, das System neuen Gegebenheiten anpassen zu können. Wegen des kontinuierlichen Datenstroms muss die Verarbeitung in Echtzeit geschehen, was in dieser Arbeit als „Online-Prozessierung“ bezeichnet wird.

1.3. Beschleunigte Bildverarbeitung

Bei ansteigendem Datenvolumen und ansteigenden Datenraten wird die Speicherung und die Datenauswertung zum Problem. Letztendlich benötigt man immer größere Massenspeicher (Daten-Cluster) bei gleich bleibender Geschwindigkeit der Datenauswertung bzw. immer leistungsfähigere Rechencluster bei gleich bleibender Speicherkapazität, um die Bilddaten auszuwerten.
Für einen genomweiten Screen wird für jedes Gen eine Zellkultur, Bild 1.3, mit vielen hundert Zellen aufgenommen. Das aufkommende Datenvolumen ist somit ein Produkt hunderter Zellen und tausender Gene. Ohne beschleunigtes Mikroskop benötigt ein genomweiten Screen Wochen für die Datenaufnahme. Mit dem beschleunigten Mikroskop werden nur noch Tage benötigt, was einer ca. 20-fachen Beschleunigung entspricht.

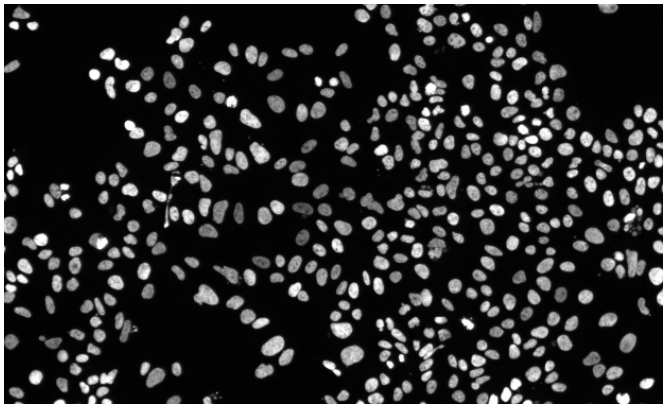


Bild 1.3.: Bildausschnitt einer Zellkultur mit hunderten Zellen.

Es versteht sich von selbst, dass eine manuelle Auswertung der Zellbilder bei der enormen Zellenanzahl bzw. Datenmenge nicht mehr möglich ist. Aus diesem Grund werden Algorithmen aus der Bildverarbeitung zur automatischen Analyse verwendet. Um möglichst genau mit der manuellen Auswertung übereinzustimmen, wurden Algorithmen identifiziert, die sich besonders gut eignen. Besonders der Haralick Texturen Merkmal Algorithmus bietet sehr gute Ergebnisse. Jedoch besitzt er den Nachteil hoher Rechenintensität, so dass einzelne Computer und sogar Rechen-Cluster Monate benötigen, um die Bildmerkmale für alle Zellen eines genomweiten Screens zu berechnen. Schon mit dem alten Mikroskop existiert eine große Diskrepanz der Aufnahmezeit von Wochen und der benötigten Zeit der Datenauswertung von Monaten. Die Auswertung der aufgenommenen Daten, die auf einem Daten-Cluster gespeichert werden, wird weiter als „Offline-Prozessierung“ bezeichnet. Wäre die Offline-Prozessierung ebenso schnell wie die Aufnahmezeit, die der Online-Prozessierung entspricht, könnte auf die Speicherung der Daten verzichtet werden, bzw. es werden nur noch die Aufnahmen dauerhaft

gespeichert, die von hohem Interesse sind. Verständlicherweise liefern nicht alle aufgenommenen Daten nützliche Ergebnisse. Mit dem beschleunigten Mikroskop ändert sich die Situation dahingehend, dass nicht nur weniger Zeit für die Datenaufnahme benötigt wird, sondern auch, dass der Datendurchsatz und die anfallende zu speichernde Datenmenge um den Beschleunigungsfaktor 20 ansteigen. Auch die Diskrepanz zwischen Aufnahmezeit und Datenanalysezeit verschlechtert sich weiter, und mehr kostenintensive Daten-Cluster werden benötigt, die viel Energie verbrauchen, um die Daten abrufbereit zu halten. Eine Beschleunigung der Offline-Prozessierung ist außerordentlich wichtig, um

- die Datenmengen, die gespeichert werden müssen, zu reduzieren.
- rechenintensive Algorithmen in der Offline-Prozessierung verwenden zu können.
- den Geschwindigkeitsgewinn des beschleunigten Mikroskops auf die Offline-Prozessierung zu übertragen, um Ergebnisse schneller auswerten zu können.

Eine Konsequenz ist, den bestehenden Rechen-Cluster durch weitere Rechner (CPUs) zu erweitern. CPUs eignen sich für alle Rechenaufgaben, was bedeutet, dass sich durch eine vermehrte Anzahl eine Beschleunigung erzielen lässt, sofern die Anzahl nicht zu groß wird. Ebenso kann eine Beschleunigung durch Beschleunigerkarten realisiert werden, die in den bestehenden Rechner (Knoten) eingesetzt werden. Beispiele für Beschleunigerkarten sind FPGAs und GPUs. In der Regel wird von ihnen eine höhere Beschleunigung gegenüber CPUs im Bereich der parallelen Algorithmen erwartet, wie z. B. bei der Bildverarbeitung. Letzteres ist der kostengünstigere und energiesparsamere Ansatz. Eine Erweiterung des Rechen-Clusters mit mehr Knoten (und Beschleunigerkarten) wird dadurch nicht ausgeschlossen.

1.4. Forschungsfragen

In dieser Arbeit werden zwei Themengebiete behandelt: Erstens die Beschleunigung des Haralick Bildmerkmalalgorithmuses, um die Geschwindigkeitsdiskrepanz zwischen der Datenaufnahme und Offline-Prozessierung zu reduzieren. Zweitens die Entwicklung einer vereinfachten Programmierung der FPGA-Karte für die Online-Prozessierung der aufgenommenen Bilddaten.

Mit einzelnen Fragen werden zentrale Forschungsfragen rund um das jeweilige Themengebiet entwickelt, die in dieser Dissertation beantwortet werden:

- Lässt sich der Haralick Bildmerkmalalgorithmus parallelisieren?
- Welche Co-Prozessoren würden für eine Beschleunigung in Frage kommen? Welche Rechenarchitektur passt besonders gut für die Beschleunigung?

- Welche Beschleunigerkarte ist besser geeignet: Grafikkarte oder FPGA-Karte?
- Lässt sich der Algorithmus im Bereich von zwei bis drei Größenordnungen gegenüber einer CPU beschleunigen, um den Biologen Wartezeiten zu ersparen?
- Welche Einflussfaktoren behindern eine weitere Beschleunigung?
- Rechtfertigt ein weiterer Arbeitsaufwand eine weitere Beschleunigung?
- Wie verhält sich die Entwicklungszeit einer GPU-Lösung zu einer FPGA-Lösung?
- Welche Programmiersprachen können genutzt werden, um die Entwicklungszeit eines FPGA-Designs zu beschleunigen und zu vereinfachen?
- Welchen Vorteil besitzt eine parallele Programmiersprache, wie z. B. OpenCL, gegenüber einer seriellen Programmiersprache zur FPGA-Beschreibung?
- Worin liegen die Unterschiede einer Architektur, in der eine Pipeline entsteht, gegenüber einer Architektur mit gleich bleibendem Rechenwerk?
- Wie sieht die Struktur eines Kompilierers für eine Pipelinegenerierung aus?
- Welche weitere Logik wird an den Schnittstellen zur generierten Pipeline benötigt?

Für das Themengebiet der Onlineprozessierung ergibt sich folgende zentrale Forschungsfrage: „**Wie sehen die Bausteine einer Übersetzungskette aus, die von der parallelen Programmiersprache OpenCL in eine VHDL-Hardwarepipeline für FPGAs übersetzen, und welche Vorteile bietet diese Struktur?**“ Zusammenfassend lässt sich für die Offlineprozessierung fragen: „**Wie lässt sich der Bildmerkmalalgorithmus von Haralick auf einer GPU beschleunigen, welcher Beschleunigungsfaktor gegenüber einer CPU ist zu erreichen und welche Einflussgrößen tragen maßgeblich zur Beschleunigung bei?**“

2. Grundlagen

2.1. Grafikkarten als Rechenbeschleuniger

2.1.1. Geschichtliche Entwicklung

Die *Graphics Processing Unit* (GPU) ist ein Prozessor, der die *Central Processing Unit* (CPU) für grafische Berechnungen entlastet. In den 90er-Jahren, als 3D-Computerspiele in Mode kamen, stieg der Bedarf, Rechenoperationen auf die GPU auszulagern. Es mussten drei-dimensionale Räume und Figuren mit Polygonen berechnet und im nächsten Schritt mit Texturen gefüllt werden (Rendern). Der stets wachsende Wunsch, 3D-Spiele immer realistischer werden zu lassen, beflügelte die Grafikkartenhersteller, immer leistungsfähigere GPUs mit immer leistungsfähigere Architekturen zu entwickeln. Wann die GPU für allgemeine Rechenaufgaben als Co-Prozessor herangezogen wurde, wird später im „Stand der Technik“, Abschnitt „GPGPU“ 3.1.2 behandelt.

2.1.2. Heutige GPU-Architektur

Heutige GPUs haben durch ihre hohe Anzahl von bis zu 512 Prozessorkernen ein vielfaches an Rechenleistung im Vergleich zu CPUs. Eine bewährte Technik, hunderte Prozessorkernen auf einem Chip zu vereinen und dessen Komplexität herunter zu brechen, ist, die Prozessorenkerne in skalierbar Strukturelemente zu gruppieren, die mehrfach auf dem Chip vorhanden sind. Dabei sind oft die Strukturelemente wiederum in gleiche Elemente unterteilt. Die GPU-Architektur, die hier beschrieben wird, ist die NVIDIA GF100-Architektur [62], die auf dem Vorgänger, der Fermi-Architektur [63], aufbaut. Die Strukturelemente der höchsten Chipebene heißen „grafische prozessierende Cluster“ (GPC). Vergleichbar mit Multikern-CPUs, besitzt jeder GPC den vollen Funktionsumfang einer GPU. Die Elemente der zweiten Ebene, aus denen die GPC bestehen, heißen Stream-Multiprozessoren (SM), die später weiter erläutert werden. Erst in der dritten Ebene sind die Threadprozessoren (TP, CUDA-Kerne) zu finden. Die dreistufige Struktur, zu sehen in Bild 2.1, ist in der Lage, bis zu 512 Threadprozessoren effizient zu verwenden.

Die schnellste in dieser Arbeit benutzte Grafikkarte (NVIDIA GTX 480) mit der GF100-

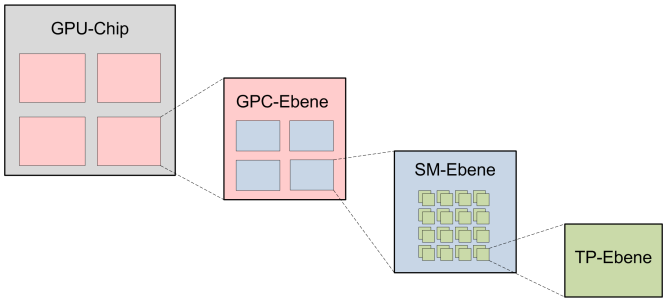


Bild 2.1.: Hierarchieebenen der GF100-Architektur

Architektur besteht aus vier GPCs. Jeder GPC kann aus bis zu vier SMs bestehen. Insgesamt gibt es in der GTX 480 allerdings nur 15 SMs, aus den vier GPCs mal vier SMs wird ein SM bestimmt und ausgeschaltet. Das wird gemacht, um eventuelle Herstellungsfehler, die den Ausfall einer SM zur Folge hätte, miteinzubeziehen. Entsprechend werden bei den anderen Grafikkarten der GF100-Serie (GTX 470 und GTX 465) mit Herstellungsfehlern höherer Dichte mehrere Einheiten abgestellt, um dennoch eine funktionstüchtige GPU mit geminderter Leistung zu erhalten. In jedem SM sind 32 TPs verbaut. Zusammen besitzt die GTX480 eben 480 Threadprozessoren ($15SM * 32TP$).

Auf der Chip-Ebene sind bis zu sechs 64 Bit breite GDDR5 Speicherkontroller über ein Kommunikationsnetz an die GPC angeschlossen. Die Gesamtbreite der Speicherschnittstelle umfasst 384 Bit, die kleiner ist als die der Vorgängerarchitektur. Sie besitzt aber wegen der höheren Transferrate von 1848 MHz insgesamt eine höhere Datentransferrate von $177,4 GBytes/s$. Die Speichertransfers werden von einem Zwei-Level-Cache-System unterstützt, das Zugriffszeiten zum 1,5 GByte großen Arbeitsspeicher reduziert. Der 786 kByte große Level-Zwei-Cache (L2-Cache) befindet sich neben den Speicherkontrollern auf der Chip-Ebene, während der L1-Cache sich in den SMs befindet. Frühere GPU-Architekturen besaßen kein Cache-System bzw. nur eines für Leseoperationen. In der GF100-Architektur ist das Cache-System für Lese- und Schreiboperationen ausgelegt und ein Protokoll hält die drei Stufen (L1, L2 und Speicher) kohärent.

Die Stream-Multiprozessoren (SM) haben eine zentrale Bedeutung in der Architektur. Auf ihr werden die Softwareprozesse, die aus vielen Threads bestehen können, ausgeführt. Die Bestandteile eines SMs sind in Bild 2.2 gezeigt.

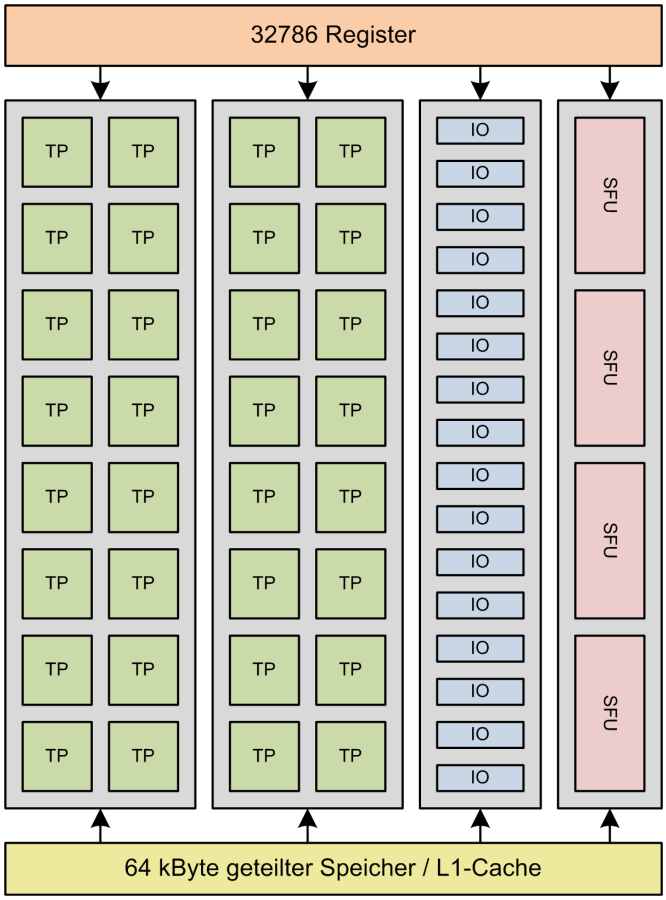


Bild 2.2.: Blockschaltbild eines 16 Stream-Multiprozessors aus der Fermi-GPU-Architektur von NVIDIA [62]

Threadprozessoren (TP). TPs, auch als CUDA-Cores bezeichnet, bestehen aus einer arithmetischen Einheit (ALU) und einer Fließkomma-Einheit (FPU), die in mehrere Pipelinestufen zerlegt sind. Sie ist die kleinste Recheneinheit, die Instruktionen und Operationen ausführen kann. Die Ausführung einzelner Operationen unterliegt allerdings Regeln, denn eine Instruktion wird parallel auf mehreren TPs nach dem SIMD-Prinzip ausgeführt.

Warp-Scheduler. Ein *Scheduler* ist eine Ausführungseinheit, die Instruktionen aus dem Speicher ließt, dekodiert und auf die TPs verteilt. Diese Verteilung passiert in Hardware, was üblicherweise ein Betriebssystem in Software erledigt. Ein *Warp* ist eine Gruppe von 32 Threads (Ausführungsfäden). D.h. dass ein Warp-Scheduler eine Instruktion auf 32 TPs parallel mit unterschiedlichen Daten ausführt. In der Hardware werden allerdings lediglich 16 TPs, das entspricht einem *half Warp*, beauftragt, mit der gleichen Instruktion, zu rechnen. Erst zu einem späteren Zeitpunkt wird die zweite Hälfte des *half Warps* ausgeführt. Damit zu jedem Zeitpunkt

alle 32 TPs einer SM beschäftigt sind, gibt es zwei Warp-Scheduler, die zu einem Zeitpunkt zwei *half Warps* unterschiedlicher *Warps* ausführen.

Lade- und Speichereinheiten (IO). Jede SM besitzt 16 Lade- und Speichereinheiten, die pro Takt ein Ziel oder eine Quelle im Speicher adressieren können und einen Speichertransfer initiieren. Der Speichertransfer richtet sich als erstes an den L1-Cache bevor die Daten vom oder zum Massenspeicher transferiert werden.

Spezielle Funktionseinheiten (SFU). In ihnen werden komplexere Instruktionen abgearbeitet, die die TPs nicht unterstützen. Beispiel für die Instruktionen sind: $sin(x)$ oder $exp(x)$.

Speicher und Register. Jeder TP hat unterschiedliche Speicherbereiche zur Verfügung, Daten abzulegen. Am schnellsten sind die 32786 Register mit je 4 Byte Speicher, die auf alle TPs aufgeteilt werden. Der geteilte Speicher (*Shared*-Speicher) ist für den Datenaustausch zwischen unterschiedlichen TPs gedacht. Es existiert ein 64 kBytes *Shared*-Speicher, der gleichzeitig als L1-Cache verwendet wird. Weitere Details verschiedener Speicher sind im Abschnitt CUDA erläutert.

Die theoretisch maximale Rechenleistung berechnet sich durch 480 $Thread prozessoren * 1,4$ $GHz * 2$ $Operationen = 1344$ $GFLOPS$ mit einfacher Genauigkeit (SP, *single precision*). Die TPs sind mit $1,4GHz$ getaktet und mit der MAC-Operation können sie gleichzeitig zwei Operationen, die Multiplikation und die Addition, ausführen. Wenn in doppelter Genauigkeit (DP, double precision) gerechnet wird, ist die Rechenleistung um den Faktor acht kleiner.

2.1.3. Programmiersprache CUDA

Die Compute Unified Device Architecture (CUDA) ist eine Programmiersprache, die auf C aufsetzt. Sie ermöglicht das Programmieren von NVIDIAs Grafikprozessoren mit nicht grafischen Anwendungen. CUDA besteht aus zwei Teilen, den Kernel-Funktionen, die parallel tausende Threads auf der GPU ausführen können, und der Laufzeitumgebung, einem Funktionsumfang, der Speichertransfers und Kernelaufrufe vom Host aus regelt, um die GPU zu steuern.

In CUDA kann der Programmierer einen Algorithmus parallelisieren und als Quelltext in Kernel-Funktionen abbilden. Die Kernel-Funktionen werden auf der GPU parallel ausgeführt. Die Skalierung der Parallelität, gemeint ist die Anzahl der Threads, die die Kernel-Funktionen durchlaufen sollen, wird beim Funktionsaufruf mit der Angabe einer Dimension für ein Grid und CUDA-Blöcken definiert, siehe Quelltext-Beispiel 2.1 auf Seite 13. Das heißt, innerhalb ei-

nes Grids werden CUDA-Blöcke bestimmter Anzahl geschaffen, die wiederum eine bestimmte Anzahl an Threads beinhalten. Bild 2.3 veranschaulicht den Zusammenhang zwischen Grid, Blöcken und Threads mit der Möglichkeit einer mehrdimensionalen Anordnung. Jeder Thread innerhalb der Blöcke führt die gleiche Kernelfunktion für unterschiedliche Daten aus. Ähnlich dem SIMD-Prinzip nennt NVIDIA dieses Prinzip *Single Instruktion Multiple Threads* (SIMT), erwähnt in der CUDA-Programmieranleitung. [65].

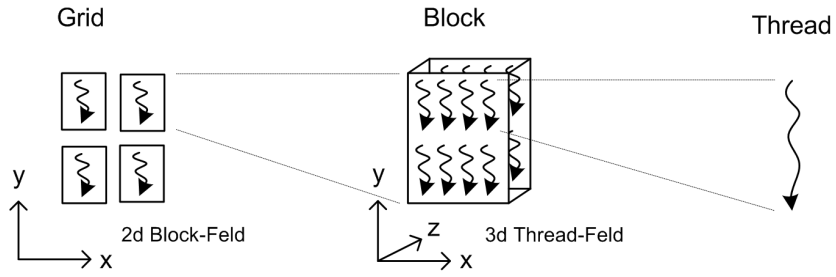


Bild 2.3.: Anordnung vieler Threads in einem Grid und Blöcken. Ein CUDA-Grid besteht aus einem bis zu zweidimensional großen Block-Feld. Ein CUDA-Block kann eine bis zu dreidimensionale Anordnung von Threads sein.

Die Anzahl der Threads, die beim Kernelaufruf bestimmt und angelegt werden, werden von der Hardware dynamisch zur Laufzeit auf die Threadprozessoren verteilt. Genauer, wird jeder CUDA-Block, vergleichbar wie ein Prozess, auf einem Stream-Multiprozessor ausgeführt, durch andere CUDA-Blöcke verdrängt bzw. beendet, wenn alle Thread des Blocks das Ende der Kernelfunktion erreicht haben. Die genaue Zuordnung der CUDA-Blöcke zu den SMs und die Verteilung der *Warps* auf die Threadprozessoren passiert dynamisch zur Laufzeit und ist deswegen nicht vorhersagbar. Die CUDA-Blöcke werden in ihrer Ausführung verdrängt, wenn ihre Threads Speichertransfers initiieren und auf die Daten vom Speicherkontroller warten. Da die Latenzzeiten viele hundert Taktzyklen dauern, lohnt es sich während dessen einen anderen CUDA-Block auszuführen. Sofern ein Vielfaches an Blöcken zu den SMs existiert, können die Latenzzeiten der Speicherzugriffe hinter der Berechnungszeit anderer CUDA-Blöcke vollständig versteckt werden.

Tabelle 2.1 zeigt alle verfügbaren Speicherbereiche, die von den Kernelfunktionen genutzt werden können.

Register. Die Programmierer können auf die Register nicht direkt zugreifen. Der CUDA-Kompilierer verwaltet die Register und strebt an, sie sehr effizient zu nutzen, da sie die geringste Latenzzeit aufweisen.
Lokale Speicher. Falls ein CUDA-Block mehr Register benötigt als vorhanden sind, müssen

	Größe	Latenz-zeiten	Chip integriert	Cached Speicher	GPU Zugriff	Zugriffs-beschränkung
Register	8192	1	Ja	Nein	lesen/schreiben	pro TP
Local	global	1/800	Nein	Ja	lesen/schreiben	pro TP
Shared	16 KB	2	Ja	Nein	lesen/schreiben	pro SM
Global	1,5 GB	1/800	Nein	Ja	lesen/schreiben	Host
Constant	64 KB	1/800	Nein	Ja	nur lesen	Host
Texture	global	1/800	Nein	Ja	nur lesen	Host

Tabelle 2.1.: Unterschiedliche Speicher der GPU-Architektur mit Eigenschaften.

Daten auf reservierte Bereiche im globalen Speicher ausgelagert werden, der sich lokaler Speicher nennt.

Shared Speicher. Dieser Speicherbereich dient zum Datenaustausch unterschiedlicher Threads innerhalb eines CUDA-Blocks. Eine Threadkommunikation in unterschiedlichen CUDA-Blökken ist nicht möglich, da sie auf unterschiedlichen SMs ausgeführt werden können. Innerhalb des CUDA-Blocks müssen die Speicherzugriffe auf den *Shared*-Speicher synchronisiert werden, da auch die Ausführungsreihenfolge der Threads zufällig ist. Beispiel: Jeder Thread wartet am Synchronisationspunkt auf alle anderen, damit die Threads, die Daten lesen, so lange warten, bis die Threads, die die Daten schreiben, fertig sind. Erst mit einer Synchronisation kann die Gültigkeit der Daten garantiert werden.

Globaler Speicher. Dies ist der Massenspeicher, über den die GPU und der Host Daten austauschen können. Er besitzt die größte Latenzzeit von 400-800 Taktzyklen und ist mit einem zweistufigen Cache versehen. Sind die Daten bereits im Cache vorhanden, kann sich die Latenzzeit bis auf einen Taktzyklus reduzieren, je nachdem ob die Daten im L1- oder L2-Cache liegen.

Konstanter Speicher. Nur der Host kann den konstanten Speicher schreiben. Die Kernelfunktionen können den kleinen Speicherbereich lediglich lesen. Dieser Speicherbereich hat an Bedeutung verloren seit dem der globale Speicher über ein Cachesystem verfügt. Denn dieser besitzt einen separaten Cache, um möglichst immer gute Cachetreffer zu erzielen.

Texturenspeicher. Dieser ist vergleichbar mit dem konstanten Speicher, der über ein separates Cache verfügt. Allerdings kann er wesentlich größer sein, indem Teile des globalen Speichers für den Texturenzugriff reserviert werden.

Wenn die Kernelfunktion viele Ressourcen benötigen, kann die Effizienz leiden. Braucht ein CUDA-Block den gesamten *Shared*-Speicher, kann somit nur dieser auf einer SMs ausgeführt werden, statt möglichen acht. Die Effizienz leidet auch, wenn wenige Threads zu viele Register benötigen; dann können eben auch nur wenige Threads gleichzeitig auf einer SM ausgeführt werden. Die limitierenden Faktoren sind in Formeln einer Excel-Tabelle hinterlegt, die die Aus-

lastung der SMs bzw. die Anzahl parallel arbeitender CUDA-Blöcke berechnet, siehe [66].

Die typische Abfolge eines einfachen CUDA Programms beginnt mit der Kopie der Eingangsdaten in den Grafikkartenspeicher. Beim Kernelaufruf wird die Anzahl an Blöcken und Threads definiert und dimensioniert, je nachdem wie häufig es gewünscht ist, das Kernelprogramm parallel auszuführen. Nach der Berechnung werden die Ergebnisse vom Grafikkartenspeicher in den Hauptspeicher des PCs kopiert. Bild 2.1 zeigt den Quelltext einer Kernelfunktion, die einer Matrixaddition berechnet und somit eine zweidimensionale Threadanordnung des Grids und der Blöcke nutzt.

```
1  __global__ void CUDA_matrixAdd(float* A, float* B, float* C)
2  {
3      // Anzahl Threads in einem Block
4      int blockSize  = blockDim.y  * blockDim.x;
5      // aktuell ausgeführte Blocknummer
6      int blockNum   = blockIdx.y  * gridDim.x  + blockIdx.x;
7      // aktuell ausgeführte Threadnummer im Block
8      int threadNum  = threadIdx.y * blockDim.x + threadIdx.x;
9      // globale Threadnummer von allen Blöcken
10     int globalNum  = blockNum * blockSize  + threadNum;
11
12     // Jeder Thread adressiert zwei Speicherzellen im
13     // Hauptspeicher, addiert und schreibt sie zurueck.
14     C[globalNum] = A[globalNum] + B[globalNum];
15 }
16
17 int main()
18 {
19   ...
20   // Kernelaufruf mit ca. 50000 Threads
21   CUDA_matrixAdd<<<256, 192>>>(A, B, C);
22   ...
23 }
```

Quelltext 2.1: Von vielen Threads durchlaufene CUDA-Kernelfunktion zur Berechnung einer Matrixaddition

Die in der Kernelfunktion eingebauten Variablen `gridDim`, `blockDim`, `blockIdx` und `threadIdx` existieren ohne Deklaration und sind Strukturen mit jeweils drei Komponenten `x`, `y` und `z`. Sie zeigen an, welcher Thread innerhalb welchen Blocks gerade in der Ausführung ist. Im Quelltextbeispiel wird in lokalen Variablen die Blockgröße, die Blocknummer im Grid und die Threadnummer im Block berechnet. Die Blocknummer und Threadnummer berechnen sich analog zu der Indizierung eines Matrixelements im linearen Speicher; $Element = Index\ i * Zeilenlaenge + Index\ j$. Die globale Threadnummer kombiniert die Threadnummer im Block und die Blocknummer zur Eindeutigkeit. Die globale Threadnummer wird für die Adressierung des Hauptspeicher benutzt.

CUDA gibt dem Programmierer die Freiheit, die Threads und die Speicher ohne Einschränkungen zu verwenden zu können. Beispielsweise funktioniert es aber nicht effizient, wenn einige Threads unterschiedliche Wege im Programm durchlaufen, obwohl die Hardware 32 Threads in einem *Warp* gemeinsam ausführen muss. Es gibt viele Regeln, die zu beachten sind, um effiziente Kernel-Funktionen zu entwickeln, die im Benutzerhandbuch der vorbildlichen Praktiken [64] beschrieben sind. Hier ein Auszug der wichtigsten Praktiken:

- Lineare Speicherzugriffsmuster sind am effizientesten, da der Speicherkontroller keinen schnelleren Betrieb hat als den Burst-Modus. Soll heißen, dass, egal wie das Speicherabbild aussieht (Feld, Matrix oder Volumen), die Threaddimensionierung dem angepasst werden muss. Dann kann ein Thread ohne komplexere Adressberechnung „sein“ Element aus dem Speicher bearbeiten. Als Beispiel wieder die Kernelfunktion vom Quelltextabschnitt 2.1, in diesem Fall existieren genau so viele Threads wie Matrixelemente, die in der gleichen Struktur angelegt sind. Bei einem Speicherzugriff auf das nullte Element werden automatisch die Elemente 0 bis 31 gelesen, somit werden mit einem Speicherzugriff die Threads 0 bis 31 des gleichen *Warps* bedient. Im Benutzerhandbuch wird dies als *coalescing* bezeichnet.

- Eine weitere Praktik, auf die geachtet werden muss, betrifft wieder die Speichertransfers. Bei der Adressierung einer Matrix oder eines Volumens muss jede Basisadresse einer Zeile ein vielfaches der *Warp*-Größe sein. Möchte man die Elemente 33,34,.. aus dem Speicher lesen, werden aber die Elemente 31,32,33,34,.. angefordert (vielfaches der *Warp*-Größe). In diesem Fall sind die Elemente 31 und 32 unnötigerweise gelesen worden. Verhindert wird dies, indem jede Zeile in der Länge erweitert wird, damit die nächste Zeile eine gültige Basisadresse besitzt. Dabei kann am Ende jeder Zeile ein Datenbereich entstehen, der nicht verwendet wird. Im Benutzerhandbuch mit dem Begriff *pitch* zu finden.

- Der *Shared*-Speicher ist in Bänke unterteilt, die unterschiedliche Speichermodule repräsentieren. Wenn viele Threads auf den *Shared*-Speicher gleichzeitig zugreifen wollen, muss garantiert werden, dass jeder Thread eine unterschiedliche Bank adressiert. Im Fall eines Bank-Konflikts, wenn mehrere Threads auf die selbe Bank zugreifen, werden die Transfers serialisiert. Dies gilt es zu vermeiden. Siehe im Benutzerhandbuch unter dem Begriff *bank conflict*.

- Wie oben bereits beschrieben, sollten alle Threads eines Warps dem selben Ausführungspfad folgen. Da eine Instruktion auf alle Threads ausgeführt wird, müssen im Fall unterschiedlicher Ausführungspfade alle durchlaufen werden. D.h. der Quelltext aller Zweige

muss ausgeführt werden. Auch dies gilt es zu vermeiden, indem die Daten zusammengetragen werden, die von 32 Threads gleich behandelt werden können. Unter dem Begriff *divergent branch* lässt sich mehr dazu im Benutzerhandbuch finden.

Für weitere Praktiken der Optimierung sei auf das Benutzerhandbuch verwiesen.

2.1.4. Programmiersprache OpenCL

OpenCL ist CUDA recht ähnlich, mit dem Unterschied, dass OpenCL keine Begrenzung der eingesetzten Geräte-Plattform hat. Genau darin liegt der große Vorteil: in OpenCL geschriebene Kernelfunktionen laufen funktionell auf allen GPUs, Prozessoren und eingebetteter Hardware, für die es eine OpenCL-Implementierung gibt, ohne den Kernel-Quelltext ändern zu müssen. Genau wie CUDA verfügt OpenCL über eine Speicherhierarchie und ein Ausführungsmodell, welches viele Threads in Gruppen aufteilt, die über gemeinsamen Speicher kommunizieren können und auf der Hardware ausgeführt werden.

CUDA-Begriff	**OpenCL-Begriff**	**deutsche Bedeutung**
	Plattformmodell (Hardwarekomponenten)	
device	*compute device*	Die Grafikkarte bzw. das Rechen-Gerät
host	*host*	Computer, in dem das Rechengerät verbaut ist
Stream-Multiprozessor	*compute unit* (CU)	Skalierbarer Parallel-Prozessor mit vielen Rechenkernen
Threadprozessor	*processing element* (PE)	Rechenkern als Teil einer größeren Einheit
	Speichermodell (Speichertypen)	
host memory	*host memory*	Hauptspeicher des Computers in dem das Rechen-Gerät ist
global memory	*global memory*	Hauptspeicher des Rechen-Geräts, verbunden mit der CU
shared memory	*local memory*	Geteilten-Speicher auf den alle PEs einen CU-Zugriff haben
local memory	*private memory*	Lokal-Speicher mit exklusivem Zugriff für ein PE
	Ausführungsmodell (Ausführung aller Threads)	
CUDA-Block	*work-group*	Rechen-Gruppe, die parallel auf einer CU ausgeführt wird
CUDA-Thread	*work-item*	Ausführungsfaden als Untermenge einer Rechen-Gruppe

Tabelle 2.2.: Fachbegriffe aus CUDA und OpenCL, die das selbe meinen.

Leider existieren unterschiedliche Terminologien in OpenCL zu CUDA, die an einer Stelle zu Verwechslung führen. Die Tabelle 2.2 listet die deutsche Bedeutung mit englischen CUDA- und OpenCL-Fachbegriffen in Bezug auf das Plattform-, Speicher- und Ausführungs-Modell auf.

In OpenCL existieren in den Kernelfunktionen keine eingebauten Variablen, die den aktuell arbeitenden Thread identifizieren, sondern Funktionen, die die *work-group*-Nummer bzw. die *work-item*-Nummer auslesen. In OpenCL existiert gegenüber CUDA die Funktion `size_t`

`get_global_id(int D)`, die die Berechnung der globalen Thread-Nummer übernimmt, wie sie im Quelltextbeispiel 2.1 auf Seite 13 berechnet wurde. Quelltextbeispiel 2.2 demonstriert die gleiche Funktion, diesmal als OpenCL-Kernel.

```
1 __kernel void CL_matrixAdd(__global float* A, __global float* B, __global float* C)
2 {
3   int row     = get_global_id(1);
4   int col     = get_global_id(0);
5   int colSize = get_global_size(0);
6   int element = row * colSize + col;
7
8   C[element]  = A[element] + B[element];
9 }
```

Quelltext 2.2: Von vielen Threads durchlaufene OpenCL-Kernelfunktion zur Berechnung einer Matrixaddition

Wie in CUDA gibt es in OpenCL einen Satz Funktionen (Laufzeitumgebung), die Kernel-Funktionen starten und Daten von und zum Geräte-Speicher transferieren. Die Laufzeitumgebung von OpenCL umfasst weitere Funktionen zum Anlegen und Auswählen eines OpenCL-Geräts sowie die Übersetzung einer Kernelfunktion zur Laufzeit. Der OpenCL-Kompilierer ist kein eigenständiges Programm sondern wird durch Bibliotheksfunktionen der Laufzeitumgebung aufgerufen und gestartet. D.h. der OpenCL-Programmierer muss erst ein Programm schreiben, um seine OpenCL-Kernelfunktion übersetzen zu können. Der volle Funktionsumfang der Laufzeitumgebung ist in der OpenCL-Spezifikation [36] beschrieben.

Das Bild 2.4 zeigt die OpenCL-Strukturen, die notwendig sind, um ein Programm zu übersetzen und zur Ausführung zu bringen. Die `cl`-Funktionen generieren die Strukturen bzw. benötigen sie für ihre Ausführung. Ein Beispiel einer OpenCL-Anwendung, wie die Funktionen der Laufzeitumgebung verwendet werden, befindet sich im Anhang A.

2.2. Rekonfigurierbare Hardware

2.2.1. Aufbau eines FPGAs

Die Abkürzung FPGA bedeutet *field programmable gate array*, frei übersetzt heißt das „feldprogrammierbarer Logikschaltkreis“. Das Bild 2.5 zeigt den schematischen Aufbau eines FPGAs. Er besteht aus vielen einzelnen Logikzellen (LZ), Blockspeichern (Block-RAM), Multiplizierern, Ein- und Ausgabeblöcken (IOB) und einem programmierbaren Verdrahtungsnetzwerk. Wie ein FPGA aufgebaut wird, ist im Nachschlagewerk "Das FPGA-Kochbuch"[78] gut erklärt.

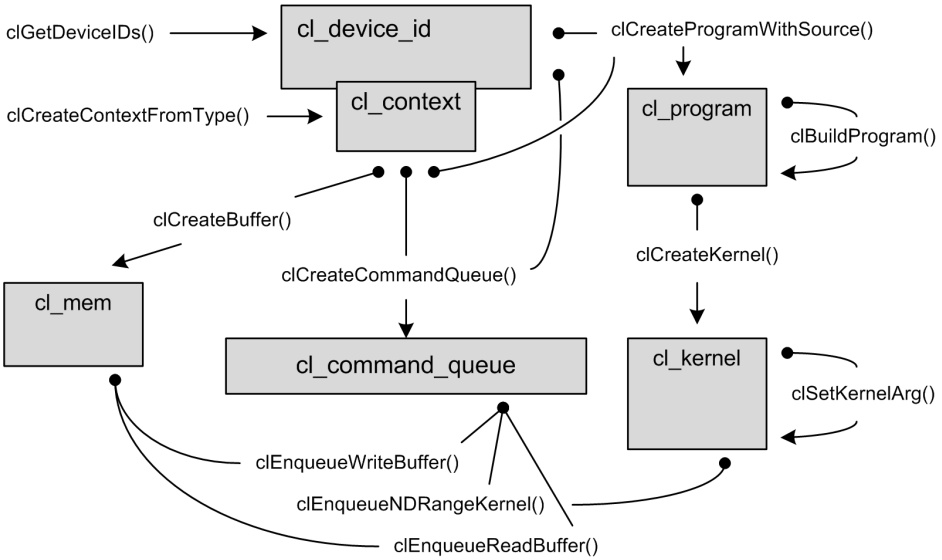


Bild 2.4.: Verwendete Strukturen und Funktionen eines minimalen OpenCL Programms, einschließlich einer Übersetzung mit Kernelaufruf. Pfeile sind als generierendes Ergebnis zu interpretieren. Punkte als Abhängigkeit.

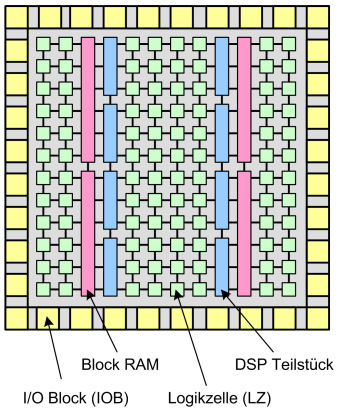


Bild 2.5.: Bestandteile des FPGAs

- Die Logikzellen sind einzelne Elemente, die durch Programmierung unterschiedliche Logikfunktionen beherrschen (CLB = *configureable logic block*). Meist bestehen sie aus einer LUT (*look-up table*), was nichts anderes als ein SRAM-Speicher ist. Der CLB kann als Speicher oder als Logikelement benutzt werden, indem über die Adressierung im Speicher Ausgangswerte einer Wertetabelle abgelegt werden. So kann ein Logikgatter imitiert werden, das die Addressleitungen als Eingänge und das Datenwort als Ausgänge verwendet.

- Der Blockspeicher (Block-RAM) ist ein fest auf dem FPGA vorgesehener Speicher, um eventuell die kostbarere Ressource der Logikzellen einzusparen. Die Block-RAMs besitzen zudem zwei Ports, die mit unterschiedlichen Taktraten angesprochen werden können.
- Wegen der Leistung werden DSP Teilstücke (DSP slices) auf dem FPGA integriert, die Aufgaben aus dem Bereich der **d**igitalen **S**ignal**p**rozessoren übernehmen können. Sie bestehen aus Hardware-Multiplizierer und -Addierer, die mit einer geringeren Verzögerung und einer dennoch hohen Taktrate arbeiten. Die Verwendung der Addierer oder Multipizierer aus den DSP slices spart viele CLBs ein, aus denen die Arithmetik sonst zusammengesetzt werden müsste.
- Die Ein- und Ausgabeblöcke (IOB) sind Puffer, die mit den physikalischen Pins verbunden sind.
- Das Verdrahtungsnetzwerk basiert auf SRAM-Technik, was bedeutet, dass durch Beschreiben von Speicherzellen die CLBs verdrahtet werden. Das heißt auch, dass der FPGA beim Spannungsverlust sein Logik-Design verliert.

Wie viel Logik in ein FPGA hineinpasst, oder wie viel Chipressourcen ein Logik-Design benötigt, hängt in erster Linie von der Anzahl der CLBs eines FPGAs ab. Da aber die Fähigkeit von CLBs verschiedener Hersteller variieren kann, wird für die Komplexität der Logikzelle bzw. des gesamten FPGAs ein Gatteräquivalent (Gates) verwendet. Ein Gatter ist definiert als ein Logikelement mit zwei Eingängen und einem Ausgang, wie beispielsweise bei einem NAND. Selbst durch die Angabe, aus wie vielen Gates ein CLB/FPGA besteht, sind die FPGAs unterschiedlicher Hersteller dennoch nicht direkt in Leistung und Ressourcenverbrauch vergleichbar. Es gibt viele verschiedene Maßeinheiten für den Ressourcenverbrauch eines Designs bzw. für die Kapazität eines FPGAs. Beispiele sind: „CLBs“, „LC“ (logic cells), „slices“, „gates“ und viele andere.

2.2.2. Beschreibungssprache VHDL

Um FPGAs zu programmieren, wird eine Hardware-Beschreibungssprache verwendet. Die bekanntesten sind VHDL (***V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage*) und Verilog. VHDL ist die erste Wahl in dieser Arbeit, weil der Autor die strenge Typprüfung bei der Komponenten-Instantiierung schätzt. Beide Sprachen sind hohe Programmiersprachen, die es erleichtern, Hardware zu synthetisieren. Die Synthese beschreibt, wie der VHDL-Quelltext im Hardware-Logik-Schaltplan bzw. in Netzlisten umgesetzt wird. Wie das passiert, wird in [46] nachvollziehbar gezeigt. Ein sehr beliebtes Nachschlagewerk der VHDL-Grammatik ist die "VHDL Quick Reference Card"[26].

Die Sprache VHDL wird für zwei Zwecke verwendet:

- Erstens für die Beschreibung von Logikschaltungen, dann spricht man von synthesefähigem Quelltext.
- Zweitens für die Beschreibung eines Simulationsmodells, entsprechend nicht-synthesefähigem Quelltext.

Die Unterschiede liegen in Anweisungen die eine zeitliche Abfolge definieren. Beispielsweise welche Latenzzeiten eine kombinatorische Logik hat, wird von den tatsächlich existierenden Logikgattern bestimmt, so dass die Angabe einer Zeit nicht synthesefähig ist.

2.2.3. Partitionelle Rekonfiguration

Die Übersetzung der Beschreibung in Hardware passiert in mehreren Schritten. Da in dieser Arbeit Xilinx-FPGAs verwendet werden, wird die Übersetzungskette anhand der Xilinx-Entwicklungswerkzeuge aus dem „System-Entwicklungs-Nachschlagewerk“ (*Development System Reference Guide*) [79] vereinfacht demonstriert.

1. **XST** ist das Synthesewerkzeug, das, wie bereits beschrieben, die Hardwarebeschreibung in eine Netzliste übersetzt.
2. **NGDBuild** generiert aus der Netzliste eine Datenbank (**N**ative **G**eneric **D**atabase), die den Logikschaltplan auf einfache Logikelemente wie UND-Gatter, ODER-Gatter, Dekodierer, Flip-Flops und RAMs herunter bricht und auflistet. Neben dem Logikschaltplan enthält die NGD-Datei auch die Logikbeschreibung.
3. **MAP** bildet die einfachen Logikelemente auf die Logikzellen eines speziellen FPGAs ab, so dass die CLBs möglichst effizient genutzt werden. Das Werkzeug liefert eine NCD-Datei (**N**ative **C**ircuit **D**escription).
4. **PAR** (**P**lace **A**nd **R**oute) platziert und verdrahtet die vorkonfigurierten CLBs im FPGA. Wieder liefert das Werkzeug eine NCD-Datei, die diesmal die benötigten Chip-Ressourcen nicht nur listet, sondern auch mit Ortsinformationen und Verbindungswegen versieht.
5. **BitGen** bringt die platzierten und verdrahteten CLBs in ein Binärformat, das fähig ist, den FPGA über die Konfigurationsschnittstelle zu programmieren. Das Ergebnis der Übersetzung ist eine BIT-Datei (**Bit**stream file) mit der ursprünglichen Hardwarebeschreibung.

Wird auch nur ein Logikgatter in der Hardwarebeschreibung verändert, muss der gesamte Übersetzungsprozess neu angestoßen werden. Dieser Nachteil lässt sich mit einer Partitionierung

eingrenzen. Die Übersetzungsschritte 1-4 können für jede Partition unabhängig geschehen. Die Änderung einer Partition, mit dem Durchlauf der Übersetzungskette, nutzt die bereits übersetzten anderen Partitionen für die Platzierung und Verdrahtung des gesamten FPGAs. In der Regel spart dieser Vorgang Zeit und bei einer Übersetzung vieler Partitionen können die Xilinx-Werkzeuge parallel auf Mehrkernprozessoren arbeiten.

Eine dynamische partielle Rekonfiguration (DPR), gemeint ist eine teilweise Neuprogrammierung eines FPGAs, ist der Austausch einer Partition mit einer anderen während des Betriebs. Voraussetzungen für das Funktionieren sind im „DPR Benutzerhandbuch“ [80] ausführlich erläutert und hier zusammengefasst:

- Die Schnittstellen der beiden Partitionen müssen gleich sein.
- Der Austausch gelingt nur zwischen dynamischen Partitionen (dynamisches Design) und statischer Logik (statisches Design).
- Es müssen auf jeder Seite, statisches und dynamisches Design, vor der Schnittstelle Registerstufen sein. Diese Anforderung existiert, weil die Xilinx-Tools über diese Grenze das Zeitverhalten nicht optimieren können.
- Vor dem Austausch muss die Grenze geschlossen werden. Das bedeutet, alle Signale vom dynamischen zum statischen Design müssen logisch entkoppelt werden. Während der Programmierung des dynamischen Designs können dessen Ausgänge undefinierte Werte annehmen.
- Das gesamte Design benötigt eine 10% höhere maximale Taktfrequenz, wenn DPR verwendet wird.

2.3. Kompiliererentwicklung

2.3.1. Frontend

Das Frontend ist der erste Teil eines Kompilierungsprozesses, der untersuchende Aufgaben hat. Dabei wird der Quelltext in ein Speicherabbild überführt und die Funktion überprüft. Diese Aufgaben werden von den drei Teilen erledigt.

Zeilenrekonstruktion. Als vorprozessierenden Schritt werden Leerzeichen, Tabulator-Zeichen, Zeilenende-Zeichen und Kommentare gefiltert, da sie für die Übersetzung in den meisten Sprachen keine Relevanz haben.

Lexikalische Analyse. Als erstes muss der Quelltext in Tokens zerlegt werden. Tokens sind grammatikalische Bausteine (z.B. Schlüsselworte, Sprachsymbole, Bezeichner, Operatoren, usw.).

Syntaktische Analyse. Im zweiten Schritt werden die Tokens den Grammatikregeln der Sprache unterzogen und geprüft, ob deren Reihenfolge zulässig ist. Dabei wird ein Syntaxbaum (AST, *abstract syntax tree*) im Speicher konstruiert. Gibt es einen Regelverstoß, bricht die syntaktische Analyse, auch Parser genannt, mit einer Syntax-Fehlermeldung ab.

Semantische Analyse. Zuletzt wird inhaltlich nach der Bedeutung und dem Sinn geprüft. Mit Hilfe von Attributen wird der Syntaxbaum mit zusätzlichen Informationen angereichert. Variablen und parametrisierte Funktionsreferenzen werden in einer Objektabelle aufgelistet und mit jeder Zuweisung einer Typprüfung unterzogen. Widerspricht das Programm der Sprachlogik, bricht der Kompilierprozess mit einer Semantik-Fehlermeldung ab.

2.3.2. Backend

Das Backend ist der zweite Teil eines Kompilierungsprozesses, der folgende Aufgaben umfasst:

Generierung einer Zwischensprache. Der AST wird in diesem Schritt in eine Zwischensprache übersetzt, die der Zielsprache ähnelt.

Optimierung der Zwischensprache. Wegen der standardisierten Zwischensprache gibt es viele Optimierer, die den Quelltext im Umfang reduzieren. Z.B. Zwischenergebnisse werden in Registern gehalten, ohne sie neu zu berechnen, oder unerreichbarer Quelltext wird entfernt.

Generierung der Zielsprache. Erst jetzt wird aus der Zwischensprache die Zielsprache generiert. Diesen Teil nennt man Assembler. Dank des bereits optimalen Quelltextes in der Zwischensprache lässt sich der Assembler für andere Computerarchitekturen austauschen. Der Assembler modifiziert den Quelltext für dessen Architektur. Beispielsweise wird die Registeranzahl und die Registerbreite in der Zielsprache angepasst.

Das Frontend und das Backend sind sehr ausführlich in „Compiler Design in C“ [41] beschrieben.

3. Stand der Technik

3.1. Co-Prozessoren

3.1.1. Beschleunigung

Co-Prozessoren werden eingesetzt, um die CPU zu entlasten und somit die gesamte Ausführungszeit eines Programms zu reduzieren. Dabei wird der rechenintensive Teil bzw. der Teil, der besonders viel Ausführungszeit beansprucht, vom parallel arbeitenden Co-Prozessor übernommen. Die gesamte Ausführungszeit eines Programms lässt sich in zwei Teile zerlegen: einen, der nicht beschleunigt wird (t_s), und einen, der beschleunigt bzw. parallelisiert wird (t_p). Welche Beschleunigung durch den Einsatz von Co-Prozessoren zu erwarten ist, lässt sich mit der Formel von Amdahl 3.1 berechnen. Der Parameter n steht für die mögliche Parallelisierung, d.h. auf wie viele Prozessoren die Rechenzeit aufgeteilt wird. Die Formel hat den Ursprung in Amdahls Veröffentlichung von 1967 [20]. Besser und mit vielen Beispielen erläutert ist sie in [40] „Computer Architecture“.

$$Beschleunigung = \frac{1}{t_s + \frac{t_p}{n}} \tag{3.1}$$

Mit genauer Betrachtung der Formel wird klar, dass für eine große Beschleunigung die Ausführungszeit des seriellen Programmteils deutlich kleiner sein muss, als die des parallelen Programmteils. Die theoretisch maximale Beschleunigung konvergiert schnell gegen einen maximalen Wert. Intuitiv kann bei einem 50 prozentigen parallelen Programmteil eine maximale Beschleunigung von zwei erwartet werden, bei 90 Prozent ergibt das einen Faktor von Zehn. Ob sich der Aufwand einer Beschleunigung lohnt, kann somit genau abgewogen werden. Daher ist es wichtig, die genauen Ausführungszeiten der seriellen und der parallelen Programmteile zu kennen. Profiling-Werkzeuge sind in der Lage, ein Profil der Ausführungszeiten aller Funktionen zu ermitteln.

Weiß man, welcher Teil der Anwendung am langsamsten ist, geht es darum, diesen zu beschleunigen. Der erste Ansatz sollte sein, den Algorithmus mathematisch zu untersuchen. Gibt es die Möglichkeit, die Formeln zu vereinfachen, um Rechenoperationen einzusparen oder rechenintensive Operationen (Divisionen, Logarithmus, ...) zu vermeiden, bietet das die größt-

mögliche Beschleunigung. Schneller, als gar nicht Rechnen zu müssen, geht nicht. Wenn der Algorithmus seriell ist, kann man in der Literatur nach einer parallelen Version suchen. Es kann durchaus sein, dass der Algorithmus für die bisher seriell rechnende Welt serialisiert wurde und ursprünglich parallel war. Erst der letzte Schritt ist die Portierung und Anpassung an die Co-Prozessor-Architektur, beispielsweise auf eine GPU oder einen FPGA.

In erster Linie wird durch eine Parallelisierung beschleunigt. Die einfachste Variante ist die Parallelisierung der Anwendung (nach Flynn MISD [28]), indem die Anwendung mehrfach ausgeführt wird. Die Parallelisierung des Datenflusses (SIMD) betrifft die Anwendung selbst, in dem die Operatoren gleichzeitig auf mehreren Werte angesetzt werden (SIMD). Eine Parallelisierung mit einer Pipeline basiert darauf, das ganze Programm in Funktionseinheiten gleicher Länge zu unterteilen. In jedem Takt können die Funktionseinheiten ihr Teilprogramm berechnen und die Ergebnisse an die nächste Einheit weiterreichen. Dieses Prinzip wird bei der Prozessorentwicklung genutzt die Taktrate einer Abarbeitung von Instruktionen zu steigern. Bei einer Parallelisierung eines Programms liegt der Gewinn in der gleichzeitigen Ausführung aller berechnenden Operatoren im Programm zu jedem Takt.

3.1.2. GPGPU

Die ersten Grafikkarten waren in ihrer Funktionalität darauf beschränkt, Text und einzelne farbige Bildpunkte aus dem Grafikspeicher in Videosignale umzusetzen. Getrieben von der Computerspieleindustrie wurde in den 90er Jahren die Funktionen um die Fähigkeit, Linien und Flächen zu zeichnen, erweitert, um den Grafikaufbau zu beschleunigen. Die *Grafics Processing Unit* (GPU) war geboren.

Mit den populären 3D-Spielen bekam die GPU eine Grafikpipeline, die mit mehreren Shader-Einheiten aus einer polygonen 3D-Rastergrafik ein 2D-Bild berechnete. Die Shader (Vertex-Shader, Geometry-Shader und Pixel-Shader) konnten programmiert werden, um visuelle Effekte (Objektverformung, Lichtquellen mit Schatten, Lichtreflexionen und Objekttexturen) der 3D-Rastergrafik hinzuzufügen.

Die zwei gängigsten Programmierschnittstellen (APIs) für die Grafikpipelines sind DirectX [2] von Microsoft und OpenGL [1] von der Khronos Group. Die entsprechenden hohen und C-ähnlichen Shaderprogrammiersprachen sind OpenGL Shading Language (GLSL) [44] und Microsofts High Level Shading Language (HLSL) [6]. Weiter gibt es die Sprache von NVIDIA „C for Grafics“ (Cg) [58], die den Einsatz auf Grafikkarten des Herstellers beschränkt.

Im Jahr 2003 übertraf die Rechenleistung der damaligen GPUs die der CPUs und setzten sich fortan weiter ab. Es war logisch, die GPU ebenso für rechenintensive, nicht grafische Algorithmen zu nutzen. Ein Weg, Algorithmen auf die GPU zu portieren, besteht in der Programmierung der drei Shadereinheiten. Die Schwierigkeiten bestanden darin,

- den Algorithmus an die grafische Pipeline anzupassen. Beispielsweise waren die Datenstrukturen auf grafische Primitive begrenzt.
- nur über die grafischen APIs programmieren zu können. Dies erfordert viel Wissen über die Entwicklung von Grafikanwendungen, was gar nicht das Ziel ist.
- keine Freiheiten zu haben, beliebig in den Grafikkartenhauptspeicher schreiben zu dürfen. Der Datenfluss einer grafischen Pipeline ist gegeben.
- die Rechenleistung der Shadereinheiten nicht optimalen nutzen zu können, weil die Speicherbandbreite nicht ausreicht.

Zwei wesentliche Entwicklungen, die auf den APIs aufsetzen, beseitigten bzw. reduzierten die Schwierigkeiten der allgemeinen GPU Programmierung, besser bekannt als GPGPU-Programmierung (***G**eneral **P**urpose **G**rafics **P**rocessing **U**nit*). Sh und Brook bieten eine einfache Programmierumgebung, ohne Kenntnisse für Grafikprogrammierung haben zu müssen und ohne Einschränkungen in den Datenstrukturen. Ein Backend übersetzt den Quelltext in die Shadersprachen. Sh [59] ist eine Metaprogrammiersprache, die 2009 von RapidMind abgelöst wurde. BookGPU [24] stell ein Programmierparadigma vor, das aus Streams, Kernels und Reduktionen besteht.

Mit dem kurzlebigen Projekt „Close to Metal“ [7] Anfang 2007 hatte der Grafikkartenhersteller Ati (später AMD), neben einer API auch eine Assemblerschnittstelle für ihre GPUs bereitgestellt. Das eröffnete ebenfalls die Möglichkeit, Algorithmen auf der GPU zu implementieren. CTM wurde 2008 von dem „Ati Stream SDK“ bzw. dem „AMD FireStream“ abgelöst.

Mitte 2007 brachte NVIDIA eine Grafikkarte auf den Markt, die Vertex-Shader, Geometry-Shader und Pixel-Shader in allgemeine Shadereinheiten vereint. Das hatte für die Grafikanwendungen den Vorteil, die Rechenlast besser auf die Threadprozessoren verteilen zu können. Die Durchlaufrate einer (Grafik-)Pipeline orientiert sich an der langsamsten Einheit, während die anderen Einheiten unbeschäftigt bleiben. Für die GPGPU-Anwendungen hatte die neue Architektur die Flexibilität, ähnlich einer CPU, Rechenoperationen in beliebiger Reihenfolge ausführen zu können. NVIDIA bot zeitgleich CUDA an, womit Programmierer einfach parallele Programme für NVIDIAs GPUs entwickelten konnten.

Mit CUDA konnten bereits viele Algorithmen aus der Forschung beschleunigt werden, siehe [61]. Typische Beschleunigungen gegenüber einer CPU-Version liegen zwischen 3 und 60. Aber auch weitaus höhere Werte (mehrere Hundert) sind in der Literatur zu finden. Das Potenzial, Algorithmen nur noch in einem Bruchteil der Zeit auszuführen, ermöglicht Simulationen mit höherer Auflösung, rechenintensive Algorithmen werden echtzeitfähig bzw. erst dann verwendbar.

Im Dezember 2008 veröffentlichte die Khronos Group die erste OpenCL-Spezifikation [36], eine parallele Sprache, die CUDA ähnlich ist, und die unter anderem für GPUs und weitere Rechen-Architekturen geeignet ist. In Abschnitt 3.4.4.1 werden die Implementierungen der Sprache weiter erläutert.

3.1.3. FPGA

Folgende Liste zeigt Beispiele, in wie vielen Bereichen FPGAs eingesetzt werden:

- Digitale Signalverarbeitung (FFTs, digitale Signalfilter, Kodierer/Decodierer, CRC).
- Bildverarbeitung (Frame-Grabber).
- Netzwerktechnik (Routen von Datenpakteten mit sehr kurzer Latenzzeit).
- *Glue-logic* als Kommunikationsbaustein, der andere digitale Bausteine miteinander verbindet.
- *System on a Chip*, auf dem mehrere funktionelle Einheiten (CPU, RAM-Controller, BUSe, Peripherie-Controller) auf einem Chip vereint werden.
- *Data Acquisition* (DAC), Datenaufnahme, Vorprozessierung und Weiterverteilung von Sensorwerten.
- Prototyping im ASIC-Entwurf, zur kostengünstigen Entwicklung.
- *High Performance Community* (HPC) Gemeinde zur Beschleunigung von Algorithmen und Rechenanwendungen.

FPGAs sind nützliche Prozessoren, die beliebige Aufgaben in der digitalen Elektronik übernehmen können. Lediglich bei den HPC-Anwendungen wird der FPGA üblicherweise in einen PC verbaut, in dem er als Co-Prozessor fungiert. In den anderen Einsatzgebieten sind FPGAs ein preiswerter Ersatz für ASICs (***A**pplication-**S**pecific **I**ntegrated **C**ircuit*). Der FPGA hat im Vergleich zum ASIC eine deutliche niedrigere Taktrate, dafür kann die Schaltung beliebig verändert werden.

Der Übersichtsbericht aus dem Jahr 2006 [76] geht auf den Einsatz von FPGAs in der HPC ein. Zu dieser Zeit wurden FPGAs als Co-Prozessoren immer häufiger eingesetzt, da ihre Logikzellen zahlreicher und komplexer wurden, um eine hohe Beschleunigung gegenüber den damaligen CPU zu erzielen.
FPGAs sind hervorragend für Logik und Integeroperationen geeignet. Aus wenigen Logikzellen lassen sich Addierer, Vergleicher und andere arithmetische Integer-Operatoren synthetisieren. Viele wissenschaftliche Algorithmen benötigen Fließkommaberechnung. Mit einer arithmetischen Analyse, die in jedem Teil der Berechnung die notwendige Bitbreite (Genauigkeit) bestimmt, lassen sich die Fließkommaberechnungen in Festkommaberechnungen umwandeln. Banerjee [21] konvertiert in seiner Arbeit mit Matlab Fließkommaberechnungen in Festkommaberechnung für rekonfigurierbare Hardware. Dies funktioniert so lange die Zahlendynamik gering bleibt.
Ein anderer Weg ist es, Operatoren für die Fließkommaberechnungen auf dem FPGA zu implementieren. Beauchamp [22] diskutiert den hohen Ressourcenverbrauch eines Fließkommaoperators in einem FPGA und verwendet die DSP-Slices, um einen Multiplikation-Addition-Operator zu beschreiben. Lienhart entwickelte in seiner Dissertation [51] eine Bibliothek mit Fließkommaroperatoren, die sich in ihrer Rechengenauigkeit parametrisieren lässt, um Hardware-Ressourcen gezielt zu sparen. Weitere Entwicklungen von Fließkommabibliotheken findet man unter [23] (2002) und [77] (2010) demonstriert an Satellitenaufnahmen mit dem k-Means-Algorithmus. Xilinx bietet heute mit dem Core-Generator ein bequemes Programm an, Operatoren für die Fließkommaberechnungen zu erstellen [84].
Aufbauend auf seiner Bibliothek entwickelte Lienhart einen Pipelinegenerator [50]. Mit einer Sprache, bestehend aus mathematischen Ausdrücken, lassen sich die Operatoren konfigurieren und zu einer Pipeline verschachteln. Bei der Generierung der Pipeline werden automatisch alle nötigen Verzögerungstufen eingesetzt. In der Arbeit von Marcus [55] wurde der Pipelinegenerator mit Formeln eines numerischen Lösungsverfahrens, genannt SPH (***S**moothed **P**article **H**ydrodynamics*) demonstriert, um astronomische Ereignisse mit geglätteter Teilchen-Hydrodynamik zu simulieren.

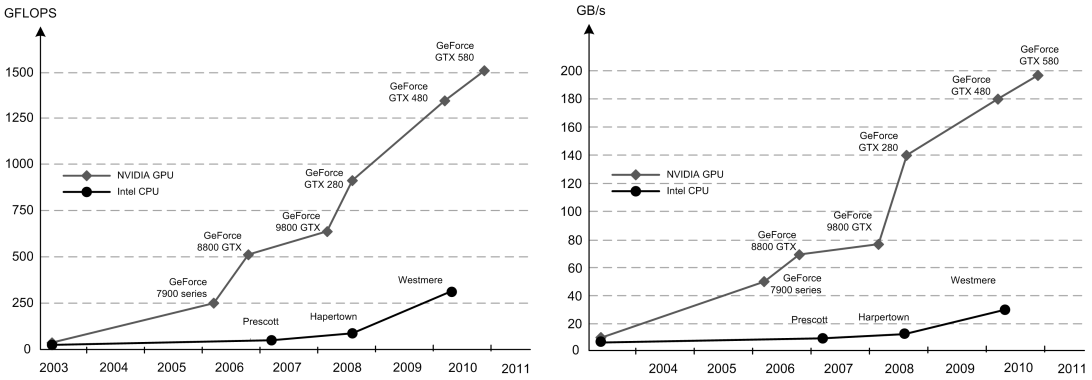
3.1.4. Vergleich

Neben den FPGAs haben sich die GPUs als Co-Prozessoren etabliert. Welcher der geeignete für eine Beschleunigung ist, hängt von der Anwendung ab. Auch der Cell-Prozessor und Programme, die mit den SSI-Befehlen auf der CPU parallelisiert wurden, haben ihren Stellenwert im Beschleunigungssegment. Es folgt eine Untersuchung der CPU, der GPU und des FPGAs mit Blick auf deren Einsatzfähigkeit, Energieeffizienz, Kosten, Rechenleistung und Entwicklungszeit. In den folgenden Abschnitten wird auf die Bewertung eingegangen und bestimmt,

welche Co-Prozessoren für diese Arbeit in Frage kommen.

CPU. Die CPU (***C**entral **P**rocessing **U**nit*) hat einen umfassenden Befehlssatz, mit dem sich Programme entwickeln lassen, die keine Einschränkung in ihrer Funktionalität haben. Eine Cache-Hierarchie reduziert die Zugriffszeiten auf den Hauptspeicher, die viele hundert Prozessortaktzyklen dauern können. Die Taktraten von CPUs um die $3GHz$ steigen kaum noch. Das mooresche Gesetz, eine Verdoppelung der Integrationsdichte bei integrierten Schaltkreisen (ICs) alle 18 bis 24 Monate, wird genutzt, um die Architektur einer CPU in ihrer Leistungsfähigkeit zu steigern. Alle modernen CPUs besitzen eine mehrstufige Pipeline um in praktisch jedem Takt eine Instruktion auszuführen. Eine Weiterentwicklung geht sogar dahin, dass mehrere Instruktionen in einem Takt ausgeführt werden können. Ein Beispiel stellt die *HyperThreading*-Technik von Intel da, die einen virtuellen Kern bietet, um in einem Takt mehrere Instruktionen ausführen zu können. Eine andere Technik, den Grad der parallelen Ausführung zu erhöhen, bieten die *Streaming SIMD Extensions*-Register (SSE). Mit ihnen können einzelne Instruktionen auf mehrere Daten angewendet werden. Seit vielen Jahren steigt die Anzahl der SSE-Register, deren Registerbreite und der SSE-Instruktionen an, um immer mehr Daten nach dem SIMD-Prinzip parallel zu verarbeiten. Die letzte Steigerung der Integrationsdichte ließ zu, mehrere Kerne auf einem Chip zu vereinen. Eine aktuelle CPU von Intel namens Xeon X7560 mit sechs Kernen und 3,2GHz Taktrate bietet $64\,GFLOPS$, das bedeutet $11\,GFLOPS$ pro CPU-Kern, siehe Intel [18]. Die Verlustleistung einer aktuellen CPU liegt bei ca. $90\,W$ mit einem Preis von ca. $500-1000\,EUR$.
Eine Anwendung, die schnell auf der CPU läuft, ist beispielsweise sehr IO-lastig mit geringem Rechenaufwand. Solche Anwendungen lassen sich durch Co-Prozessoren nicht weiter beschleunigen, da die arithmetische Rechenleistung der CPU die Last bewältigen. Eine andere Anwendung würde aus einem seriellen Algorithmus mit geringem Speicherbedarf bestehen. Der kleine Speicherbedarf verspricht eine hohe Trefferrate im CPU-Cache. Parallele Architekturen bieten keine Möglichkeit einer Beschleunigung serieller Algorithmen. Die CPU ist ein ausbalanciertes Rechenwerk für alle Arten von Anwendungen und ist somit das Herzstück aller Berechnungen. Die Co-Prozessoren auf Beschleunigerkarten können der CPU einen spezialisierten Teil der Rechenlast abnehmen, umgekehrt benötigt der Co-Prozessor eine CPU, die ihm eine Aufgabe gibt.

GPU. Die Architektur der GPU wurde bereits im Abschnitt 2.1.2 erläutert. Zusammenfassend bietet eine aktuelle GPU $1500\,GFLOPS$, hat eine Verlustleistung von $300-400\,W$ und einen Preis von ca. $500-1000\,EUR$. Das Bild 3.1 zeigt die Leistung verschiedener Prozessoren von NVIDIA und Intel mit deren Rechenleistung und der Speicherbandbreite zwischen Prozessor und Speicher. Viele Algorithmen werden von der verfügbaren Speicherbandbreite gebremst. Gerade deswegen ist das Wachstum der Speicherbandbreite ebenso wichtig wie die Rechen-



(a) Rechenleistung (b) Speicherbandbreite

Bild 3.1.: Vergleich CPU und GPU der maximal theoretischen a) Rechenleistung und b) Bandbreite [65]

leistung. Die Grafikkarte eignet sich besonders für daten-parallele Anwendungen mit Datenvolumen bis zur Speichergröße der Grafikkarte, bei dem jeder Thread unterschiedliche Daten verarbeitet. Aber auch task-parallele Anwendungen werden von den neueren GPUs unterstützt, indem mehrere Kernelfunktionen zeitgleich gestartet werden. Für eine Echtzeitverabeitung eignet sich die GPU nur bedingt, da die Daten zwischen CPU-Speicher und Grafikkartenspeicher kopiert werden müssen. Ebenso wird die Grafikkarte erst ab einer gewissen Masse an Daten, die blockweise verarbeitet werden, effizient. D.h. es muss gewartet werden, bis ein Block mit Daten gefüllt ist, dieser dann übertragen, berechnet, und wieder zum CPU-Speicher zurück kopiert ist. In CUDA gibt es Pipelinetechniken, gleichzeitig Daten zu übertragen und Kernelfunktionen auszuführen, um die Wartezeiten zu reduzieren. Je nachdem, wie viel Zeit die Echtzeitanforderung erübrigt, können GPUs eingesetzt werden. Die Entwicklungszeiten für CUDA-Programme sind vergleichbar mit denen für die CPU.

FPGA. Im Gegensatz zu den GPUs verarbeiten FPGAs die Daten als Strom von Einzelelemeneten. Vorteile der gestreamten Verarbeitung in einer Pipeline sind, dass

- die Latenzzeiten kurz sind. Sobald Daten bereit liegen, können sie verarbeitet werden. Dies führt zu einer Echtzeitfähigkeit, da die Ergebnisse in vorhersagbarer Zeit (Latenzzeit der Pipeline plus Übertragungszeit) produziert werden.
- die Daten für die Verarbeitung nicht zwischengespeichert werden müssen, im Vergleich zur blockweisen Verarbeitung. Die Pipelineverarbeitung benötigt keinen bzw. wenig Zwischenspeicher wenn man Register nicht zu den Speichern zählt.
- die Pipeline parallel arbeitet und im FPGA an neue Rechenanforderungen angepasst werden kann.

Wie bereits erwähnt, eignen sich FPGAs besonders gut für Integer-Arithmetik, Logik-Aufgaben und Bit-Operationen. Die ideale Anwendung müsste eben aus einer Pipeline bestehen, mit genau den Operationen, die der FPGA gut kann. Der FPGA erzielt besonders hohe Rechengeschwindigkeit dadurch, dass in jedem Takt alle Operationen der Pipeline parallel ausgeführt werden. Wenn die Abwärme bzw. Verlustleistung ein Problem darstellen, werden FPGAs bevorzugt eingesetzt. Eine Untersuchung [43] von Williams zeigt, dass FPGAs einen besseren Wirkungsgrad, gemeint ist Rechenleistung pro Watt, als GPUs haben. Der FPGA ist bei Integeroperationen um das sechsfache und bei Fließkommaoperationen um das zweifache effizienter im Energieverbrauch für die gleiche Rechenleistung. Ein FPGA benötigt ca. $25\,W$ Leistung, hat aber die Nachsicht bei den Anschaffungskosten von $1000 - 10000\,EUR$. Die Kosten vereinen die Entwicklung oder die Anschaffung einer Steckkarte mit einem FPGA-Chip. Eine Aussage über die GFLOPS zu treffen, ist nicht einfach. Ein Virtex-5 mit 330 Tausend Logikzellen konnte laut [76] 56 GFLOPS erreichen, einschließlich aller arithmetischen Operatoren aus den DSP slices und den vorhandenen Logigblöcken. Inzwischen gibt es Virtex 6 FPGAs und Virtex 7 sind angekündigt, diese laufen mit höheren Taktraten und haben mehr Logikzellen. Für die neuste Generation wird die Rechenleistung auf 100 GFLOPS geschätzt. Es ist uns nicht wichtig, eine genaue Zahl zu berechnen, da es sich lediglich um die theoretisch maximale Rechenleistung handelt. Die Entwicklungszeiten sind sehr lang. Aus diesem Grund gibt es eine Fülle von Sprachen, die die FPGA Beschreibung vereinfachen können, die noch weiter vorgestellt und bewertet werden.

Vergleichstabelle. Zur besseren Übersicht werden die Kenndaten in einer Tabelle dargestellt.

	CPU	**GPU**	**FPGA**
theoretische Rechenleistung	64 GFLOPS	1500 GFLOPS	ca. 100 GFLOPS
maximale Verlustleistung	90 W	300-400 W	25 W
Anschaffungskosten	500-1000 EUR	500-1000 EUR	1000-10000 EUR
Entwicklungszeit	kurz	kurz	sehr lang
Einsatzfähigkeit	sequentiell	blockweise	gestreamt

Tabelle 3.1.: Vergleichstabelle der gängigen Co-Prozessoren mit der CPU.

Die sequentielle Einsatzfähigkeit der CPU wird als Vorteil gewertet, denn sie ist in der Lage seriellen Quelltext sehr effizient auszuführen, was die GPU bzw. der FPGA nicht gut können. Die CPU ist als Universalprozessor ebenfalls in der Lage, parallelen Quelltext auszuführen, eine GPU kann dies aber in der Regel effizienter.

3.2. Haralick Texturen Bildmerkale

3.2.1. Beschleunigende Vorarbeiten

Es gibt zwei wichtige Arbeiten, auf die diese Arbeit aufsetzt. Erstens die biologische Anwendung, die die Haralick Bildmerkmale benötigt [25], und zweitens eine andere, die bereits den Haralick-Algorithmus in rekonfigurierbarer Hardware beschleunigt hat [71].

Die erste Arbeit handelt von der automatischen Analyse von Mikroskopaufnahmen mit hohem Bilddurchsatz. Es wurden geeignete Klassifikations- und Bildmerkmal-Algorithmen studiert und ein entsprechendes Softwarepaket entwickelt. Das Softwarepaket klassifiziert die Zellen in Proteinfunktionen und Phänotypen mit einer Präzision von 83%. Für die automatische Analyse der Mikroskopaufnahmen besteht ein hoher Rechenbedarf in der Größenodnung von Monaten. Das Profiling des Softwarepakets zeigt, dass der Haralik Texturen Merkmal Algorithmus (Haralick Algorithmus) den größten Teil an Rechenzeit konsumiert. Im Quelltext des Algorithmuses wurden bereits Bemühungen unternommen, die Laufzeit mit optimierenden Maßnahmen zu beschleunigen, die in Kapitel 4.1 verbessert und erweitert werden.

Die zweite Arbeit benutzt ein FPGA, auf dem die co-occurrence Matrizen und die Haralick Texturen Bildmerkmale in einem speziell angepassten Hardwaredesign berechnet wurden. Die Berechnungen der Matrizen, deren Normierungen sowie der Bildmerkmale wurden mit Handel-C in einem Design realisiert. Die Ergebnisse zeigen eine Beschleunigung von 4,75 bei der Berechnung der co-occurrence Matrizen und eine Beschleunigung von 7,3 bei der Berechnung der Bildmerkmale gegenüber einer CPU Implementierung.
Bei dieser Entwicklung wurden lediglich sieben Bildmerkmale implementiert, was für das VIROQUANT-Projekt nicht reicht, denn es werden 13 Bildmerkmale für die Auswertung der Bilder benötigt. Der verwendete FPGA hat 20 tausend Slices die mit 85% vom Design verwendet wurden. Neuere FPGA-Hardware bietet 240 tausend Slices, zwölf mal mehr Platz, um die fehlenden sechs Bildmerkmale beschreiben zu können und um weitere Beschleunigung zu erziehlen.
Die zu erwartende Beschleunigung die legt folgenden Überlegungen zugrunde, die mit Faktoren geschätzt werden: Die damalige Vergleichs-CPU (Pentium4 mit 2,4 GHz) ist circa um den Faktor zwei langsamer als eine heutige CPU (Intel i7 mit 3,4 GHz). Die FPGA Designfrequenz hat sich seitdem circa um den Faktor zwei erhöht. Die fehlenden Bildmerkmale würden schätzungsweise 12 Tausend Slices benötigen, da diese aus komplexeren Berechnungen bestehen. Das Design würde dann 30 tausend Slices verwenden, die acht mal in einen heutigen FPGA hinein passen würde. Vorausgesetzt, es lässt sich ein Konzept entwickeln, das diese Parallelität zulässt. Die geschätzte Beschleunigung eines neuen Designs auf aktueller Hardware

verglichen mit einer aktuellen CPU liegt bei circa 50. Der Schätzwert ermittelt sich aus der damaligen Durchschnittsbeschleunigung der Matrizen und der Bildmerkmale $(4,75 + 7,3)/2$ und unterliegt folgenden den Faktoren : 2 (langsamere CPU), $*2$ (höhere Designfrequenz) und $*8$ (mehr FPGA-Ressourcen).
Die Entwicklungszeit, das komplexe Design zu überarbeiten, und die hohen Kosten für die FPGA-Hardware, rechtfertigen die erwartete Beschleunigung nicht. Das Ziel einer Beschleunigung liegt im Bereich von zwei bis drei Größenordnungen. Folgender Abschnitt zeigt einen Ansatz, der den Anforderungen gerecht werden kann.

3.2.2. Fazit für eine Beschleunigung

Diese Arbeit strebt den Ansatz an, mit GPUs den Haralick Algorithmus inklusive aller 13 Bildmerkmale zu beschleunigen. Folgende Gründe sprechen dafür:

- Der Algorithmus enthält überwiegend Fließkommaberechnungen, die auf der GPU einfach zu rechnen sind.
- Die Entwicklungszeit, den Algorithmus auf die GPU zu parallelisieren ist kürzer als die Entwicklungszeit eines FPGA-Designs auf der neusten Hardwaregeneration mit den fehlenden Bildmerkmalen.
- Die GPU übersteigt die theoretische maximale Rechenleistung der Fließkommaberechnungen eines FPGAs.
- Wegen der Offline-Prozessierung ist es nicht wichtig, die Berechnung nahe an die Sensor-Elektronik zu knüpfen. Auch der Energiebedarf der GPU ist im Rechencluster nicht kritisch. Eine höhere Latenzzeit bei der blockweisen Berechnung ist bei der Offline-Prozessierung ebenfalls nicht relevant.

3.3. Kompiliererentwicklung

Die Online-Prozessierung hat die Anforderung, den FPGA-Algorithmus den Bedürfnissen der Biologen entsprechend verändern zu können. Hardwarebeschreibungssprachen, mit denen man den Algorithmus verändern kann, sind für die Biologen zu komplex. Aus diesem Grund muss ihnen eine einfache Sprache zur Verfügen gestellt werden, die sich in ein Hardwaredesign übersetzen lässt. Dieser Abschnitt behandelt die Entwicklung von Kompilierern, um eine geeignete Sprache in ein Hardwaredesign übersetzen zu können.

3.3.1. Übersicht Kompilierer-Baukästen

Die ersten Entwickler von Kompilierern mussten Assembler benutzen, was viele Jahre Entwicklungszeit benötigte. Von da an konnten weitere Kompilierer mit hohen Programmiersprachen schneller entwickelt werden. Auch das war noch immer mühselig, da die Komplexität der Grammatiken, beispielsweise in C, sehr hoch ist. Es wurden Programme entwickelt wie z.B. LEX, die die lexikalische Analyse übernahm und YACC [45], die eine programmierte Grammatik syntaktisch analysiert und einen AST konstruiert. Die Open-Source-Varianten heißen FLEX und BISON. Der nächste Schritt der Entwicklung war es, ein Framework zu programmieren, das hilft, Kompilierer zu entwickeln. Es enthält viele vorgefertigte modulare Strukturelemente eines Kompilierers, zusammengefasst in eine Bibliothek mit Hilfsprogrammen zur Analyse. Omniware [19] bietet ein Framework mit dem Fokus auf die sichere Ausführung von Programmmodulen in mobilen Endgeräten an, ohne die laufende Ausführungsumgebung schädigen zu können. Das Kompilierer-Framework Phoenix [60] ist für die Entwicklung von Kompilierer-Backends gedacht, die auf Microsoft Kompilierer aufsetzen. ROSE [69] ist ein freies Kompilierer-Framework, das speziell für objektorientierte Sprache geeignet ist. LLVM ist derzeit das gängigste und nützlichste Kompilierer-Framework, auf das folgend weiter eingegangen wird.

3.3.2. LLVM

***L**ow **L**evel **V**irtual **M**achine* (LLVM) [48] ist ein Kompilierer-Framework, das eine ständige Optimierung der zu übersetzenden Programme ermöglicht. Ständig bedeutet, zum Zeitpunkt des Kompilierens, des Linkens, zur Ausführungszeit und während der Ausführungspausen. Das Kompilierer-Framework ist modular aufgebaut, so dass ein Austausch des Frontends (Quellsprache) und des Backends (Zielsprache) problemlos möglich ist. LLVM definiert eine allgemeine Zwischensprache namens ***I**ntermediate **R**epresentation* (IR), für die es viele Optimierer gibt. IR ist unabhängig von einer bestimmten Prozessorarchitektur, jedoch einer Assemblersprache recht ähnlich.

```
1 int foo(int a, int b, int c)
2 {
3   return (a * (b - c) * (b - c));
4 }
```

Quelltext 3.1: Beispiel C-Funktion.

Als Beispiel ist die C-Funktion `int foo(...);` in 3.1 gegeben. Die Übersetzung und Optimierung in LLVM-IR zeigt der Quelltextabschnitt 3.2. Das Beispiel ist mit der Demo-Webanwendung der Seite [53] übersetzt worden.

```
1 define i32 @foo(i32 %a, i32 %b, i32 %c) nounwind readnone {
2   %1 = sub nsw i32 %b, %c
3   %2 = mul i32 %1, %a
4   %3 = mul i32 %2, %1
5   ret i32 %3
6 }
```

Quelltext 3.2: Beispiel LLVM-IR.

Die Basisdatentypen in LLVM-IR sind Boolean, Integer mit wählbarer Bitbreite, Floattypen und Zeiger. Die Speicherzugriffe werden ausschließlich mit den `load/store`-Befehlen getätigt. Weiter verfügt LLVM-IR über einen unendlichen Vorrat an Registern, die im Beispiel mit einem vorangestellten `%` durchnummeriert werden. Jedes Register wird nur einmal beschrieben, das der ***S**ingle **S**tatic **A**ssignment*-Form (SSA) aus dieser Arbeit [27] entspricht. Das Beispiel zeigt weiter, dass der Ausdruck `(b-c)` in IR nur einmal berechnet wird, was auf eine erfolgreiche Optimierung schließen lässt.

LLVM bietet fünf Vorteile, die andere Kompilierer-Frameworks nur teilweise haben.

1. Das Kompiliermodell hält die Zwischensprache LLVM-IR zu allen Programmlebenszeiten (Zeitpunkt des Kompilierens, des Linkens, zur Ausführungszeit und während der Ausführungspausen) bei. Dies ermöglicht eine Analyse und Optimierung des Programms in allen Stadien des Programms.

2. Neben der Zielsprachengenerierung zur Laufzeit unterstützt LLVM auch eine rechenintensive Generierung zur Entwicklungszeit.

3. Während der Laufzeit kann ein Laufzeitprofil des Programms erstellt werden, das zur Optimierung herangezogen wird.

4. Das Laufzeitmodell ist transparent. Gemeint ist, dass die LLVM-IR keinem speziellen Objektmodell unterliegt, auch keine spezifische Semantik für Exception verwendet wird und keine bestimmte Laufzeitumgebung benötigt. D.h. LLVM-IR kann für alle möglichen Sprachen kompiliert und verwendet werden.

5. Wegen der Sprachunabhängigkeit ist es LLVM möglich, alle Quelltexte, auch Sprachspezifische, einheitlich mit dem gleichen Kompilierer und Optimierer zu verwenden.

Clang [8] ist ein Open-Source-Projekt, das mehrere Frontends mit dem LLVM-Kompiliererframework entwickelt. Die Front-Ends gehören zu den Sprachen C, C++, Objektive C und Objective C++. Im Vergleich zu GCC haben Programme, die mit einem LLVM-Frontend übersetzt wurden, eine vergleichbare kompakte Programmgröße, besitzen besser optimierten Programmtext, der sich schneller ausführen lässt und mit komplexen hohen Sprachkonstrukten ebenso gut

zurecht kommt, trotz einfacher Quelltextrepräsentation.

3.4. Software-Hardware Kompilierer

Dieser Abschnitt untersucht mehrere Sprachen, welche für die Biologen gedacht sind, um den FPGA-Algorithmus in der Offline-Prozessierung verändern zu können.

3.4.1. Übersicht

Die Designs für FPGAs haben lange Entwicklungszeiten wegen

- der komplexen Beschreibung der algorithmischen Aufgabe in der Register-Transfer-Ebene,
- der langen Simulationszeiten,
- der langen Übersetzungszeiten,
- schwer zu deutender Übersetzungsfehler,
- eines möglichen Ungleichverhaltens zwischen Simulationssynthese und Hardwaresynthese,
- der Schwierigkeit, Fehler schnell aufspüren zu können (lange Debug-Zyklen).

Wegen dieser Gründe existieren sehr viele Software-Hardware Kompilierer, die die Hardwaresynthese mit C-ähnlichen Sprachen beschleunigen und erleichtern. Im Idealfall muss der Programmierer keine Hardwarekenntnisse haben, keine Simulation mehr ausführen und das Programm übersetzt fehlerfrei, mit wenig FPGA-Ressourcen und hoher Designfrequenz.

Mittlerweile gibt es sehr viele C-ähnliche Sprachen, die Entwicklungszeit und die Komplexität der Hardwarebeschreibungssprachen vermeiden oder verringern. Der englische Wikipediaeintrag "Hardware description language"[15] wurde in der Vergangenheit stets aktuell gehalten und beinhaltet eine Liste gängiger Hardwarebeschreibungssprachen mit Kommentaren über deren Herkunft bzw. deren Eigenschaften. Zur Zeit sind 31 Sprachen gelistet, darunter die bekannten Sprachen *Impulse C* von Impuls Accelerated Technologies [5], *JHDL* von der Brigham Young Universität [3] und *SystemC* von ARM, CoWare, CynApps und Synopsys [4]. Über die Liste hinaus gibt es viele weitere Sprachen, die global nicht bekannt wurden wie z.B. *CHDL* von der Universität Mannheim.

Die Sprachen, auf die hier weiter eingegangen wird, sind in serielle, parallele und architekturübergreifende Sprachen für den FPGA gegliedert. Es ist gängig, Hardware-Sprachen auf existierende Software-Sprachen aufzusetzen. Seriell, parallel und architekturübergreifend bezieht sich auf die ursprünglich Sprache.

3.4.2. Serielle C-Sprachen für den FPGA

3.4.2.1. Handel-C

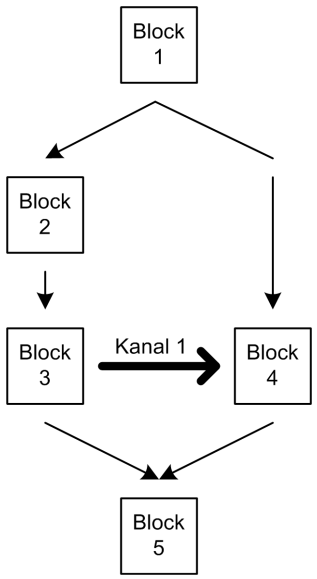
Handel-C wurde 2009 von Mentor Graphics in ihr Synthese-Softwarepaket aufgenommen. Die Sprache ist für hardwarekundige Programmierer entwickelt worden, die eine einfache C-ähnliche Sprache gegenüber den gängigen Hardware-Beschreibungssprachen bevorzugen. Im Benutzerhandbuch [42] sind alle nicht standardisierten C-Spracherweiterungen erläutert, die eine gezielte Hardwaresynthese auf einer höheren Abstraktionsebene ermöglichen.

Die Konzepte von Handel-C basieren auf seriellen und parallelen Programmabläufen und Kanal-Kommunikation. Das Bild 3.2 demonstriert die Konzepte und zeigt Details der Sprache.

```
void main(void)
{
  int 2  a;   int  4 b;
  int 8  c;   int  8 d;
  int 10 e;
  chan ch with { fifolength = 8 };

  b = a * a;          // Block 1
  par
  {
    seq {
      b = b + a;      // Block 2
      c = b * 3;      // Block 3
      ch ! c;         // Kanal 1
    }
    seq {
      ch ? d;
      d = d >> 2;     // Block 4
    }
  }
  e = c + d;  // Block 5
}
```

(a) Handel-C Quelltext



(b) Strukturdiagramm

Bild 3.2.: Handel-C Quelltextbeispiel a) mit Strukturgraph b) so angeordnet, dass die parallelen Abläufe deutlich werden.

- In Handel-C werden Integer-Variablen mit einer frei wählbaren Bitbreite angegeben.

Weiter hat man die Wahl zwischen Vorzeichen-behaftetem und Vorzeichen-losem Integer.

- Die Ausführung wird in serielle Abläufe (`seq`), in b) von oben nach unten, und in parallele (lstinline|par|), in b) auf gleicher Ebene, unterteilt. Im parallelen Segment gibt es zwei serielle Abläufe.
- Jede Zuweisung entspricht einer Taktstufe mit Register. Beispielsweise liegt das Ergebnis nach zwei Takten im Block 2 fest. Der rechte Teil der Zuweisung entspricht einer kombinatorischen Logik und darf mehrere Operatoren enthalten, wobei die maximale Durchlaufzeit die Taktrate mindert.
- Der Kanal 1 entspricht einem FiFo, der zwischen den beiden seriellen Pfaden im parallelem Segment einen Datenaustausch zulässt. Dabei verhält sich das FiFo wie eine Synchronisation, da normalerweise Block 2 gleichzeitig mit Block 4 ausgeführt werden würde. Der `!`-Operator speichert einen Wert im Kanal ab, der `?`-Operator wartet so lange bis ein Wert bereit liegt und liest ihn aus.

Handel-C hat einige Einschränkungen gegenüber ANSI-C, die für eine Hardware-Übersetzung plausibel sind. Es werden keinerlei Fließkommadatentypen oder **`union`**-Datentypen unterstützt, es gibt keine dynamische Speicherverwaltung mit (`malloc` oder `free`) und für die Hardware schwierig realisierbare Rekursionen funktionieren nicht. Es gibt weitere Sprachkonstrukte, die von Handel-C nicht übersetzt werden, beispielsweise darf die **`sizeof`**-Funktion oder die `main`-Funktion keine Parameter und keinen Rückgabewert haben. Weitere ANSI-C Sprachelemente, die in Handel-C nicht konform sind, lassen sich im Benutzerhandbuch nachschlagen.

3.4.2.2. TRIDENT

Trident [74] kompiliert C/C++ Funktionen mit **`float`**- oder **`double`**-Berechnungen in eine Hardwarebeschreibungssprache und bietet die Synthese in ein FPGA-Design. Der Kompilierer gibt dem Programmierer die Wahl, unterschiedliche Fließkommabibliotheken zu verwenden. Der Übersetzungsprozess ist in vier Phasen unterteilt worden, zu sehen in Bild 3.3.

1. **Das LLVM-Frontend** übersetzt die Funktionen mit den Berechnungen in LLVM-IR-Bytecode. Der Bytecode ist eine von Menschen nicht lesbare Repräsentation, auf die dann ausgewählte Optimierer angesetzt werden. Zuletzt wird der Bytecode in Tridents eigene IR abgebildet, bei der nur eine Untermenge an Sprachkonstrukten erlaubt ist. Der Trident Kompilierer setzt dabei auf die Arbeit des SeaCucumber Kompilierers [75] auf.

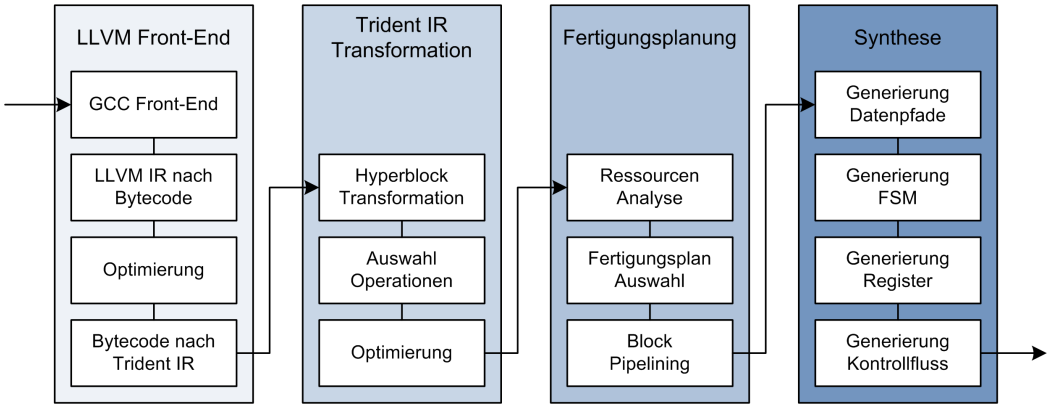


Bild 3.3.: Übersetzungskette des Trident-Kompilierers

2. **Trident IR Transformation**. Die Trident-IR Repräsentation enthält zusätzliche Information, die die Abbildung der Operatoren (Basisblöcke) auf Hyperblöcke erlaubt. Ein Hyperblock [54] ist die Vereinigung mehrerer Basisblöcke, die Kontrollflusssignale nur am Eingang benötigt. Sämtliche bedingte `if`-Verzweigungen werden von einer Kontrollabhängigkeit in eine Dantenabhängikeit überführt. Die Auswahl identifiziert die benötigten Operationen, während die anderen vom Optimierer entfernt werden. Die vollständige Optimierung ist erst nach der `if`-Transformation möglich.

3. **Fertigungsplan**. Die Analyse findet im Trident-IR die Anzahl maximal gleichzeitig genutzter Operationen. Wenn es fünf Addierer gibt, aber nur drei zu beliebigen Zeitpunkten gleichzeitig genutzt werden, benötigt man nun so viele in der Hardware. Unter Berücksichtigung der verfügbaren FPGA-Ressourcen und der verfügbaren Speicherbandbreite wird ein Kontrollflussgraph mit Latenzzeiten erstellt.

4. **Synthese**. Einer der ersten Schritte in der Synthese ist die Auswahl der Fließkommabibliothek, die zugrunde gelegt wird. Die Ausgabe der Synthese ist ein Hardwaredesign in VHDL. Die Synthese umfasst die Generierung der Datenpfade in einer Pipeline, dies passiert für mehrere Blöcke. Zu jedem Block wird eine Zustandsmaschine (FSM) geheftet, die das Zeitverhalten aus dem Kontrollflussgraph berücksichtigt. Die generierten Register dienen der Blockkommunikation untereinander. Ein Kontrollmodul steuert die Ausführung aller Blöcke.

Laut der Autoren funktioniert der Trident Kompilierer für zahlreiche einfache FPGA-Anwendungen. Die Leistung des FPGA-Designs hängt maßgebend von der Fließkommabibliothek ab, was die Entwickler so wollten. Man hat die Wahl, Bibliotheken zu verwenden, die auf Ressour-

cenbedarf oder auf Geschwindigkeit optimiert sind.

3.4.2.3. CHiMPS

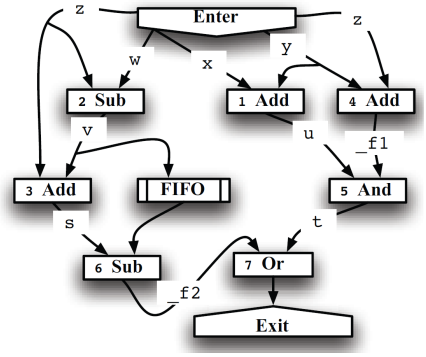
***C**ompiling **H**igh-level Languages **i**nto **M**assively **P**ipelined **S**ystems* (CHiMPS), entwickelt von Xilinx und der Universität Washington [68], vereinfacht HPC-Programmierern den Umgang mit FPGAs ohne weitere FPGA-Kenntnisse anlernen zu müssen. Die Idee ist es, ein FPGA in einen CPU-Sockel einer Mehrsockel-Hauptplatine zu setzen. Als Entlastung der CPU lässt sich C-Quelltext in eine Pipeline für den FPGA übersetzen und ausführen. Der FPGA und die CPU teilen sich den gleichen Adressraum. D.h. der FPGA arbeitet mit gleich hoher Speicherbandbreite und mit sehr geringer Latenzzeit, vergleichbar wie die der CPU. Auch Interprozesskommunikation zwischen CPU und FPGA ist sehr effizient möglich.
Bild 3.4 zeigt ein ANSI-C-Beispiel, das in die Zielsprache ***C**HiMPS **T**arget **L**anguage* (CTL) übersetzt wird. CTL unterstützt 42 Assemblerbefehle, zu denen jeweils ein VHDL-Block existiert, auf die im letzten Schritt der Übersetzung die Befehle abgebildet werden. Neben den arithmetischen Instruktionen gibt es wenig andere, die den Datenfluss beeinflussen (bsp. `if-else`-Konstrukte bzw. `for`-Schleifen).

```
int foo(int x, int x,
        int y, int z)
{
   int u = x + y;
   int v = w - z;
   int s = z + v;
   int t = u & (y + z);
   int q = t | (s - v);
   return q;
}
```

(a) C-Quelltext

```
Enter foo; w,x,y,z
 reg u,v,s,t,q
 reg _f1, _f2
  1: add x,   y;   u
  2: sub w,   z;   v
  3: add z,   v;   s
  4: add y,   z; _f1
  5: add u, _f1;   t
  6: sub s,   v; _f2
  7: or  t, _f2;   q
Exit foo; q
```

(b) CTL-Instruktionen



(c) VHDL-Blöcke

Bild 3.4.: Übersetzungsverlauf von CHiMPS. Der C-Quelltext a) wird in die Zielsprache b) übersetzt und auf VHDL-Blöcke c) abgebildet.

Das Bild c) zeigt im Datenflussgraph ein FIFO, das als Cache von der Übersetzung hinzugefügt wurde. CHiMPS verfolgt den Ansatz, viele kleine lokale verteilte Caches zu verwenden, anstatt wenige große mehrstufige Caches. Dies spart FPGA-Ressourcen für die Cache-Kohärenz ein, da jeder lokale Cache einer einzelnen unabhängigen Funktion zugeordnet ist. Um sicher zu stellen, dass keine zwei Zeiger auf das selbe Feld zeigen und somit zwei Caches instantiiert werden, die inkohärent werden können, muss das ANSI-C Schlüsselwort `restrict` verwendet

werden. Auch wenn es unüblicher Programmierstil ist, mehrere Zeiger für das selbe Objekt zu benutzen, darf das Schlüsselwort nirgendwo fehlen, auch wenn man es implizit annehmen könnte. Eine weitere Ergänzung im Quelltext, CHiMPS nutzen zu können, sind die `#pragma`-Anweisungen, die den Teil markieren, der für den FPGA übersetzt und ausgeführt werden soll. Mit diesen wenigen Änderungen des Quelltextes lässt sich ein Beschleunigungsfaktor von 2.8 bis 36.9 für bekannte Anwendungen aus der HPC (*Black-Scholes*, *Smith-Waterman*, *immul*, *sobol* und *swm*) erreichen.

Es gibt zusätzliche `#pragma`-Anweisungen, die dem Kompilierer Informationen liefert, wie bessere Rechenleistung zu erwarten ist. Ein hardwarekundiger Programmierer muss die Anweisungen an geeigneten Stellen im Quelltext ergänzen.
Cache Anweisungen. Der Programmierer kann Cachegröße, Zeilenlänge, Anzahl der Bänke, Cache Lese-Schreibrechte konfigurieren und hat damit einen entscheidenden Einfluss auf den optimalen Datenfluss.
Separate Speicher. Das Einbinden weiterer Speicher reduziert den Zugriff auf den langsameren Hauptspeicher.
Implementierungsstil. Der FPGA kann eine TCL-Operation mit unterschiedlichen Ressourcen umsetzen. Mit Anweisungen wird dem Kompilierer mitgeteilt, welche Operation beispielsweise mit Logikzellen, mit DSP-Slices oder sogar mit einem Softcore-Prozessor implementiert wird.
Schleifen aufrollen. Die Anzahl der Schleifendurchläufe kann durch Daten-Parallelität reduziert oder gar ganz aufgelöst werden.
Integer-Bitbreiten spezifizieren. Nicht in jedem Programm wird die Bitgenauigkeit eines Integers voll ausgenutzt. Kennt der Programmierer den Wertebereich der Berechnungen, können durch Eingrenzungen der Bitbreite FPGA-Ressourcen eingespart werden.

3.4.3. Parallele C-Sprachen für den FPGA

3.4.3.1. FCUDA

FCUDA [67] ist eine CUDA-Erweiterung, die Kernels auf den FPGA abbildet. Mit ergänzenden Anweisungen werden die Kernel-Funktionen in eine C-Variante namens AutoPilot-C [86] übersetzt. Die Syntheseprogramme von AutoPilot erstellen eine Netzliste, die mit der Xilinx-Werkzeugkette das Design auf den FPGA zur Ausführung bringt.

Der erste Übersetzungsschritt in Bild 3.5 zeigt die Aufgabe, den Kernel-Quelltext mit `#pragma` -Anweisungen zu ergänzen. Die wichtigsten Anweisungen parametrisieren: die parallele Ausführung, wie das CUDA-Grid auf dem FPGA aufgeteilt wird (FCUDA GRID und FCUDA

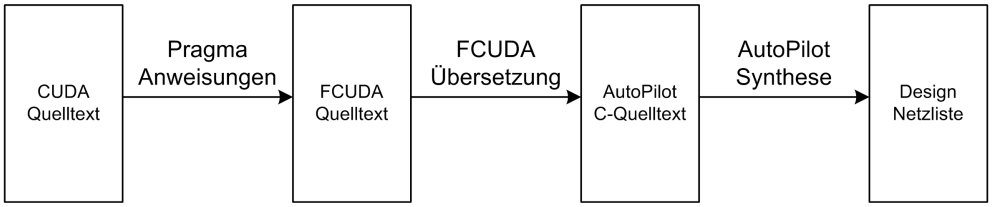


Bild 3.5.: CUDA-FPGA Übersetzungskette

BLOCKS), die Synchronisation, welcher Barrieretyp verwendet werden soll (FCUDA SYNC), die Speicherübertragung, wie die Kernelfunktion auf den Speicher zugreifen kann (FCUDA TRANSFERS) und die Berechnungen, wie viele FPGA-Rechenkerne verwendet werden (FCUDA COMPUTE).
Im zweiten Schritt der Übersetzungen wird der FCUDA-Quelltext in AutoPilot-C konvertiert. Die Konvertierung ersetzt die eingebauten parallelen Index-Variablen durch Thread-Schleifen. Dieser Schritt ist vergleichbar mit einer Serialisierung, d.h. die mehrfache Ausführung der Kernel-Funktion geschieht jetzt mit Hilfe von Schleifen. Eine Synchronisation wird mit zwei Teilschleifen ersetzt, eine Schleife vor der Synchronisation und eine danach. Wenn die erste Teilschleife endet, haben alle serialisierten Threads den Synchronisationspunkt erreicht. Mit Hilfe der Anweisungen wird die Softwarestruktur hergestellt, die AutoPilot für eine parallele Abarbeitung benötigt.
Im dritten Teil der Übersetzung wird der AutoPilot-Quelltext auf FPGA-Softcore-Prozessoren abgebildet. Der serialisierte Quelltext wird durch mehrfache Funktionsaufrufe wieder parallelisiert, die nacheinander auf den Softkernprozessoren ausgeführt werden.

3.4.3.2. OpenRCL

OpenRCL ist eine Entwicklung die OpenCL für den FPGA nutzbar macht. Der Konferenzbeitrag [52] von 2010 beschreibt die Architektur des entsprechenden FPGA-Designs, erläutert den Übersetzungsvorgang und zeigt vergleichbare Ausführungsgeschwindigkeiten gegenüber GPUs. Ziel der Entwicklung ist es, erstens FPGAs für die Architektur übergreifende Sprache OpenCL nutzbar zu machen, und zweitens, einen besseren Wirkungsgrad im Energieverbrauch zu erreichen als die Vergleichsarchitekturen GPU und CPU.

FPGA-Design. Bild 3.6 zeigt die unterschiedlichen Speicher, ein Kommunikationsnetzwerk (Kreuzschienenverteiler) und die Threadprozessoren. Jeder Threadprozessor (PE) hat einen privaten und einen lokalen Speicher (P/L). Auf den privaten Speicher kann nur der anliegende Threadprozessor zugreifen. Auf den lokalen Speicher haben alle Threads der selben *work-group* gegenseitigen Zugriff. Für den Datenaustausch nutzen sie den Kreuzschienenver-

teiler mit eignen Speichern (SP), der mit Blick auf eine maximale Leitungslänge im FPGA sehr viele Kommunikationsteilnehmer erlaubt. Auf den global-geteilten Speicher können alle Threads direkt zugreifen. Die Architektur ist der einer GPU sehr ähnlich. Tatsächlich sind

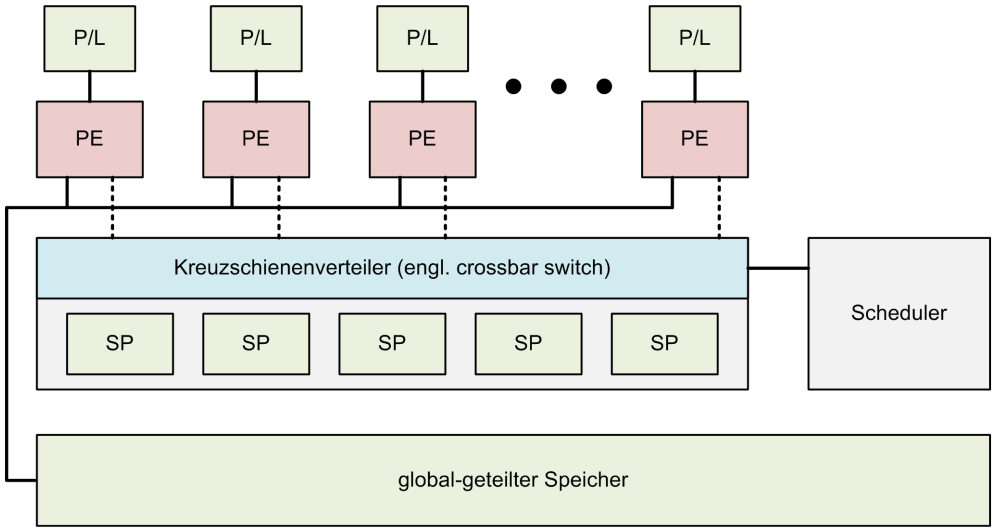


Bild 3.6.: Schematische Architektur der Speicher und der prozessierenden Elemente

die Threadprozessoren aus einfachen fünfstufigen MIPS-Multithread-Prozessoren mit dynamischen Prozess-Schedulern implementiert worden. Jeder Prozessor ist parametrisiert, so dass die Anzahl der Threadprozessoren und die Bitbreite je nach FPGA-Größe gewählt werden kann. Weiter haben die Autoren die Idee, spezifische Instruktionseinheiten für eine Leistungssteigerung hinzuzufügen.

Übersetzungsvorgang. Um den Programmieraufwand einer Übersetzung gering zu halten, werden so viele existierende Kompiliertechiken und Programmteile verwendet, wie nur möglich. Zum einen spart dies viel Entwicklungszeit und zum anderen erhöht dies die Zuverlässigkeit, da die Kompilierer-Programmteile bereits weitläufig genutzt werden. Der Übersetzungsvorgang ist in Bild 3.7 dargestellt. Es gibt zwei Quellen für die Übersetzung, einmal die OpenCL Kernelfunktionen und einmal die Laufzeitumgebung, die in unterschiedlichen Sprachen geschrieben werden können. Das GCC-Frontend kann beide Quellen übersetzen. Während die Laufzeitumgebung die LLVM-Optimierer durchlaufen und effiziente übersetzte Objekte generieren, werden die übersetzten Kernelfunktionen einer statischen Speicherzugriffsanalyse unterzogen. Somit kann der Datenfluss zur Kompilierzeit optimal konfiguriert werden, wodurch die OpenRCL-Implementierung ihre Geschwindigkeitsleistung erzielt. Die Analyse ist neben dem Kernel-Scheduler die entscheidende Änderung zur klassischen OpenCL-Übersetzungskette. Die Generierung der Zielsprache und das Linken wurde auf ein existierendes LLVM-Backend aufgesetzt und der speziellen MIPS-Multithread-Archtitektur auf dem FPGAs ange-

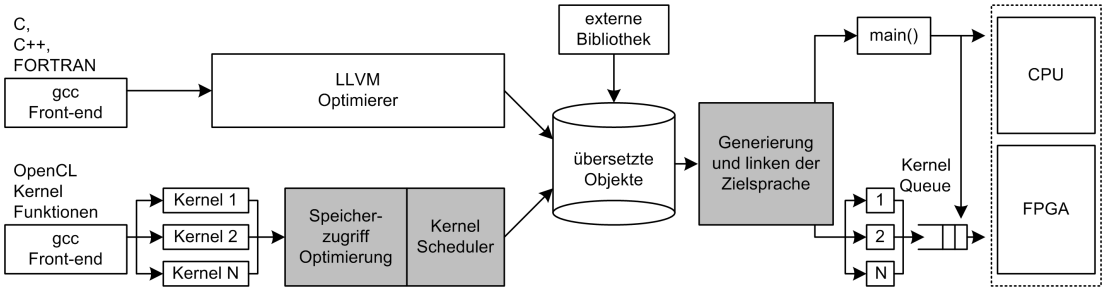


Bild 3.7.: Leicht modifizierte OpenCL-Übersetzungskette. Die grauen Blöcke machen den Unterschied von der OpenRCL-Implementierung aus.

passt. Die Main-Funktion wird auf der CPU ausgeführt, die Kernelfunktionen aus einer Queue startet, die ihrerseits auf dem FPGA ausgeführt werden.

3.4.4. Architekturübergreifende C-Sprachen für den FPGA

3.4.4.1. OpenCL

In der Spezifikation von OpenCL [36] wird angedeutet, dass die Sprache für alle möglichen Architekturen, unabhängig vom Betriebssystem, geeignet ist. Dazu zählen Vielkern-CPUs, GPUs, Cell-Prozessoren und DSP. Weiter gibt es, für den eingebetteten Bereich, eine Spezifikation, die geringere Anforderungen an die Funktionalität stellt. Die folgende Tabelle listet existierende und verfügbare OpenCL-Implementierungen auf.

Datum	Entwickler	Architektur	Quelle
20.04.2009	NVidia	GPU	[11]
05.08.2009	AMD	GPU und CPU	[9]
30.06.2010	IBM	Power und Cell	[12]
13.09.2010	Intel	CPU	[13]

Tabelle 3.2.: OpenCL-Implementierungen aufgelistet nach Erscheinungsdatum.

Apple [10] entwickelte für die GPUs von Nvidia und AMD eine eigene OpenCL-Implementierung, die im Betriebssystem „Mac OS X Snow Leopard“ verankert ist. ARM, S3 und VIA bieten bereits Produkte an, die OpenCL-fähig sind, jedoch existieren noch keine Softwarepakete für potenzielle Entwickler. Eine vollständige Liste von OpenCL-Implementierungen bzw. Produkten, die OpenCL-fähig sind, gibt es auf der Seite von Khronos [14].

OpenCL für den FPGA nutzbar zu machen, wie es das OpenRCL-Projekt 3.4.3.2 gemacht hat,

bietet den Vorteil, keine weitere Programmiersprache lernen und keine weitere Entwicklungszeit für eine Portierung investieren zu müssen.

3.4.4.2. Microsoft Accelerator

Die Veröffentlichung „Computer without Processors“ [70] vertritt die Meinung, dass der Rechenbedarf von *Cloud Computing* am besten mit einer Mischung vieler Rechenarchitekturen (CPU, GPU und FPGA) erfüllt wird und dass in der Zukunft häufiger Mischarchitekturen zum Einsatz kommen. Das Entwicklungswerkzeug „Accelerator“ [72] von Microsoft zielt auf die Unabhängigkeit der Architektur ab. Die Entwicklung ist keine neue Sprache, sondern eine Bibliothek, die zu unterschiedlichen Sprachen (C++, C#, Haskell, ...) hinzu gebunden wird. Accelerator ermöglicht eine architekturübergreifende Übersetzung für unterschiedliche Zielgeräte. Das Quelltextbeispiel 3.3 zeigt eine eindimensionalen Faltung in der Sprache C#, die auf mehreren Zielgeräten ausgeführt werden kann.

```
1  using Microsoft.ParallelArrays;
2  using A = Microsoft.ParallelArrays.ParallelArrays;
3  namespace AcceleratorSamples
4  {
5    public class Convolver
6    {
7      public static float[] Convolver1D(Target computeTarget, float[] a,
           FloatParallelArray x)
8      {
9        var n = x.Length;
10       var y = new FloatParallelArray(0.0f, new [] {n});
11
12       for (int i = 0; i < a.Length; i++)
13         y += a[i] * A.Shift(x, -i);                    // Hier wird nichts berechnet.
14
15       float[] result = computeTarget.ToArray1D(y);     // Berechnung auf dem
             Zielgeraet.
16       return result;
17     }
18   }
19 }
```

Quelltext 3.3: Beispielquelltext in C# der Accelerator-Bibliothek [70]

In Zeile 10 wird ein `FloatParallelArray` erstellt, das der Speicherung der Ergebnisdaten auf dem Zielgerät dient. Ebenso wird der Funktion ein paralleles Array übergeben, dessen Daten bereits auf dem Zielgerät sind. Zeile 13 zeigt die Rechenvorschrift der Faltung, ohne jegliche Berechnung. Der Ausdruck `A.Shift(x, -i)` wird auf dem Zielgerät in eine Speicherzugriffsmatrix gewandelt. Ist das Zielgerät ein FPGA, bedeutet dies, dass die Daten, entsprechend des Schleifenindex, an die entsprechende Stelle geschoben werden. Das ist eine gute Lösung, um

Speicherzugriffe zu reduzieren, da die Daten einmal aus dem Speicher gelesen werden, aber aus dem Schieberegister mehrfach verwendet werden. Die Berechnung wird in Zeile 15 auf dem Zielgerät zur Ausführung gebracht, je nachdem welches `Target` instantiiert wurde. Es gibt Implementierungen für FPGAs (`FPGATarget`), für GPUs (`DX9Target`) die auf die Shadersprache DirectX Version 9 aufsetzt und CPUs (`X64MulticoreTarget`) für Vielkern-CPUs mit Ausnutzung der SSE3-Vektorisierung.

Die Veröffentlichung demonstriert in den Ergebnissen, wie die Beschleunigung der DX9-Version auf Ati und NVidia-GPUs mit länger werdenden Datenreihen linear skaliert. Während die Beschleunigung mit der Vielkern-CPU-Implementierung, aufgrund der Cachegröße, ein Maximum hat und die Beschleunigung mit länger werdenden Datenreihen mit kleinerem Beschleunigungsfaktor nicht weiter ansteigt. Ziel der Entwicklung ist es, Algorithmen in einer Sprache zu schreiben und für viele Zielgeräte nutzen zu können, Entwicklungszeit bzw. Portierungszeit einzusparen. Möchte man das Maximum an Leistung des Zielgerätes ausschöpfen, bleibt nichts anderes übrig, als die Nativ-Sprachen (CUDA, VHDL, ...) der Zielgeräte zu benutzen.

3.4.5. Fazit der FPGA-Sprachen

Diese Arbeit strebt den Ansatz an, einen FPGA für die Online-Prozessierung einzusetzen. Folgende Gründe sprechen dafür:

- Der FPGA wird als Kommunikations- und Schnittstellenbaustein benötigt, um die Daten vom Sensor in den PC-Speicher zu transportieren. Da der FPGA sehr leistungsfähig ist, kann er auch Co-Prozessor Aufgaben übernehmen.
- Vom Sensor kommen Daten, die in den FPGA gestreamt werden. Im FPGA werden sie mit geringer Latenzzeit in gestreamter Weise weiterverarbeitet.
- Die Tatsache, dass der Sensor Integerdaten liefert, spricht ebenso für den FPGA.

Um den Biologen eine Programmierung der Online-Prozessierung auf dem FPGA zu ermöglichen, wird in dieser Arbeit ein Kompilierer entwickelt, der eine Kernelfunktion der Sprache OpenCL in eine Hardwarepipeline in VHDL übersetzt (Abschnitt 5.3). Die VHDL-Pipeline soll mit umliegender Logik (Abschnitt Rahmendesign 5.4) und einer OpenCL-Laufzeitumgebung (Abschnitt 5.5) auf einer FPGA-Karte zur Ausführung gebracht werden. Folgende Argumente zeigen den Bedarf des Ansatzes:

- In der Regel beherrschen Biologen kein VHDL, sie brauchen eine einfache Sprache, die sie für die Anpassungen der Online-Prozessierung einsetzen. Handel-C ist keine Option,

da sie lediglich eine vereinfachte Hardwaresprache für Programmierer mit Hardwarekenntnissen ist. Auch CHiMPS bedarf komplexer Pragma-Anweisungen, um die Hardware zu beschreiben.

- Im Allgemeinen gilt, dass serielle Sprachen entweder zusätzliche Anweisungen vom Programmierer für eine parallele Ausführung benötigen (Handel-C und CHiMPS) oder ein Algorithmus versucht, den seriellen Quelltext zu parallelisieren (Trident). Die parallelen Sprachen haben den Vorteil, bereits ein Paradigma zu liefern, wie die Parallelität auf dem FPGA abgebildet werden kann.

- Für den Biologen würden die parallelen Sprachen OpenCL bzw. CUDA und die Microsoft Accelerator Bibliothek in Frage kommen. Sie sind leicht zu erlernen und erfordern nicht zwingendermaßen Hardwarekenntnisse. Für alle Genannten gibt es eine FPGA-Entwicklung. Das Derivat FCUDA, CUDA für den FPGA, benötigt allerdings zusätzliche Anweisungen für den Übersetzungsprozess und scheidet deswegen aus. OpenRCL, OpenCL für rekonfigurierbare Logik, hat alle wünschenswerten Eigenschaften, mit einer Ausnahme (nächster Punkt). Der Microsoft Accelerator ist nicht so bekannt und verbreitet wie OpenCL (oder CUDA) und die Accelerator Bibliothek ist im Vergleich nicht intuitiv zu erlernen. OpenCL ist von der Programmierwelt besser akzeptiert.

- Die OpenRCL (und die FCUDA) Entwicklung verfolgt den Ansatz einer Rechnerarchitektur, wie die einer GPU, vielen Threadprozessoren nachzuahmen. Die Berechnungen werden auf mehrere Threadprozessoren verteilt, wobei jede Operation mehrere Takte Rechenzeit beanspruchen kann. FPGAs entfalten ihr Potenzial mit einer Hardwarepipeline, in der alle Operationen parallel in jedem Takt berechnet werden. Auch die Latenzzeit ist in einer Pipelineverarbeitung geringer als die einer Vielkernrechnerarchitektur.

- Der gravierende Vorteil einer OpenCL-Implementierung ist die doppelte Einsatzfähigkeit. In dieser Arbeit können Algorithmen, die für die Online-Prozessierung (FPGA) entwickelt werden, dann auch für die Offline-Prozessierung (GPU) eingesetzt werden.

Der Ansatz dieser Arbeit, einen OpenCL-Kompilierer zu entwickeln, ist vergleichbar mit der Mischung der drei gezeigter Ansätze. Erstens, es wird eine ähnliche Pipelinegenerierung wie die von CHiMPS verwendet, zweitens wird die parallele Sprache OpenCL für den FPGA benutzt, nach der Grundidee von OpenRCL und drittens lehnt der Übersetzungsvorgang an Trident an, LLVM verwenden zu wollen. Die Mischung der drei existierenden Ansätze zu einem besitzt Vorteile gegenüber jedem der einzeln gezeigten. Dieser Ansatz vereint die Vorteile:

- FPGAs mit der einfachen parallelen Programmiersprache OpenCL beschreiben zu können.

- Über einen Pipelinegenerator zu verfügen, der die Vorteile der Streambarkeit und des hohen Rechendurchsatzes nutzt.
- Die Übersetzung auf LLVM aufzubauen, um die Optimierer nutzen zu können und die Übersetzung einfach entwickeln zu können.

4. Haralick-Algorithmus GPU-beschleunigt

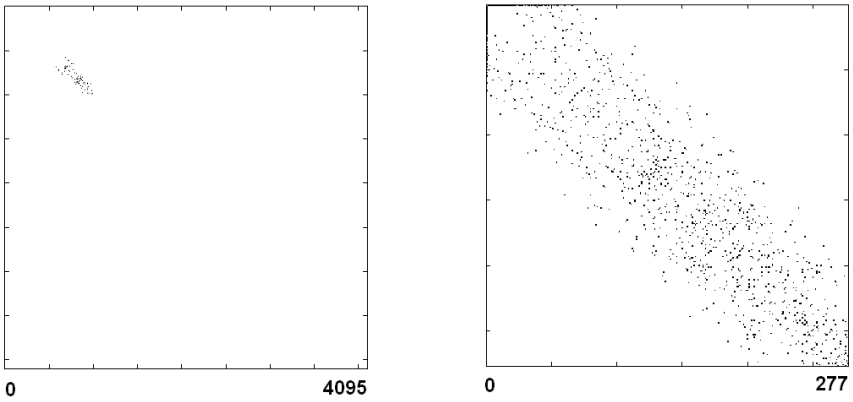
4.1. Untersuchung des Haralick Algorithmusses

Der Haralick Algorithmus besteht aus zwei Teilen. Im ersten Teil werden von den Mikroskopbildern Co-occurrence Matrizen erstellt, die eine Art zweidimensionale Histogramme darstellen. Auf den Matrizen werden die Bildmerkmale berechnet. Die Matrizen und die Bildmerkmale werden im Folgenden beschrieben und für eine beschleunigte Berechnung analysiert.

4.1.1. Co-occurrence Matrizen

Die Berechnung der Co-occurrence Matrizen (Co-Matrizen) basiert auf einer Statistik zweiter Ordnung und ist in [37] und [38] beschrieben. Es werden Histogramm-Matrizen (Co-Matrizen) anhand benachbarter Pixelpaare (erstes und zweites Pixel) aus dem Quellbild gebildet. Die Pixelpaare werden für eine bestimmte Sichtweise angeordnet, gemeint ist ein bestimmter Pixelabstand und ein Winkel des ersten und des zweiten Pixel zueinander. Wobei der Grauwert des ersten Pixel die Zeile der Co-Matrizen adressiert und das zweite Pixel die Spalte. Die ermittelte Zelle wird, entsprechend eines Histogramms, um Eins akkumuliert. Die Co-Matrix ist vollständig, wenn alle Pixelpaare, die aus dem Quellbild entstehen können, aufsummiert worden sind. Ein ausführliches Beispiel gibt es in [37]. Für jede Sichtweise wird eine eigene Co-Matrix gebildet, die ein Vorkommen der Grauwerte unter einer gewissen Anordnung der Pixelpaare repräsentiert. Veranschaulicht sind die Matrizen eine Kombination aus Ortsinformationen und zweidimensionalem Histogramm.
Die Größe der Co-Matrizen ist abhängig von der Anzahl an möglichen Grauwerten im Quellbild. Bei einer Bittiefe von 12 Bit existieren 4096 unterschiedliche Grauwerte und es entsteht mit dem Datentyp `float` eine Matrix mit der Größe von $4096 * 4096 * 4Bytes = 64MBytes$ Speicherbedarf. In einen $1024MBytes$ großen Grafikkartenspeicher würden nur 16 Co-Matrizen hineinpassen, was wiederum die GPU nur zu einem Bruchteil auslasten würde. Für eine massive Parallelisierung des Algorithmusses müssen die Co-Matrizen kleiner werden.
Tatsächlich sind die Co-Matrizen nur spärlich besetzt. Das liegt daran, dass die Zellbilder nicht rein zufällig sind und die Pixelpaare bevorzugte Grauwerte haben. Beispielsweise hat der Zellenrand nur einen kleinen Wertebereich, was bedeutet, dass der Zellenrand rechts und links sehr ähnlich ist. Das gleiche trifft für den Zellkern zu. Besonders der Hintergrund des seg-

mentierten Zellbildes hat überall den gleichen Intensitätswert. Allgemein entstehen bei der Co-Matrix-Berechnung mehr oder weniger bevorzugte Regionen, in denen die Pixelpaare gezählt werden, während andere Regionen komplett leer sind. Bild 4.1 (a) zeigt eine spärlich besetzte Co-Matrix.



(a) (b)

Bild 4.1.: Binärbild einer vollen Co-Matrizen (a) und einer gepackten (b). Weiße Pixel entsprechen dem Wert Null, schwarze Pixel einem von Null verschiedenen Wert.

Um Speicherplatz einzusparen, werden alle Zeilen (Aufgrund der Symmetrie auch Spalten), die ausschließlich Nullwerte enthalten, entfernt. Bild 4.1 (b) zeigt, dass nur noch 278 Zeilen und Spalten gespeichert werden, während die Co-Matrix aus Bild 4.1 (a) in voller Größe 4096 Zeilen und Spalten benötigt. Für dieses Beispiel konnte der Speicherplatzbedarf von 64Mbyte auf 300kBytes reduziert werden. Im Durchschnitt liegt die gepackte Co-Matrix-Größe bei 1,5 MBytes.
Für die spätere Bildmerkmalsberechnung ist die Position eines Wertes in der Co-Matrix von Bedeutung. Die Position repräsentiert einen Grauwert in der vollen Co-Matrix und fließt bei manchen Bildmerkmalen mit in die Berechnung ein. Um später den Grauwert bestimmen zu können, wird neben der gepackten Co-Matrix eine Index/Grauwert-Tabelle konstruiert.
Kompressionsmethoden, die spärlich besetzte Matrizen als Position-Werte-Paar in einer Strukturliste speichern, haben den Nachteil, die Elemente indirekt adressieren zu müssen, was den Speicherzugriff erheblich verlangsamen würde. Diese Methode der gepackten Matrizen stellt einen Kompromiss zwischen wenig Speicherbedarf und direkten Speicherzugriffen dar.

4.1.2. Haralick Textur Merkmale

Die Haralick Texturen Merkmale umfassen 14 Bildmerkmale, zusammengefasst in [73]. In dieser Implementierung wurden die Bildmerkmale (4.1) bis (4.13) optimiert. Bildmerkmal Nummer 14 *Maximul Correlation Coefficient* wurde bereits in der Implementierung der Biologen weggelassen. Sie haben festgestellt, das Bildmerkmal Nummer 14 keinen Beitrag zur besseren Klassifikation der folgenden linearen Diskriminanzanalyse leistet, was eine aufwendige Berechnung unnötig macht. Die erstellten Co-Matrizen sind mit $P_{i,j}$ gekennzeichnet. Alle anderen Definitionen werden weiter unten eingeführt.

$$f_1 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}^2 \tag{4.1}$$

$$f_2 = \sum_{k=0}^{Ng-1} k^2 \left(\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)} \right)_{|i-j|=k} \tag{4.2}$$

$$f_3 = \frac{1}{\sigma^2}\left(\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} (ij)\, P_{(i,j)} - \mu^2 \right) \tag{4.3}$$

$$f_4 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} (i-\mu)^2\, P_{(i,j)} \tag{4.4}$$

$$f_5 = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} \frac{1}{1+(i-j)^2}\, P_{(i,j)} \tag{4.5}$$

$$f_6 = \sum_{k=2}^{2Ng} k\, P_{x+y}(k) \tag{4.6}$$

$$f_7 = \sum_{k=2}^{2Ng} (k - F_{SAVG})^2\, P_{x+y}(k) \tag{4.7}$$

$$f_8 = -\sum_{k=2}^{2Ng} P_{x+y}(k)\; log[P_{x+y}(k)] \tag{4.8}$$

$$f_9 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\; log[P_{(i,j)}] \tag{4.9}$$

$$f_{10} = \sum_{k=0}^{Ng-1} \left[P_{x-y}(k)\; (k - \sum_{l=0}^{Ng-1} l\, P_{x-y}(l)\,)^2 \right] \tag{4.10}$$

$$f_{11} = -\sum_{k=0}^{Ng-1} P_{x-y}(k)\; log[P_{x-y}(k)] \tag{4.11}$$

$$f_{12} = \frac{f_9 - HXY1}{H} \tag{4.12}$$

$$f_{13} = \sqrt{1 - \exp\left[-2.0\,|HXY2 - F_9|\right]} \tag{4.13}$$

Die Definitionen zu den Bildmerkmalen sind in den Gleichungen (4.14) bis (4.21) aufgelistet.

$$P_{x+y}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\ k = i+j, k = 2,3,...2Ng-2 \tag{4.14}$$

$$P_{|x-y|}(k) = \sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\ k = |i-j|, k = 0,1,...Ng-2 \tag{4.15}$$

$$p_{(i)} = \sum_{j=1}^{Ng} P_{(i,j)} \tag{4.16}$$

$$\mu = \sum_{g=1}^{Ng} g\, p_{(g)} \tag{4.17}$$

$$\sigma^2 = \sum_{g=1}^{Ng} p_{(g)}\,(g-\mu)^2 \tag{4.18}$$

$$HXY1 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i,j)}\, log[P_{(i)}P_{(j)}] \tag{4.19}$$

$$HXY2 = -\sum_{i=1}^{Ng}\sum_{j=1}^{Ng} P_{(i)}\,P_{(j)}\, log[P_{(i)}P_{(j)}] \tag{4.20}$$

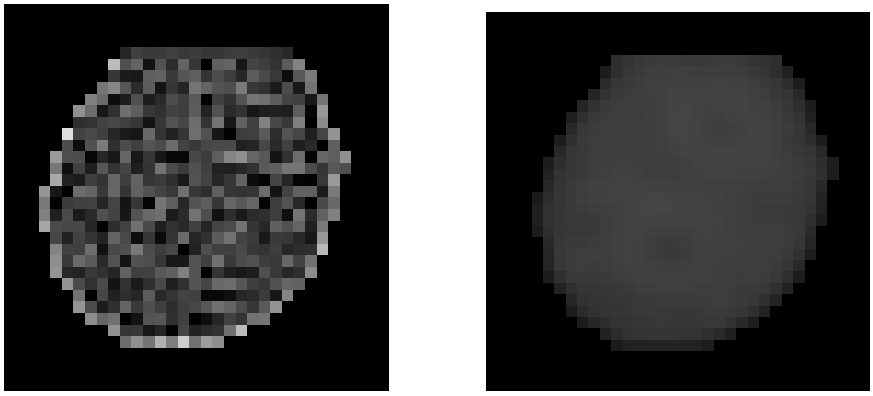
$$H = \sum_{g=1}^{Ng} p_{(g)}\, log[p_{(g)}] \tag{4.21}$$

Die meisten der gezeigten Bildmerkmale haben eine visuelle Bedeutung. Bild 4.2 zeigt die gleiche Zelle, einmal mit zusätzlichem Rauschen (a) und einmal als weichgezeichnetes Bild (b). Für beide Zellbilder sind die Bildmerkmale *Contrast* (4.2), *Inverse Different Moment* (4.5) und *Entropy* (4.9) berechnet und in der Tabelle 4.1 dargestellt.

	Abb. 4.2 (a)	Abb. 4.2 (b)
Contrast (4.2)	$3.625E5$	$1.035E4$
Inverse Different Moment (4.5)	0.5558	0.5715
Entropy (4.9)	5.5187	5.4807

Tabelle 4.1.: Bildmerkmalswerte für ein verrauschtes Zellbild 4.2 (a) und ein weichgezeichnetes (b)

Der Wert für das Bildmerkmal *Contrast* ist größer für kontrastreichere Bilder. Im eben genannten Beispiel ist der Wert *Contrast* für die verrauschte Zelle tatsächlich größer als der Wert für die weichgezeichnete Zelle. Umgekehrt ist für das Bildmerkmal *Inverse Different Moment* der Wert kleiner für kontrastreiche Bilder, wie in der Tabelle zu erkennen ist. Die *Entropy* ist ein



(a) (b)

Bild 4.2.: Zwei gleiche Zellbilder, (a) mit zusätzlichem Rauschen und (b) weichgezeichnet

Maß für eine Zufälligkeit, deren Wert für weiche Bilder kleiner ist. Weitere Beispiele für die Anschaulichkeit der Bildmerkmale sind in [73] und [39] zu finden.

Die Gleichungen der Bildmerkmale aus dem Buch [73] sind für den allgemeinen Fall symmetrische und asymmetrische Co-Matrizen. Die aufgelisteten Bildmerkmale sind für symmetrisch quadratische Co-Matrizen vereinfacht worden, gemeint sind die gleichen Berechnungsergebnisse für Zeilen und Spalten. Somit konnten in den Bildmerkmalen *Correlation* (4.3), *Information Measure I* (4.3) und *Definitionen Mean* (4.17), *Variance* (4.18), *Entropy* (4.18) Terme gekürzt oder durch Ausdrücke mit weniger Rechenoperationen ersetzt werden.

Die restlichen Bildmerkmale und Definitionen sind unverändert und werden zur Vollständigkeit mit Namen aufgelistet: *Angular Second Moment* (4.1), *Contrast* (4.2), *Variance* (4.4), *Inverse Difference Moment* (4.4), *Sum Difference Average* (4.6), *Sum Variance* (4.7), *Sum Entropy* (4.8), *Entropy* (4.9), *Difference Variance* (4.10), *Difference Entropy* (4.11) und *Information Measurement II* (4.13).

Die meisten Bildmerkmale, (4.1)-(4.4), (4.6)-(4.8) und (4.10)-(4.13), hängen von anderen Bildmerkmalen und Zwischenergebnissen bzw. der Definitionen ab. Um eine zeitintensive doppelte Berechnung der Bildmerkmale und Zwischenergebnisse zu vermeiden, muss die richtige Reihenfolge ermittelt werden. Die Abhängigkeiten wurden analysiert und in einem Graphen in Bild 4.3 dargelegt. Jeder Kreis des Graphen ist ein Bildmerkmal, für dessen Berechnung alle anderen (mit Linien) verbundenen Kreise bzw. Kästchen (mit Zwischenergebnissen) berechnet sein müssen, da dessen Ergebnisse in die Berechnung mit einfließen. Zum Beispiel muss erst das Ergebnis von Bildmerkmal (4.7) ermittelt werden, weil Bildmerkmal (4.6) es für seine Berechnung benötigt.

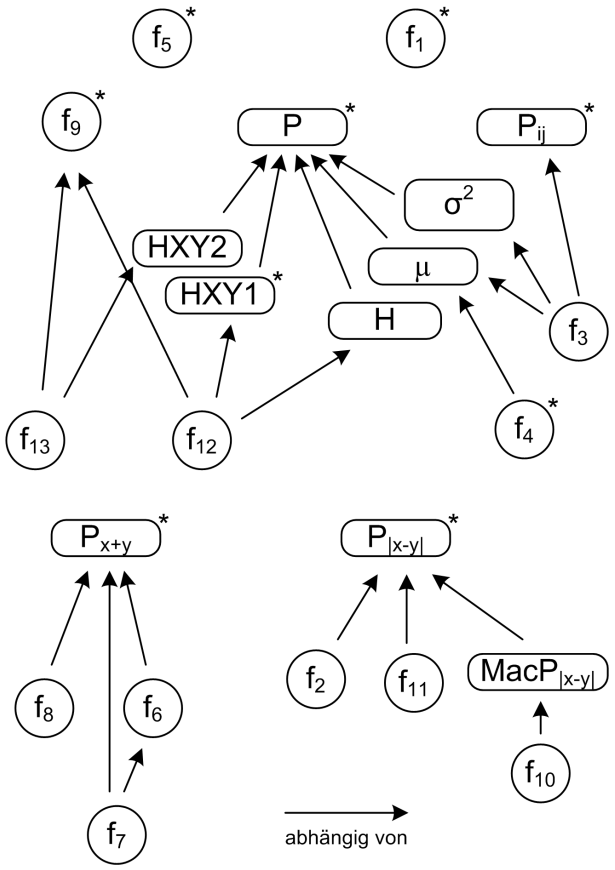


Bild 4.3.: Abhängigkeitsgraph für die Berechnung des Haralick Texturen Bildmerkmals (in Kreise) und Zwischenergebnisse (in Kästchen). Alle Blätter, gekennzeichnet mit einem Stern (*), sind ihrerseits von den Co-Matrizen abhängig.

Der Abhängigkeitsgraph bietet mehrere Optimierungen für eine Implementierung. Wie bereits erwähnt, zeigt er die optimale Reihenfolge der Berechnungen, um doppelte zu vermeiden. Des weiteren können Programmstrukturen abgeleitet werden. Alle Blätter eines Zweigs können gemeinsam in einer Funktion bzw. Schleife implementiert werden. Daraus ergibt sich, dass für diesen Zweig die Quellen nur einmalig gelesen werden, was wiederum unnötige Speichertransfers reduziert. Ebenso wird das Speicherzugiffsverhalten auf kleinere Regionen des Speichers konzentriert, worin Vorteile für Architekturen mit Caches entstehen.

4.2. CPU Implementierung

Das erste Ziel war, die bereits existierende Software Version zu optimieren, um sie auf einem Computercluster als Ein-CPU-Applikation laufen zu lassen. Durch mehrfache Ausführung mit unterschiedlichen Zellbildern kann der gesamte Rechenaufwand parallelisiert werden.

Bild 4.4 zeigt die Softwarestruktur in einem Struktogramm. Die äußere Schleife iteriert über alle Zellen (C) eines Multizellbildes. Innerhalb der Schleife werden alle Co-Matrizen generiert. Weiter folgt eine Doppelschleife über den Winkel (A) und Distanz (D). Das heißt, für jede Zelle (C) existieren (A)*(D) Co-Matrizen, die für alle 13 Bildmerkmale seriell berechnet werden müssen.

In der Doppelschleife werden die Bildmerkmale in der Reihenfolge berechnet, welche der Abhängigkeitsgraph in Bild 4.3 empfiehlt. In den ersten beiden Blöcken werden die Zwischenergebnisse P_{xy}, $P_{|x-y|}$ und P_{x+y} sowie die Bildmerkmale (4.1), (4.5) und (4.9) berechnet. Diese Reihenfolge liefert ideale Raten von Cache-Treffern in der CPU, da stets aus der selben Speicherregion gelesen wird, in der sich die Co-Matrix befindet. Auch in den folgenden Blöcken bleiben die Speicherzugriffe auf die P_{xy} und die $P_{|x-y|}$ Speicherregion begrenzt. Nur in den letzten drei Blöcken können die Speicherzugriffe nicht regional begrenzt werden. Besonders die Lesezugriffe auf die bereits berechneten Bildmerkmale (4.6) und (4.9) ersparen eine doppelte Berechnung, und im Fall von (4.9) wird eine dreifache Berechnung eingespart.
Die eben vorgestellte Struktur wurde in einer C++ Klasse implementiert, die mit Inline-Funktionen den Funktionsaufruf-Overhead einspart. Das Programm wurde mit der besten Kompiliereroptimierung übersetzt. Das beinhaltet Bemühungen vom Kompilierer, die Berechnungen in den Schleifen mit den SSE-Instruktionen zu vektorisierten. Aus diesem Grund sind die Schleifen im Quellcode kurz und einfach gehalten.

4.3. GPU Implementierung

In diesem Kapitel wird dargelegt, wie der Haralick Algorithmus auf der GPU-Architektur parallelisiert wird und es werden Implementierungsdetails dargelegt.

4.3.1. Parallele Struktur

In CUDA werden Kernelfunktionen auf der GPU vielfach mit unterschiedlichen Threads ausgeführt. Die Threads sind in Blöcken (CUDA-Blöcken) strukturiert, die wiederum in einem Grid angeordnet sind, siehe in den GPU Grundlagen 2.1.2. D.h. jeder Thread in jedem Block

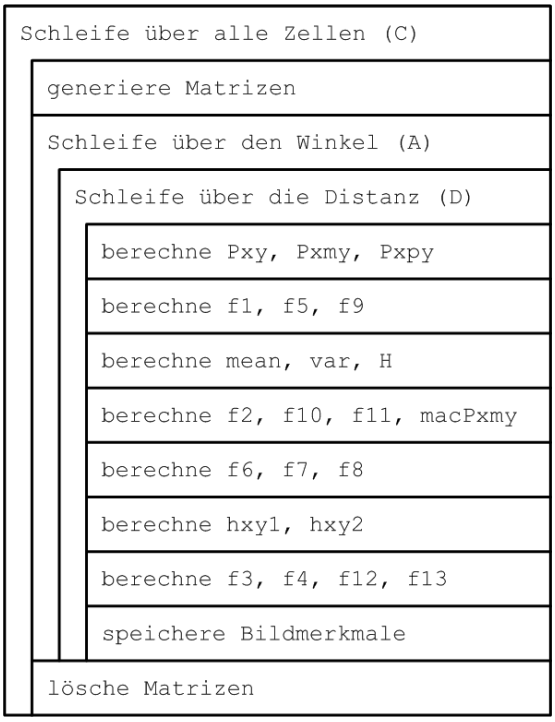


Bild 4.4.: Struktogramm der optimierten Softwareversion

des Grids führt die gleiche Kernelfunktion auf unterschiedlichen Daten aus. Es besteht die Freiheit, die angelegten Blöcke ein- oder zweidimensional in einem Grid und die Threads dreidimensional innerhalb eines Blocks anzuordnen. Dabei entstehen bis zu fünf Indices (`tx`, `ty`, `tz` für die Threads und `bx`, `by` für die Blöcke), mit denen jeder Thread nummeriert wird. In einer Kernelfunktion werden die Indices genutzt, für jeden Thread unterschiedliche Datenelemente zu adressieren. Dabei ist es wichtig, die Blöcke und das Grid so zu dimensionieren, dass die Indizierung zur Datenstruktur im Speicher passt. Die passende Zuordnung der vielen Threads zu den Datenelementen im Speicher erleichtert die Adressierung und bietet lineares Lesen, für schnelle Datentransfers.

In dieser Anwendung wird für jede generierte Co-Matrix ein eigener Block geschaffen. Jede Operation, die auf die Blöcke angewendet wird, impliziert die parallele Ausführung auf alle Co-Matrizen. In Bild 4.5 ist die Struktur der GPU Version in einem Struktogramm erläutert. Die äußerste Schleife iteriert über alle Zellen des Multizellbildes. Der Unterschied zur CPU Version ist, dass mit jedem Scheifendurchlauf C Zellen gleichzeitig gelesen werden. Innerhalb der Schleife werden für alle C Zellen wiederum gleichzeitig alle Co-Matrizen[1] AD erstellt.

[1] Für jede Sichtweise des Winkel A und der Distanz D wird eine eigene Co-Matrix generiert. Aus diesem beläuft sich die Anzahl an Kombinationen zu $A * D = AD$.

Die Anzahl der CUDA-Blöcke beläuft sich auf $C * AD$, in denen die Zwischenergebnisse und Bildmerkmale berechnet werden. Am Ende des Schleifenkörpers werden die Bildmerkmale gespeichert und die Co-Matrizen gelöscht.

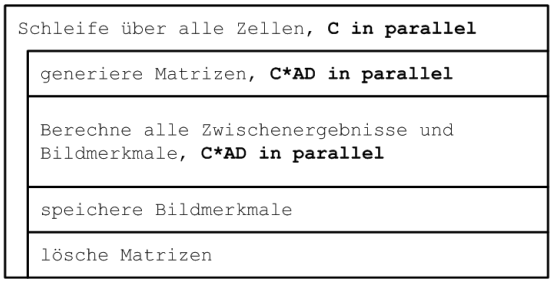


Bild 4.5.: Struktogramm der parallelen GPU Version

Die Berechnung des Haralick Algorithmusses mit allen Bildmerkmalen und Zwischenergebnissen ist komplex. Implementiert in eine einzige Kernelfunktion, würde das den Quellcode seitenlang werden lassen und die benötigten Ressourcen der GPU würden nicht ausreichen. Für eine effiziente Auslastung der GPU wurde der Haralick Algorithmus auf 26 Kernelfunktionen herunter gebrochen, die einzelne Zwischenergebnisse bzw. Bildmerkmale berechnen.

Die Reihenfolge, in der die Zwischenergebnisse und Bildmerkmale berechnet werden, lässt sich auch für die GPU Implementierung vom Abhängigkeitsgraph (Bild 4.3, Seite 54) ableiten. Mit der optimalen Reihenfolge sind die Kernelfunktionen gruppiert worden, entsprechend der Leseregion im Grafikkartenspeicher, siehe hierzu Tabelle 4.2. Die Gruppierung sorgt für regional konzentrierte Speicherzugriffe, die bei den Texturencaches höhere Trefferraten mit resultierendem Geschwindigkeitsgewinn versprechen.
Wie bereits erwähnt, steht für die Berechnungen jeder individuellen Co-Matrix ein Block zur Verfügung. Das heißt, die Anzahl der Blöcke ist konstant, nur die Anzahl der Threads innerhalb der Blöcke muss optimal dimensioniert werden, passend zum Ressourcenverbrauch jeder der 26 Kernelfunktionen. Die vielen Threads können genutzt werden, um gleichzeitig Elemente von der Co-Matrix zu lesen und zu berechnen.

Mit vielen parallelen Threads werden die Speichertransporte durch die Berechnungseinheiten verborgen. Während ein Teil der Threads auf Daten vom Grafikkartenspeicher warten, können andere Threads die Berechnungseinheiten belegen, die bereits mit Daten versorgt wurden. Das Verstecken der Latenzzeit des Grafikkartenspeichers funktioniert gut, sofern genügend Threads existieren, die abwechselnd warten und berechnen können. Ziel ist es, die Auslastung der Berechnungseinheiten zu optimieren.

Initialisierungsteil	
Kernelfunktion 0A	erstelle Index- / Grautontabellen
Kernelfunktion 0B	setze Co-Matrizen auf den Wert Null
Kernelfunktion 0C	berechne die Co-Matrizen
Kernelfunktion 0D	normalisiere die Co-Matrizen
Teil 1, Lesen von den Co-Matrizen	
Kernelfunktion 1A	berechne f_1
Kernelfunktion 1B	berechne f_5
Kernelfunktion 1C	berechne f_9
Kernelfunktion 1D	berechne P
Kernelfunktion 1E	berechne $P_{\|x-y\|}$
Kernelfunktion 1F	berechne P_{x+y}
Teil 2, Lesen von P	
Kernelfunktion 2A	berechne *mean*
Kernelfunktion 2B	berechne *var*
Kernelfunktion 2C	berechne H
Teil 3, Lesen von $P_{\|x-y\|}$	
Kernelfunktion 3A	berechne f_2
Kernelfunktion 3B	berechne f_{11}
Kernelfunktion 3C	berechne $MacP\|x-y\|$
Kernelfunktion 3D	berechne f_{10}
Teil 4, Lesen von P_{x+y}	
Kernelfunktion 4A	berechne f_6
Kernelfunktion 4B	berechne f_8
Kernelfunktion 4C	berechne f_7
Teil 5, Lesen von den Co-Matrizen	
Kernelfunktion 5A	berechne f_3 mittels $P_{(i,j)}$
Kernelfunktion 5B	berechne f_4
Kernelfunktion 5C	berechne $HXY1$, f_{12}, lese von P
Kernelfunktion 5D	berechne $HXY2$, f_{13}, lese nur von P

Tabelle 4.2.: Liste aller Kernelfunktionen in der Ausführungsreihenfolge. Linke Spalte beinhaltet den Funktionsnamen; Rechte Spalte beschreibt die Berechnung.

4.3.2. Details der Implementierung

Dieser Abschnitt beschreibt Details der Implementierung und Probleme mit deren Lösungen. Die Beschreibung ist chronologisch zu den Berechnungen, beginnt mit dem Speichertranfser in die GPU und endet mit dem Auslesen der Ergebnisse. Alle parallelen Operationen werden vereinfacht als singulärer Fall ausgedrückt. Dabei darf nicht vergessen werden, dass die Operationen parallel auf vielen Matrizen und vielen Zellen aufgeführt werden.

4.3.2.1. Kopie der Zellen

Im ersten Schritt wird das Multizellbild in den Arbeitsspeicher der GPU kopiert. Nachdem die Zelle mit gegebenen Koordinaten identifiziert wurde, werden sie in einen separaten Speicherbereich für dessen direkten Zugang transferiert. Der Zweck des letzten Speichertransfers ist die Erweiterung des Zellbildes mit einem Rahmen der Distanz $D = 5$ Pixel, bestehend aus Nullen, zum Zellbild. Dieser Schritt ist für die spätere Pixelpaarbildung notwendig, damit auch Pixel am Zellbildrand Paare über die Grenze des Zellbildes hinaus bilden können. Letztendlich wird ein Texturencache auf das Zellbild, zum beschleunigten Lesen, eingerichtet.

4.3.2.2. Lookup Tabellen

Es gibt zwei sortierte Nachschlage-Tabellen (*Lookup*-Tabellen). Die erste enthält Index-Grauwertpaare, aufsteigend sortiert mit dem Index, und die zweite Grauwert-Indexpaare, aufsteigend sortiert mit dem Grauwert. Die Index-Grauwerttabelle wird bei der Merkmalsberechnung benötigt. Mit dem Index der gepackten Co-Matrix wird der zugehörige Grauwert ermittelt. Die Grauwert-Indextabelle wird bei der Co-Matrixgenerierung benötigt. Für den Grauwert der Zelle wird der Index der gepackten Co-Matrix nachgeschlagen.
Die Grauwerte-Index Zuordnung für eine Zelle ist eine 1:1 Beziehung. Ermittelt wird die Zuordnung, indem alle vorkommenden Grauwerte in einem Vektor markiert werden und die Zeilen der unmarkierten Grauwerte entfernt werden. Die Größe des übrig gebliebenen Vektors entspricht der Anzahl der vorkommenden Grauwerte aus der Zelle. Aus der eben gewonnenen Index-Grauwerttabelle wird durch Umkehrung die Grauwert-Indextabelle gewonnen.

4.3.2.3. Gepackte Co-Matrix gezielt generieren

Wie viele Grauwerte in einer Zelle tatsächlich vorkommen, ist von der Generierung der Lookup Tabellen bekannt. Die Anzahl der vorkommenden Grauwerte ist gleich der Kantenlänge der zu generierenden gepackten Co-Matrix. Somit wird die gepackte Co-Matrix gezielt generiert, ohne eine volle Co-Matrix im Speicher halten zu müssen. Nachdem ein Grauwertepaar gebildet wurde, werden dessen Grauwerte mit der Grauwert-Indextabelle durch Indices der gepackten Co-Matrix ersetzt. Das identifizierte Element der gepackten Co-Matrix wird um eins erhöht, ähnlich wie bei einem Histogramm.
Bei der Adressierung einer Datenreihe im Grafikkartenspeicher gibt es für effiziente Speicherzugriffe die Bedingung, dass jede Zeilenstartadresse ein vielfaches von 256 sein muss. Bei Matrizen, die aus mehreren Zeilen bestehen, kann am Zeilenende eine Lücke mit unbenutzten Elementen entstehen. Die Implementierung berücksichtigt Lücken am Zeilenende.

4.3.2.4. Normalisierte Co-Matrix

Die Co-Matrizen müssen für die Bildmerkmalsberechnung normalisiert sein, indem jedes Element durch die Matrixsumme dividiert wird. Da die Division eine zeitaufwendige Operation ist, multipliziert eine Kernelfunktion alle Matrixelemente mit dem Kehrwert der Matrixsumme. Die Ausführungszeit wird effizienter, weil nur noch eine Division berechnet werden muss.

4.3.2.5. Merkmale erzielen durch Aufsummieren

Die Gleichungen der Bildmerkmale (4.1 - 4.13) sowie die Gleichungen der Definitionen (4.14 - 4.21) bestehen hauptsächlich aus Summen. Gleichzeitig verfügt jede Kernelfunktion (Tabelle 4.2) über eine ressourcenangepasste Anzahl an parallel arbeitenden Threads, die für die Summenbildung genutzt werden. Die Quelldaten werden blockweise gelesen, mit arithmetischen Operationen (Multiplikation, Logarithmus, und andere) verrechnet und zu den bereits prozessierten Daten im *Shared*-Speicher parallel addiert. Ergebnis ist ein Block mit Zwischensummen, der mit einer parallelen Reduktion zu einer einzelnen Summe addiert wird. Die Blockgröße wird definiert durch die Anzahl vorhandener Threads. Es gibt zwei Implementierungen der Blockstruktur, eine Vektor-Block-Struktur, bei der eine Untermenge der Quellzeile eingelesen wird und eine Matrix-Block-Struktur, bei der ein Ausschnitt aus der Quellmatrix eingelesen wird. Aufgrund eines besseren Verhaltens beim Datentransport wird für die Bildmerkmale f5, f6 und f3 die Matrix-Block-Struktur zur Summenbildung verwendet.

4.3.2.6. Index abhängige Merkmal Gleichungen

Ein Blick auf die Gleichungen der Bildmerkmale und der Definitionen zeigt, dass einige den Index mit in die Berechnung einbeziehen. Gemeint ist, dass der Grauwert entsprechend der Position in der vollen Co-Matrix mit einbezogen wird. Indexabhängige Gleichungen sind (4.3), (4.4), (4.5), (4.14), (4.15), (4.17) und (4.18), die die Bildmerkmale $f3$, $f4$, $f5$ und die Definitionen P_{x+y}, $P_{|x-y|}$, *mean* und *var* berechnen. Aufgrund der gepackten Co-Matrix entsteht für die Berechnung der indexabhängigen Gleichungen ein Mehraufwand, da ein zusätzlicher Lesezugriff auch die Lookup Tabellen für die Grauwertbestimmung notwendig ist. Um den Geschwindigkeitseinbruch klein zu halten, wird ein Texturencache für die Lesezugriffe auf die Lookup Tabelle verwendet.

4.3.2.7. Zwischenergebnisvektor Px+y(k)

Der Zwischenergebnisvektor $P_{x+y}(k)$ mit $k = 2,3,..2Ng-2$ und $Ng = 4096$ hat eine Größe von 8188 Elementen, zu groß für den *Shared*-Speicher mit 4096 Elementen (16kBytes / 4Bytes pro Element). Falls doch der gesamte *Shared*-Speicher verwendet wird, könnte aufgrund der Knappheit nur ein Thread pro Stream-Multiprozessor laufen (siehe Kapitel 2.1.2).
Zur Bestimmung der effizientesten Implementierung wurde die Kernelfunktion 1F (siehe Tabelle 4.2) mehrfach in CUDA implementiert. In der ersten Implementierung wurde der Zwischenergebnisvektor unterteilt, so dass jeder Abschnitt in den *Shared*-Speicher passt. Eine weitere Implementierung lagert den Zwischenergebnisvektor in den größeren lokalen Speicher aus. Die optimale Implementierung liest blockweise eine Matrixzeile, berechnet die Indizierung des Zwischenergebnisvektors und summiert das entsprechende Element. Die Blockgröße ist gleich der Anzahl vorhandener Threads, d.h. ein Block wird gleichzeitig gelesen und verarbeitet. Die Index-Berechnung basiert auf der Indizierung der gelesenen Daten i und j, zu denen die Grauwerte I und J aus der Lookup Tabelle bestimmt werden. Die Summe aus I und J indiziert das Element des Zwischenergebnisvektors, auf den die zuvor gelesenen Daten addiert werden.

4.3.2.8. Zwischenergebnisvektor Px-y(k)

Die Größe des Zwischenergebnisvektors $P|x-y|(k)$ beträgt nur 4096 Elemente mit $k = 0,1,2..Ng-2$ und $Ng = 4096$. Auch dieser Zwischenergebnisvektor ist zu groß, er würde den *Shared*-Speicher komplett ausfüllen und aufgrund des mangelnden Speichers keine weiteren Threads zur Ausführung zulassen.
Für die Implementierung ist die gleiche Strategie optimal, wie sie auch beim Zwischenergebnisvektor P_{x+y} genutzt wird. Der Unterschied liegt in einer etwas höheren Komplexität. Nach-

dem die Grauwerte I und J zu den gelesenen Daten der Position i und j nachgeschaut wurden, müssen die Grauwerte subtrahiert und der Absolutwert gebildet werden ($k = |I - J|$). Der neu gewonnene Index (k) ist die Position im Zwischenergebnisvektor $P|x-y|(k)$, an dem die zuvor gelesenen Daten summiert werden.

4.3.2.9. Test und Kontrollimplementierung

Neben den Funktionen, die den Algorithmus widerspiegeln, gibt es weitere für Testzwecke und für die Ein/Ausgabe. Die Aufgaben der zusätzlichen implementierten Funktionen lauten:

- die Laufzeitmessung der einzelnen Kernelfunktionen,
- die Berechnung der Datenraten auf den Grafikkartenspeicher (in GBytes/s),
- die Berechnung der Rechengeschwindigkeit (in GFLOPS),
- das Zählen des Speicherbedarfs auf der GPU, um einen Überlauf zu erkennen,
- die Ergebnisausgabe in eine Datei,
- eine Bildschirmausgabe als Fortschrittsanzeige,
- die Fehlersuche, um den Inhalt des Grafikkartenspeichers zu kopieren und darzustellen.

4.3.3. Profiling

Mit Hilfe des Profilers konnte aus der bisherigen GPU Version eine schnellere GPU Version II implementiert werden. Engpässe im Quellcode konnten erkannt, sowie weitere Stellen für Optimierungen identifiziert werden. Viele verzweigende Ausführungswege und Synchronisationspunkte konnten mit Änderungen in der Struktur optimiert und beschleunigt werden.

Speicherzugriffe auf die Matrixzeilen erfolgen gleichzeitig und blockweise, entsprechend der *Warp*-Größe. Oft ist die Blockgröße kein Vielfaches der Matrixzeilengröße, so dass am Zeilenende Verzweigungen für die Threads innerhalb eines Blocks entstehen, manche Threads lesen und andere warten. Die Verzweigungen führen in der SIMT Architektur zu einer unvermeidlichen seriellen Ausführung aller Zweige. Damit keine Threads über das Zeilenende hinaus arbeiten, ist es notwendig zu prüfen, ob der aktuelle CUDA-Block mit seinen Berechnungen bereits an den Zeilenenden der Matrix angekommen ist. Diese prüfende `if`-Anweisung konnte für einen allgemeinen Fall weg optimiert werden. Im allgemeinen Fall werden die Matrixzeilen ohne `if`-Anweisung gelesen, ohne den Teil am Zeilenende zu lesen. Der danach ausgeführte Grenzfall beinhaltet die `if`-Anweisung für die verzweigenden Threads am Zeilenende. Diese strukturelle Änderung beschleunigt das Leseverhalten der Matrixzeilen, weil im häufigeren allgemeinen Fall keine Verzweigung mehr existiert, die ausgewertet werden muss. Dieses Konzept wurde in vielen Kernelfunktionen umgesetzt.

Eine weitere beschleunigende Strukturänderung liegt darin, auf den *Shared*-Speicher zu verzichten. Der übliche Weg, GPU Daten zu prozessieren ist, sie aus dem Grafikkartenspeicher in den *Shared*-Speicher zu lesen, anschließend zu synchronisieren, Operationen auf dem *Shared*-Speicher anwenden, wieder zu synchronisieren und am Ende die prozessierten Daten in den Grafikkartenspeicher zu kopieren. In der Implementierung der GPU Version II werden in den meisten Kernelfunktionen die Operationen direkt auf dem Grafikkartenspeicher angewandt. Lesen, Ausführen der Operationen und Speichern der prozessierten Daten geschieht ohne den *Shared*-Speicher zu verwenden. Synchronisation wurde hinfällig, was zu einer beschleunigten Ausführung geführt hat. Auch die Caches, die in den neueren GPU-Generationen vorhanden sind, tragen dazu bei, den *Shared*-Speicher nicht mehr als Datenpuffer einsetzen zu müssen.

Mit den gezeigten Maßnahmen reduzieren sich die Ausführungszeiten im Durchschnitt um den Faktor 1,4. Eine detaillierte Auflistung der Kernelfunktionen mit den Ausführungszeiten, mit und ohne Optimierungen, ist im Kapitel Ergebnisse 6.1 auf Seite 111 zu finden. An dieser Stelle ist ebenso ein Geschwindigkeitsvergleich zur CPU und das Skalierungsverhalten mit der erreichten Rechenleistung in GFLOPS dargestellt.

5. OpenCL zu FPGA Übersetzer

5.1. Konzept

Der beschriebene Ansatz für einen OpenCL FPGA Übersetzer aus Kapitel 3.4.5 wird hier verfeinert und gegenüber Alternativen abgewogen. Einen FPGA einzusetzen ist notwendig, weil eine FPGA-Karte einen Datendurchsatz von mehreren hundert MBytes/s aufnehmen kann. Die Sensoren werden über differentiale Signale (LVDS) mit dem FPGA verbunden. Andere übliche Schnittstellen wie zum Beispiel USB oder Ethernet bieten bei weitem nicht genug Bandbreite und deren Protokolle werden von den Sensoren nicht unterstützt.
Die FPGA-Karte soll neben der Datenaufnahme (*data acquisition*, DAC) ebenso die gestreamten Daten online prozessieren. Eine Anpassung des Hardware-Designs mit neuen Algorithmen ist in der Regel nicht trivial, besonders für Programmierer ohne Hardwarekenntnisse. Aus diesem Grund wurde im Ansatz die Sprache OpenCL ausgewählt, die als einfache Sprache zur Beschreibung der Hardware verwendet werden soll. Die Sprache OpenCL bietet in Hinblick auf einen FPGA-Übersetzer mehrere Vorteile gegenüber anderes Sprachen:

- OpenCL ist eine parallele Programmiersprache. Sie gibt eine Modell vor, die Parallelität in einem FPGA umzusetzen. Serielle Programmiersprachen haben diesen Vorteil nicht und es ist schwierig, die Parallelität aus einem seriellen Programm zu extrahieren.

- Der Hauptvorteil von OpenCL ist die vielseitige Einsatzmöglichkeit. Mit ihr können unterschiedliche Architekturen programmiert werden, wie z.B. GPUs, CPUs und DSPs. Die parallele Programmiersprache CUDA ist speziell für GPUs von NVIDIA entwickelt worden. Man könnte einen CUDA-Kompilierer entwickeln, der Programme für FPGAs übersetzt. FCUDA ist beispielsweise eine solche Entwicklung aus der Zeit, als es noch kein OpenCL gab. Warum FCUDA nicht weiter geeignet ist, wird im „Fazit der FPGA-Sprachen“ erläutert (3.4.5). Es liegt allerdings näher, einen Kompilierer für OpenCL zu entwickeln als für CUDA, da OpenCL einem offenen Standard unterliegt. Es gibt einen standardisierten und gut dokumentierten Funktionssatz, was bei CUDA nicht der Fall ist.

- Weiter ist OpenCL (oder CUDA), eine weit verbreitete Sprache, die von vielen Programmierern akzeptiert wurde. Das macht eine Einarbeitung in die Sprache nützlich, da es mehrere Anwendungsgebiete gibt. Sprachen, die mit Sprachkonstrukten erweitert werden müssen, haben es schwerer, sich durchzusetzen.

Die weiteren Gedanken lehnen daran an, wie der Kompilierer OpenCL in ein FPGA-Design umsetzt. Die OpenRCL Entwicklung simuliert eine GPU-ähnliche Architektur im FPGA, für die ein Programm übersetzt werden kann. Der Ansatz dieser Arbeit sieht vor, einen Pipelinegenerator zu entwickeln, der das Programm in eine Hardwarepipeline übersetzt. Somit werden die Ressourcen zielgerichtet für das OpenCL-Programm eingesetzt und das Pipelinekonzept verspricht eine hohe und effiziente Auslastung der synthetisieren Hardware.

5.2. Übersicht

Die entwickelten Bestandteile des Übersetzer-Konzepts sind miteinander verwoben. Aus diesem Grund wird hier eine Übersicht in Bild 5.1 gegeben, die nur im Groben die Bestandteile beschreibt und deren Interaktionen mit Pfeilen darlegt.

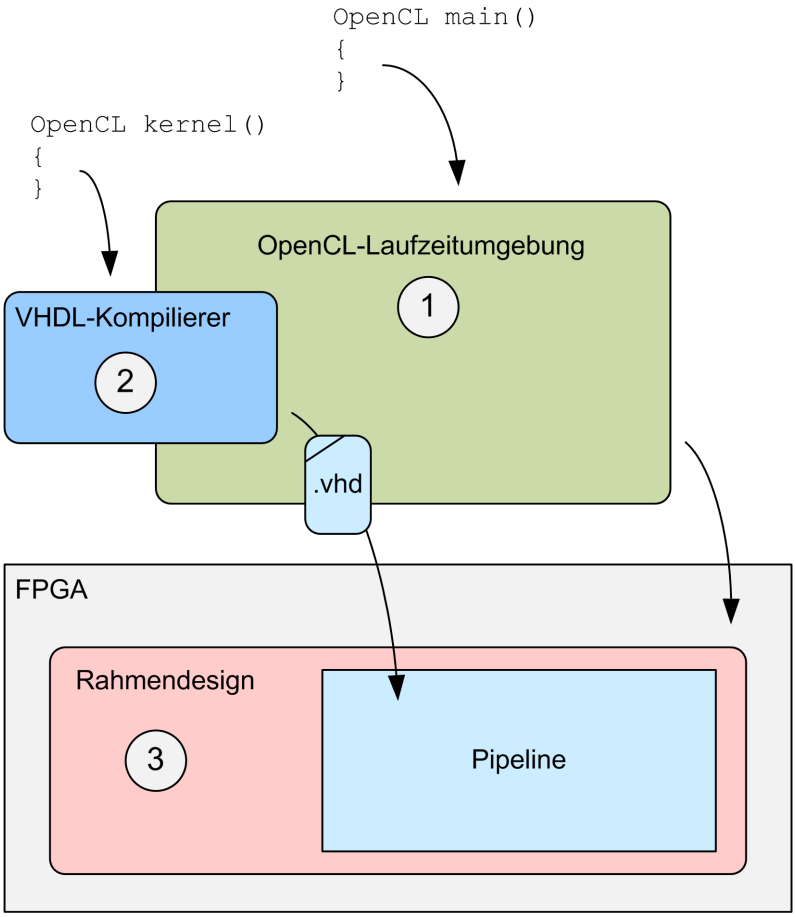


Bild 5.1.: Bestandteile für einen OpenCL-FPGA Übersetzer

1. Die **OpenCL-Laufzeitumgebung** ist eine Bibliothek mit Funktionen aus dem OpenCL-Standard [36], siehe Bild 2.4 aus dem Grundlagenkapitel. In der Bibliothek ist ein Kompilierer integriert, der Kernelfunktionen, in unserem Fall, für den FPGA übersetzt. Allgemein verwaltet die Laufzeitumgebung die Übersetzung und die Ausführung der Kernelfunktionen, die Speichertransfers und die Registerzugriffe auf den FPGA.

2. Der **VHDL-Kompilierer** besteht aus drei Programmen (Clang-Frontend, LLVM-IR-Optimierer und VHDL-Backend), die zusammen eine Übersetzungskette bilden. In ihr wird eine OpenCL-Kernelfunktion in eine VHDL-Pipeline übersetzt.

3. Das **Rahmendesign** bietet für die generierte VHDL-Pipeline einen Rahmen mit umliegender Logik (Speicherkontroller, PCIe-Core mit DMA-Engine und Kontrolllogik). Erst mit ihr kann die Pipeline mit Daten gefüllt und eine Berechnung gestartet werden.

Der nächste Abschnitt 5.3 beginnt mit dem VHDL-Kompilierer und beschreibt die Glieder der Übersetzungskette sowie deren Implementierung. Im anschließenden Abschnitt 5.4 wird gezeigt, welche Logik im Rahmendesign für eine Ausführung notwendig ist und welche Implementierungsansätze gewählt worden sind. Erst im letzten Abschnitt 5.5, nachdem das Rahmendesign bekannt ist, wird die Implementierung der Softwareschnittstellen mit den OpenCL-Funktionen beschrieben.

5.3. VHDL-Kompilierer

5.3.1. Übersetzungskette

Es gibt viele Ansätze, wie der OpenCL-Quelltext übersetzt werden kann. Beispielsweise lässt sich mit Lex und Yacc ein OpenCL-Frontend entwickeln, das den Quelltext in einen Parserbaum im Speicher umsetzt. Diese Aufgabe benötigt viel Entwicklungszeit. Statt dessen könnte man das C-Frontend von GCC verwenden, dann bräuchte man keinen C-Parser entwickeln und man könnte die Entwicklung auf die Zwischensprache aufsetzen. Da das GCC C-Frontend in die Jahre gekommen ist, wurde es mit LLVM überarbeitet und erneuert. Auch diese Arbeit setzt auf LLVM auf. Das LLVM C-Frontend (Clang) übersetzt in eine standardisierte Zwischensprache, für die es in dieser Arbeit einen Parser und ein Backend für den FPGA zu entwickeln gilt. Die Übersetzungskette lässt sich in drei Glieder aufteilen, die in Bild 5.2 angeordnet sind.

Das Clang-Frontend wird für die Übersetzung der Kernelfunktion in LLVM-IR verwendet. Besser wäre es, ein OpenCL-Frontend zu verwenden, da die Kernelfunktion OpenCL-spezifische Schlüsselworte enthalten kann, die von Clang nicht identifiziert werden. Beispielsweise spezifizieren die Schlüsselworte `local` und `global` die Speicherbereiche, in denen die Variablen

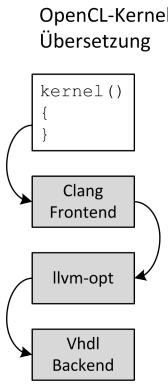


Bild 5.2.: Übersetzungskette des VHDL-Kompilierers

abgelegt werden. Folglich kann das Clang-Frontend keine Variable im *Shared*-Speicher hinterlegen. Diese Einschränkung hat Einfluss auf die Fähigkeiten des Kompilierers. Daten können zwischen den Threads nur über den globalen Speicher ausgetauscht werden.

Ursprünglich war geplant, ein OpenCL-Frontend zu verwenden. Zu Beginn der Entwicklung war bekannt, dass OpenCL-Entwickler daran arbeiten, Clang für OpenCL-Kernelfunktionen zu erweitern. Heute ist bekannt, dass Apple und NVIDIA ein OpenCL-Frontend basierend auf Clang und LLVM verwenden [17], um Quelltext in OpenCL für ihre Plattformen zu übersetzen. Wir hatten darauf spekuliert, dass das OpenCL-Frontend mit der Zeit veröffentlicht wird, um es als Ersatz zu Clang verwenden zu können, was bisher nicht geschehen ist. Clang in dieser Arbeit für OpenCL Kernelfunktion abzuändern, war zeitlich keine Option, denn der Fokus dieser Arbeit liegt auf dem neuartigen VHDL-Backend.

Ein Vorteil, LLVM-IR zu verwenden, ist es, den umfangreichen Optimierer für diese Zwischensprache nutzen zu können. Bild 5.3 zeigt ein Beispiel, wie die Kernelfunktion vom Clang-Frontend in die Zwischensprache LLVM-IR übersetzt wird und danach die Anzahl der Instruktionen vom LLVM-Optimierer reduziert wird. Ohne Optimierer beträgt der LLVM-IR-Quelltext über 40 Zeilen, mit lediglich 12, was einer Reduktion von 70% entspricht. Die Kommandozeilen-Dokumentation des Optimierers zeigt, welche Verfahren für die Quelltext-Reduktion existieren. Der Optimierer wird wegen seiner intensiven Nutzung gut gepflegt, d.h. die Verfahren werden stets verbessert bzw. neue hinzugefügt.

```
__kernel void matrixAdd(int* A, int* B, int* C, int w)
{
    int idx = get_global_id(0);
    int idy = get_global_id(1);

    C[idy * w + idx] = A[idy * w + idx] + B[idy * w + idx];
}
```

./clang

```
define void @matrixAdd(i32* %A, i32* %B,
                       i32* %C, i32 %w) nounwind {
  %1 = alloca i32*, align 4
  %2 = alloca i32*, align 4
  %3 = alloca i32*, align 4
  %4 = alloca i32, align 4
  %idx = alloca i32, align 4
  %idy = alloca i32, align 4
  store i32* %A, i32** %1, align 4
  store i32* %B, i32** %2, align 4
  store i32* %C, i32** %3, align 4
  store i32 %w, i32* %4, align 4
  %5 = call i32 @get_global_id(i32 0)
  store i32 %5, i32* %idx, align 4
  %6 = call i32 @get_global_id(i32 1)
  store i32 %6, i32* %idy, align 4
  %7 = load i32* %idy, align 4
  %8 = load i32* %4, align 4
  %9 = mul nsw i32 %7, %8
  %10 = load i32* %idx, align 4
  %11 = add nsw i32 %9, %10
  %12 = load i32** %1, align 4
  %13 = getelementptr inbounds i32* %12, i32 %11
  %14 = load i32* %13
  %15 = load i32* %idy, align 4
  %16 = load i32* %4, align 4
  %17 = mul nsw i32 %15, %16
  %18 = load i32* %idx, align 4
  %19 = add nsw i32 %17, %18
  %20 = load i32** %2, align 4
  %21 = getelementptr inbounds i32* %20, i32 %19
  %22 = load i32* %21
  %23 = add nsw i32 %14, %22
  %24 = load i32* %idy, align 4
  %25 = load i32* %4, align 4
  %26 = mul nsw i32 %24, %25
  %27 = load i32* %idx, align 4
  %28 = add nsw i32 %26, %27
  %29 = load i32** %3, align 4
  %30 = getelementptr inbounds i32* %29, i32 %28
  store i32 %23, i32* %30
  ret void
}
```

./llvm-opt

```
define void @matrixAdd(i32* nocapture %A, i32* nocapture %B,
                       i32* nocapture %C, i32 %w) nounwind {
  %1 = tail call i32 @get_global_id(i32 0) nounwind
  %2 = tail call i32 @get_global_id(i32 1) nounwind
  %3 = mul nsw i32 %2, %w
  %4 = add nsw i32 %3, %1
  %5 = getelementptr inbounds i32* %A, i32 %4
  %6 = load i32* %5, align 4
  %7 = getelementptr inbounds i32* %B, i32 %4
  %8 = load i32* %7, align 4
  %9 = add nsw i32 %8, %6
  %10 = getelementptr inbounds i32* %C, i32 %4
  store i32 %9, i32* %10, align 4
  ret void
}
```

Bild 5.3.: Übersetzung der Kernelfunktion `matrixAdd` in die Zwischensprache LLVM-IR

5.3.2. Softwarearchitektur VHDL-Backend

In einem Blockschaubild 5.4 wird die Struktur des VHDL-Backend in funktionale Blöcke gegliedert und anschließend erläutert. Weiter wird mit UML (*Unified Modeling Language*), einer grafischen Modellierungssprache, die Softwarearchitektur des Übersetzers in Klassendiagrammen visualisiert.

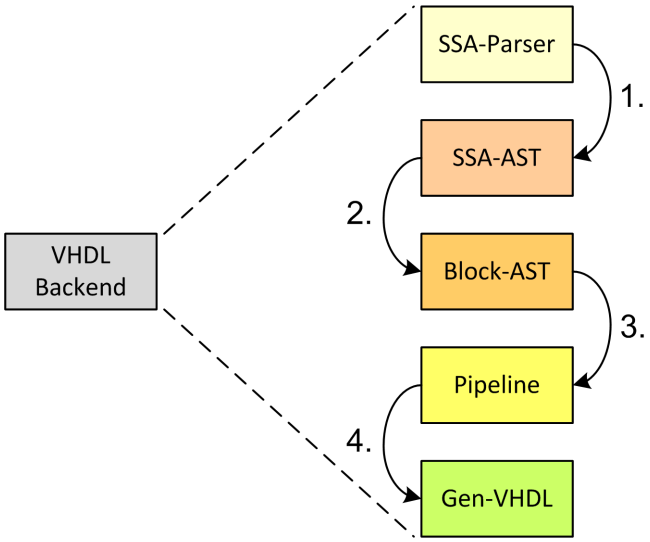


Bild 5.4.: Die vier Übersetzungschritte des VHDL-Backends

1. Ein SSA-Parser liest den optimierten LLVM-IR-SSA-Quelltext einer OpenCL-Kernelfunktion ein und generiert aus den Instruktionen einen Syntaxbaum (AST), hier SSA-AST genannt. Wie SSA mit LLVM zusammenhängt, war in 3.3.2 erwähnt. Man hätte auch die Speicherrepräsentation von LLVM als AST verwenden können. Mit einem LLVM-Programm lässt sich der LLVM-AST aus dem LLVM-IR-SSA-Quelltext generieren. Einen Parser für LLVM-IR-SSA zu entwickeln, ist einfach, weil die assemblerähnlichen Instruktionen zeilenweise zu lesen und zu interpretieren sind. Den komplexeren LLVM-AST zu verwenden, weil er schon existiert, war nicht Grund genug für dessen Verwendung. Weiter kann der eigene Parser gezielt für die VHDL-Übersetzung entwickelt werden. Eine Änderung des LLVM-AST für VHDL wäre weniger einfach gewesen.

2. Aus dem SSA-AST wird ein weiterer Syntaxbaum, der Block-AST, generiert. Die Knoten des neuen Syntaxbaums sind keine Instruktionen mehr, sondern Blöcke aus einem Baukastensystem für Hardware-Pipelines. Für einige Instruktionen ist die Abbildung auf die Blöcke direkt möglich. Hierbei zahlt es sich aus, auf SSA-Quelltext gesetzt zu haben, da dieser einen unendlicher Registervorrat besitzt, vergleichbar mit Signalen im

VHDL-Quelltext. Jedes Ergebnis wird in einem eindeutigen Register hinterlegt (`%5 = add i32 %0, %3`), entsprechend gibt es für jedes Ergebnis in VHDL ein eindeutiges Signal (`s5 <= s0 + s3`).

3. Die Pipeline enthält weitere Logik, die neben dem Block-AST existieren muss. In diesem Schritt werden die Latenzzeiten berechnet und eventuelle Verzögerungsglieder hinzugefügt. Ebenso müssen die Verbindungsglieder zum Rahmendesign konfiguriert werden.

4. Im letzten Schritt wird aus allen gesammelten und berechneten Informationen eine VHDL-Datei mit der Hardware-Pipeline generiert.

Das UML C++ Klassendiagramm in Bild 5.5 zeigt die Softwarestruktur des VHDL-Backends. Das Konzept baut auf Klassen mit Kompositionsbeziehungen auf, die farbig markiert sind. Jede Klasse steht in einer existenzabhängigen Teil-Von-Beziehung zur darüber liegenden, d.h. jede Klasse hält eine Referenz der abhängigen Klasse, die beim Konstruktoraufruf mit angegeben werden musste. Aus dieser Struktur ergibt sich die Reihenfolge, in der die Instanzen angelegt werden müssen. Ebenso ergibt sich die Reihenfolge der Methodenausführung.

Die Instanz von `Source` hält lediglich den SSA-Quelltext und übernimmt die Aufgaben der lexikalischen Analyse, den Quelltext in Tokens zu zerlegen. Die existenzabhängige Instanz von `SourceCheck` prüft im Quelltext, ob genau eine Kernelfunktion enthalten ist, die Kernelfunktion keinen Rückgabewert hat, dafür Parameter besitzt und ob die Syntax des Funktionsrumpfes keine Fehler aufweist. Wenn die Prüfung fehl schlägt, ist es garantiert, dass die folgenden Instanzen ihre Aufgaben nicht ausführen können und der Übersetzungsvorgang bricht an dieser Stelle ab. Die nächste Instanz, `SourceAttrib`, extrahiert Attribute aus dem Quelltext. Ein Attribut ist die Parameterliste der Funktion, die Parameterpaare listet. Für jedes Paar wird der Name und der Typ eines Parameters festgestellt und in einer Instanzvariablen gespeichert. Weitere Attribute, die herausgefunden werden, betreffen die Quelltextzeilen, in welcher der Funktionskopf, die erste und die letzte SSA-Instruktion zu finden ist. Mit den Zeileninformationen kann die `SourceParser`-Instanz zeilenweise die Instruktionen lesen und einen SSA-AST aus `Instruction`-Objekten generieren (Abschnitt 5.3.3). Der `SourceAnalyser` beschäftigt sich mit der Semantik des Programms und analysiert Programmteile wie z.B. welche Adressierungsart verwendet wird (Abschnitt 5.3.4). Mit der Analyse sind alle notwendigen Informationen zusammengetragen, um den Block-AST aus dem SSA-AST zu generieren (Abschnitt 5.3.5). Die Instanz `VhdlArchitecture` benutzt den Block-AST, bestehend aus `VhdlBlock`-Instanzen, den aufbereiteten Informationen der Analyse, und generiert daraus eine VHDL-Datei (Abschnitt 5.3.6).

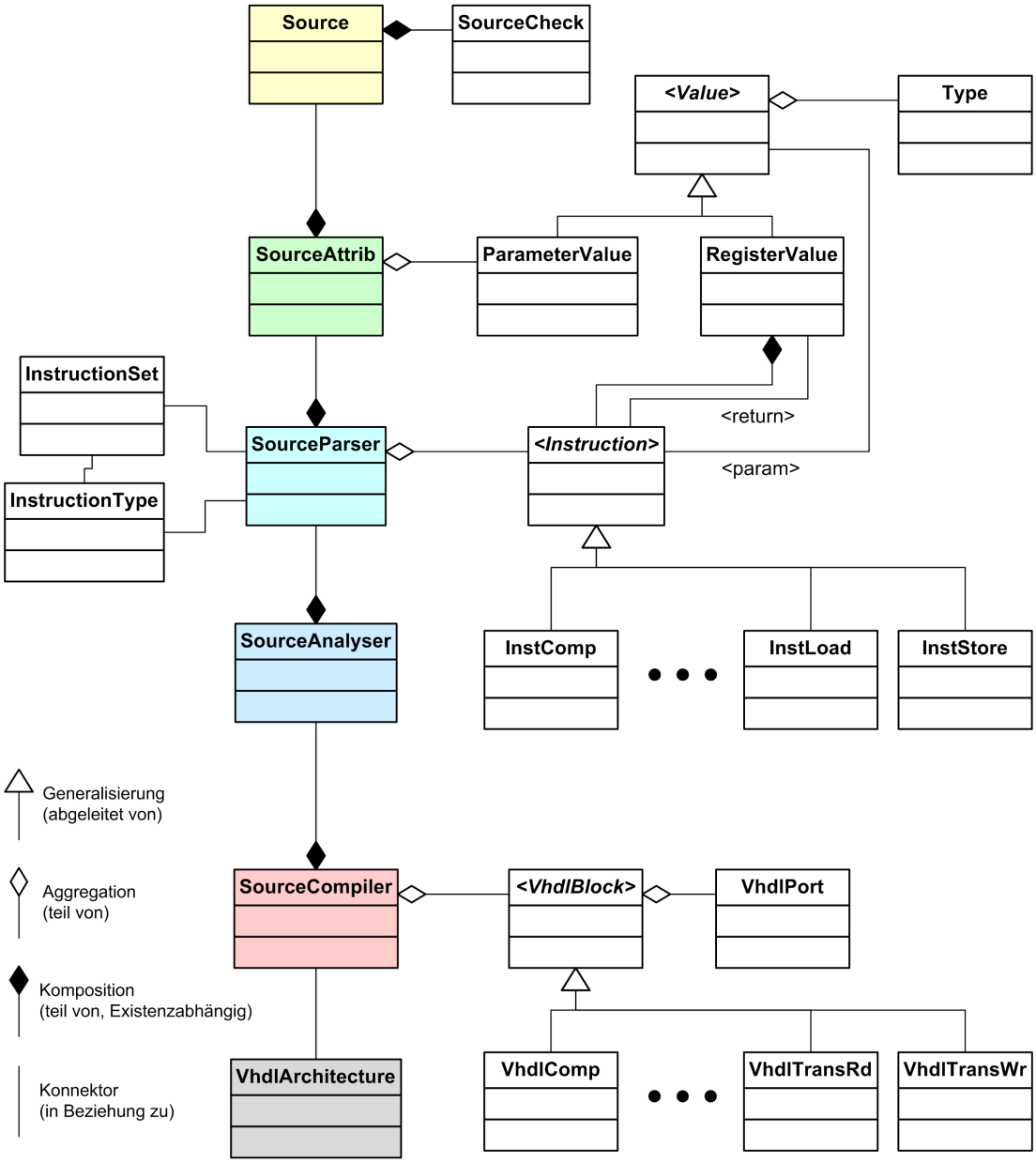


Bild 5.5.: Klassendiagramm des VHDL-Backends

In den folgenden Abschnitten wird die Implementierung der Übersetzungsschritte 1 bis 4 erläutert und anhand von Beispielen verdeutlicht.

5.3.3. Parsebaum Generierung

Dieser Abschnitt entspricht dem Übersetzungsschritt 1 aus dem Blockschaubild 5.4. Bild 5.6 zeigt die Klasse `SourceParser` mit allen nötigen Klassen, um den SSA-AST zu generieren. Das Parsen geschieht in der Methode `parse()` in einer Schleife, die jede Zeile des SSA-Quelltextes mit drei Hilfsmethoden auswertet.

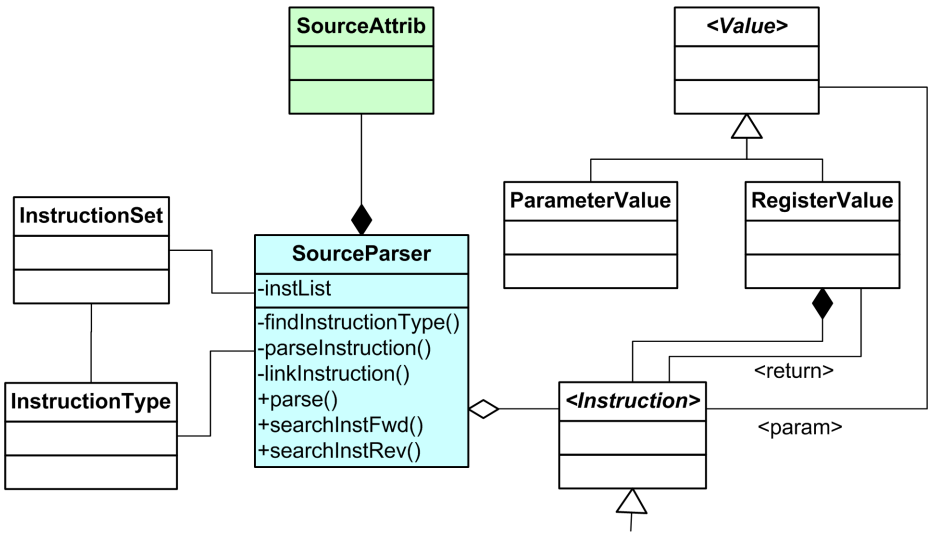


Bild 5.6.: Klassendiagramm des Parsers mit unvollständiger Bezeichnung der Instanzvariablen und der Methoden

`findInstructionType()` identifiziert die Instruktion der Quelltextzeile und ordnet sie einer Kategorienummer zu. In der Klasse `InstructionSet` sind alle Instruktionen als `string` gespeichert, in der die Methode nach dem entsprechenden Eintrag suchen kann. Die Klasse `InstructionType` enthält die Zuordnung der Kategorienummer, eine Art Enumerationstyp, mit der sich die Instruktion besser identifizieren und vergleichen lässt, als mit einem `string`.
`parseInstruction()` erstellt anhand der Kategorienummer eine spezialisierte Instanz von der abstrakten Klasse `Instruction` (im Bild nicht dargestellt). Das Parsen, also das Auswerten der Quelltextzeile, passiert im Konstruktor der abgeleiteten Klasse. Jede Instruktion hat ihre eigene Parametrisierung und somit auch ihren eigenen Parseralgorithmus, der die Anzahl der Parameter und zusätzlich verwendete Schlüsselworte kennt. Die meisten Instruktionen speichern ein Ergebnis in einem neuen Register ab. Für jedes Ergebnisregister wird eine Instanz von `RegisterValue` angelegt, die vom `Value` abgeleitet ist. Die `Value`-Instanzen sind die Zweige

im AST, die die Instruktionsknoten verbinden.
`linkInstruction()` findet im AST die Quellregister, die von der aktuellen Instruktion verwendet werden und verlinkt sie mit entsprechenden Zeigern. Ein Quellregister kann entweder das Ergebnisregister (`RegisterValue`) einer Instruktion oder ein Parameterregister (`ParameterValue`) aus dem Funktionskopf sein, dessen Instanzen die `SourceAttrib`-Klasse angelegt hatte. Wegen der zwei möglichen Quellen wird auf die abstrakte Basisklasse `Value` verwiesen.
Die `SourceParser`-Klasse bietet viele Suchfunktionen, Einträge aufsteigend (`searchInstFwd()`) oder absteigend (`searchInstRev()`) im SSA-AST nach allen möglichen Suchkriterien zu suchen. Die Softwarearchitektur ist modular angelegt, damit jederzeit weitere Befehle durch Ableiten der Basisklasse in den Funktionsumfang aufgenommen werden können.

Spezialisierung	**LLVM-IR-Befehl**	**Bedeutung**
InstCall	`call`	Funktionsaufruf, zur Bestimmung der Threadindizes der OpenCL-Kernelfunktion
InstCmp	`icmp`	Vergleich mit parametrisiertem Operator
InstComp	`add`	Addition
	`sub`	Subtraktion
	`mul`	Multiplikation
	`udiv`	vorzeichenlose Division
	`sdiv`	Division
	`urem`	vorzeichenlose Modulo
	`srem`	Modulo
	`shl`	links Schieben
	`lshr`	rechts Schieben, MSB aufgefüllt mit Nullen
	`and`	Und
	`or`	Oder
	`xor`	exklusives Oder
InstGetElemPtr	`getelementptr`	Operator zur Adressberechnung von Vektoren und Matrizen
InstSelect	`select`	bedingte Zuweisung
InstTransfer	`load`	lesender Speicherzugriff
	`store`	schreibender Speicherzugriff

Tabelle 5.1.: Abgeleitete Instruktions-Klassen, die LLVM-Befehle repräsentieren, und kurze Beschreibung

Im *LLVM Language Reference Manual* [49] sind alle LLVM-IR-Befehle erläutert. Tabelle 5.1 listet die `Instruction`-Spezialisierungen auf und zeigt, welche LLVM-IR-Befehle sie abdecken

können.

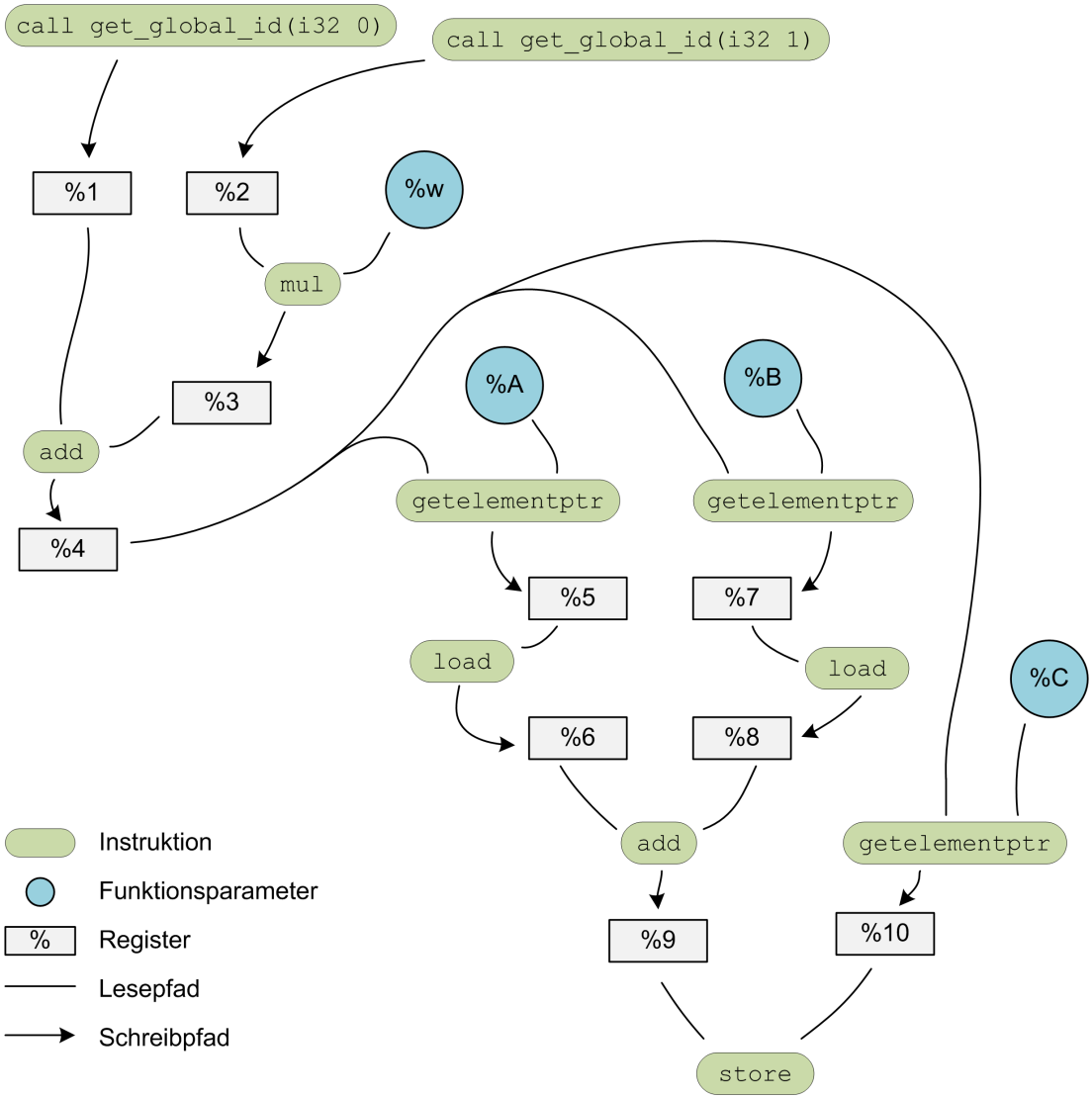


Bild 5.7.: Beispiel des generierten SSA-AST der Kernelfunktion `matrixAdd`

Als Beisliel der SSA-AST-Generierung wird wieder die Kernelfunktion `matrixAdd` aufgegriffen. Aus dem optimierten LLVM-IR-Quelltext entsteht das Speicherabbild des Syntaxbaums, der mit dem Bild 5.7 veranschaulicht wird.

5.3.4. Parsebaum Analyse

Der `SourceAnalyser` benötigt ein `SourceParser`-Objekt, das bereits den SSA-AST generiert hat. Bild 5.8 zeigt, dass nur die Methode `parse()` öffentlich ist. Sie ruft intern die privaten Methoden in entsprechender Reihenfolge auf, siehe Bild 5.8.

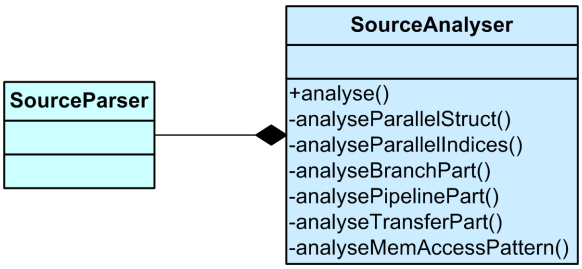


Bild 5.8.: Auszug Klassendiagramm um die Analyseklasse.

`analyseParallelIndices()` untersucht, welche parallele Indizierung im Quelltext vorhanden ist. Das Untersuchungsergebnis ist, welche Dimensionierung die Parallelindizes benutzen. Die Analysemethode sucht im Parsebaum nach `call`-Instruktion, hinter der sich folgende Funktionsaufrufe [35] befinden können:

- `uint get_work_dim()` Gibt die Anzahl benutzter Dimensionen zurück.
- `size_t get_global_size(uint D)` Dimensionsgröße der globalen *work-items*.
- `size_t get_global_id(uint D)` Globale *work-item* Nummer der entsprechenden Dimension.
- `size_t get_local_size(uint D)` Dimensionsgröße der lokalen *work-items*.
- `size_t get_local_id(uint D)` Lokale *work-item* Nummer der entsprechenden Dimension.
- `size_t get_num_groups(uint D)` Dimensionsgröße der *work-groups*.
- `size_t get_group_id(uint D)` Nummer der *work-group* der entsprechenden Dimension.

Mit der Abfrage des Parameters `D` kann die Dimension der entsprechenden Gruppe ausgelesen werden. Die Dimensionierung gibt Hinweise auf das Zugriffsmuster der Speichertransfers, z.B. das Lesen eines Arrays, einer Matrix oder eines Volumens.

Die Methode **`analyseParallelStruct()`** ist ohne Implementierung. Sie ist ein Platzhalter und gewinnt an Bedeutung, wenn es ein OpenCL-Frontend gibt, das *Shared*-Speicher unterstützt.

Sie sollte analysieren, wie der *Shared*-Speicher genutzt wird, und man könnte diese Gebrauchsmöglichkeiten unterscheiden: Der *Shared*-Speicher

1. wird nicht benutzt,
2. wird als Puffer für Speicherzugriffe auf den Hauptspeicher benutzt,
3. wird verwendet, um Daten darin zu sortieren, um lineare Hauptspeicherzugriffe zu erzielen,
4. wird für die Threadkommunikation innerhalb einer *work-group* benutzt.

Auch ohne den *Shared*-Speicher unterstützt der jetzige Kompilierer den zweiten Punkt, weil in der Pipeline Datenpuffer existieren, um eine kontinuierliche Datenversorgung aufrecht zu erhalten.

`analyseBranchPart()` identifiziert mögliche Verzweigungen im Quelltext:

- `if-else`-Verzweigung. Sie ist bisher nicht implementiert. Wenn sie implementiert wird, müssen der `if`-Zweig und der `else`-Zweig gleiche Latenzzeiten (gleich viele Instruktionen) besitzen. Bei ungleichen Latenzen muss der kürzere Zweig mit zusätzlichen Pipelinestufen erweitert werden. Im Prototyp wird für die Demonstration lediglich der bedingte Zuweisungs-Operator implementiert, siehe nächster Punkt.
- Bedingte Zuweisung. Sie ist eine Spezialform einer `if-else`-Verzweigung und entspricht dem „`? :` Operator“ aus C++. Je nachdem, ob ein boolescher Ausdruck „wahr“ oder „falsch“ ist, wird entweder der eine oder der andere Wert als Ergebnis übernommen. In dieser Implementierung darf der boolesche Ausdruck lediglich eine Vergleichsoperation mit einer Konstanten sein. Für eine Demonstration genügt die Annahme einer Konstante, wodurch bereits viele Beispielprogramme übersetzungsfähig werden. Die Konstante lässt sich in einer Weiterentwicklung durch eine Variable ersetzen, indem der Parser angepasst wird.

`analysePipelinePart()` und `analyseTransferPart()` weisen alle arithmetischen Operatoren der Pipeline oder des Datentransfers zu. Die Pipeline soll nur die Operationen enthalten, die zur Ergebnisberechnung beitragen. Die Operationen, die zur Berechnung der Adresse benötigt werden, beispielsweise die Indexberechnung eines Elements in einem Array $Index = Basis + i$, soll in einer speziellen Komponente des Pipeline-Baukastens passieren. Wie die beiden Analysemethoden die Zuordnung herausfinden, wird am Beispiel einer Matrixaddition verdeutlicht. Die Kernelfunktion besteht im wesentlichen aus der Zeile $C[idy * w + idx] = A[idy * w + idx] + B[idy * w + idx]$. Der daraus entstehende SSA-AST ist in Bild 5.9 dargestellt.

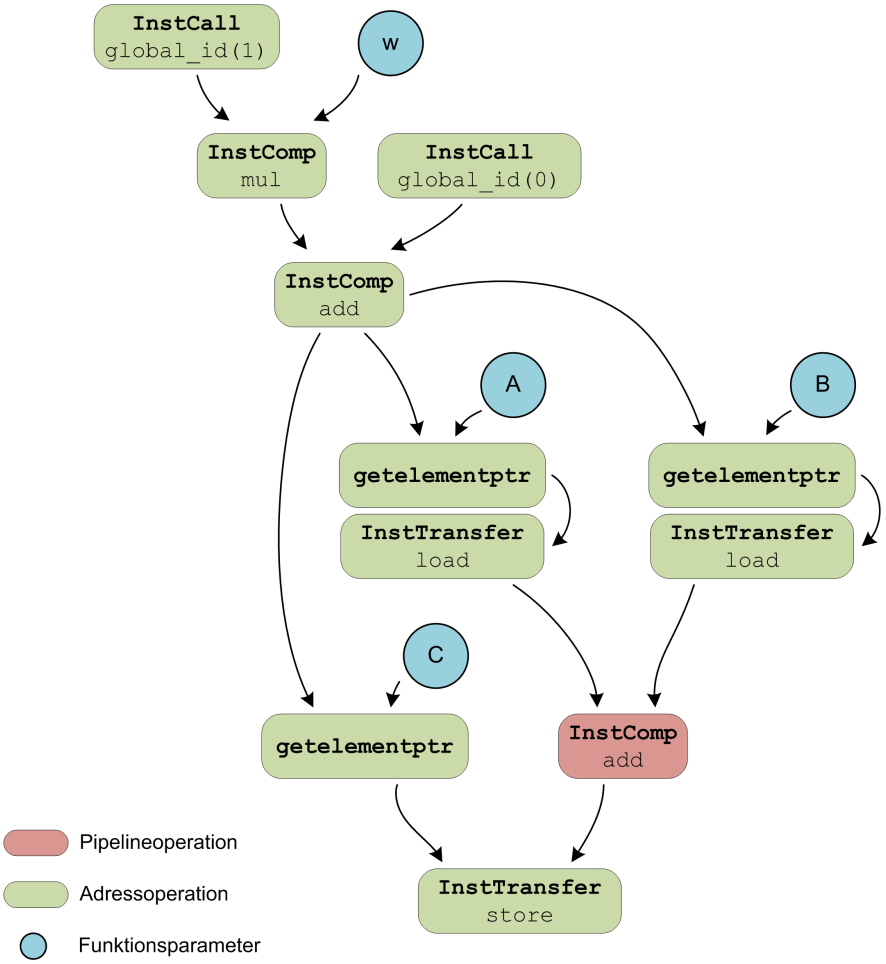


Bild 5.9.: Vereinfachter SSA-AST der Matrixaddition mit zugeordneten Operationen

Da jedes Programm Daten aus dem Speicher liest, darauf rechnet und Ergebnisse wiederum schreibt, befinden sich die Rechenoperationen der Pipeline zwischen den `load`-Operationen und den `store`-Operationen. Im Bild ist das lediglich die rote `add`-Operation, die nacheinander alle Elemente der Matrix addiert. Die `analysePipelinePart()`-Methode durchsucht den Parsebaum beginnend bei den InstTransfer-`load`-Operationen „vorwärts - in Pfeilrichtung“ zu den InstTransfer-`store`-Operationen. Mit rekursiven Aufrufen werden alle Verzweigungen durchlaufen und alle identifizierten Rechenoperationen als Teil der Pipeline, im Bild mit rot, markiert.

Die `load`- und die `store`-Operation benötigt die Angabe, an welcher Adresse gelesen bzw. geschrieben werden soll. Die Adresse wird mit der Instruktion `getelemntptr` und eventuell weiteren arithmetischen Operationen berechnet, die markierte werden sollen. Die Markierung hilft, die arithmetischen Operationen zur Berechnung der Adresse mit denen aus der Pipeline zu unterscheiden. Die unterschiedlich markierten arithmetischen Operationen werden spä-

ter in unterschiedlichen VHDL-Komponenten berechnet. Die Methode `analyseTransferPart()` durchläuft den Parsebaum beginnend bei den InstTransfer-`load`-Operationen „rückwärts“ zu den Kernel-Funktionsparametern, im Bild zu den blauen Kreisen. Beim Durchlaufen werden alle Äste nummeriert und als Transferteil, im Bild mit grün, markiert. Zuletzt muss beginnend bei den InstTransfer-`store`-Operation dessen Transferteil „rückwärts“ durchlaufen werden. Wenn eine Operation gefunden wird, die bereits der Pipeline zugeordnet wurde, bricht die weitere Suche auf diesem Zweig ab.

`analyseMemAccessPattern()` ist ein Platzhalter für die nächste Version, um die statischen bzw. variablen Zugriffsmuster auf dem Speicher zu analysieren. Im Prototyp wird diese Methode nicht benötigt, da die Anforderung existiert, dass ausschließlich statische Speicherzugriffe im Quelltext erlaubt sind.

5.3.5. Parsebaum Übersetzung

Dieser Abschnitt entspricht dem Übersetzungsschritt 2 aus dem Blockschaubild 5.4.

5.3.5.1. SSA-AST zu Block-AST

Nach der Analyse kann eine Instanz der Klasse `SourceCompiler` angelegt werden, siehe Bild 5.10. Die öffentliche Methode `compile()` generiert aus dem SSA-AST einen neuen Block-AST, der aus Knoten besteht, die der VHDL-Pipeline entsprechen. Dabei wird der SSA-AST Knoten für Knoten durchlaufen, um den neuen Baum entstehen zu lassen.

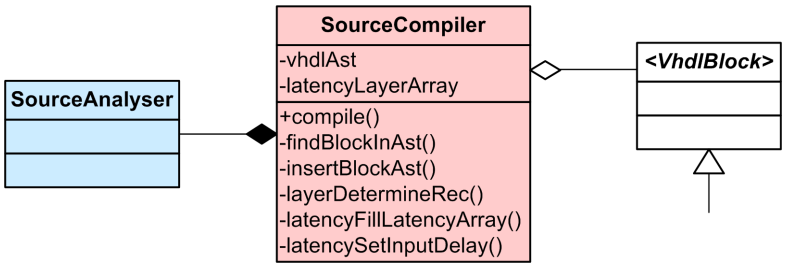


Bild 5.10.: Auszug Klassendiagramm um die Compilerklasse

Mit einer Übersetzungstabelle, siehe nächster Abschnitt, wird auf jeden Knoten des SSA-ASTs eine Vorschrift angewendet, wie ein entsprechender Knoten im Block-AST zu generieren ist. Dieser Vorgang ist in private Hilfsmethoden unterteilt worden, die von `compile()` aufgerufen werden. Im privaten Bereich, unter der Instanzvariable `vhdlAst` wird der Block-AST entstehen. Die Hilfsmethode `findBlockInAst()` sucht im Block-AST eine bestimmte Stelle, an der

die Hilfsmethode `insertBlockAst()` einen neuen Block einfügen kann.

Die Knoten des Block-ASTs sind Spezialisierungen der abstrakten Klasse `VhdlBlock`. Die Abgeleiteten Klassen werden im nächsten Abschnitt vorgestellt.

5.3.5.2. Zuordnung Instruktion zum VHDL-Block

Der Pipeline-Baukasten enthält fünf VHDL-Blöcke, aus denen die Pipeline zusammengesetzt wird. Mit wenigen Blöcken auszukommen hat den Vorteil, dass ein Zusammensetzen wenig komplex ist. D.h. es gibt nicht viele Möglichkeiten, die Blöcke miteinander zu verbinden. Der entstehende Nachteil ist, dass viel Funktionalität aus der OpenCL-Kernelfunktion auf wenig Blöcke verteilt werden muss. Diese Arbeit verfolgt den Ansatz, die Funktionalität der VHDL-Blöcke für Standardfälle auszulegen und zu optimieren. Die Tabelle 5.2 geht in der ersten Spalte von den Instruktion-Klasseninstanzen aus, die in der zweiten Spalte auf VHDL-Blöcke widergespiegelt werden.

Klassenname	**VHDL-Block**	**Kommentar**
InstCall	-	Wird nicht in einen VHDL-Block übersetzt.
InstCmp InstSelect	BlockCondAssign	Beide Instruktionen werden kombiniert und in einen VHDL-Block übersetzt.
InstComp	BlockComp	Wird mit einer Eins-zu-Eins Beziehung übersetzt. Die SSA-Instruktion parametrisiert die VHDL-Komponente.
InstGetElemPtr	-	Die Berechnung der Adresse passiert in der BlockAddr-Komponente, kombiniert mit der Analyse der parallelen Struktur.
InstTransfer	BlockAddr	Diese Komponente ist immer an eine BlockTransfer-Komponente gekoppelt, je nachdem welche Instruktion gemeint ist.
	BlockTransRd	Wird generiert wenn die Instruktion ein `load` ist.
	BlockTransWr	Wird generiert wenn die Instruktion ein `store` ist.

Tabelle 5.2.: Übersetzungstabelle vom SSA-AST zum Block-AST

Die OpenCL-Kernelfunktion wird von vielen Threads durchlaufen, genauso soll die Pipeline von vielen Datenelementen gestreamt werden. Folglich existiert eine Verbindung zwischen dem *n*-ten Datenelement mit dem *n*-ten Thread aus dem *work-items*-Vorrat. Für die Rechenpi-

peline ist es nicht wichtig zu wissen, von wie vielen Datenelementen (Threads) sie durchlaufen wird. Aus diesem Grund wird die `InstCall`-Klasse für den Block-AST nicht benötigt. Auch das Zugriffsmuster auf den Speicher ist für die Rechenpipeline uninteressant. Die Komponenten BlockTransRd und BlockTransWr bieten lediglich einen lesenden bzw. schreibenden Zugang zum Speicher. In der BlockAddr-Komponente wird der Index und die Zieladresse des aktuell gestreamten Datenelements, mit der entsprechenden Threadnummer (globale *work-item*), berechnet. Sie enthält das Speicherzugriffsmuster, das vom `SourceAnalyser` analysiert wurde und die Basisadresse, die von den Kernel-Funktionsparametern stammt. Eine `InstTransfer` -Klasseninstanz wird immer in eine Kombination von BlockAddr mit BlockTransRd oder BlockTransWr umgesetzt. Die Funktion der `InstGetElemPtr`-Klasse rückt in die BlockAddr-Komponente. Eine bedingte Zuweisung besteht aus einem Vergleich und einer Auswahl der weiter gestreamten Daten, vergleichbar mit einem Multiplexer.

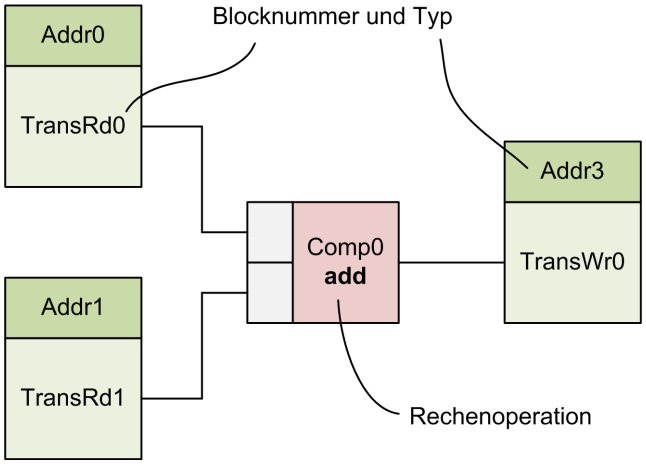


Bild 5.11.: Block-AST Beispiel der Matrixaddition

Anhand des Beispiels Matrixaddition soll gezeigt werden, wie der Block-AST aussieht. Bild 5.11 zeigt den entstandenen Pipeline Baum. Im Vergleich zum SSA-AST 5.7 auf Seite 75 ist dieser einfacher in seinem Aufbau. Das liegt daran, dass viele Instruktionen auf wenige VHDL-Blöcke verteilt wurden. Auch die Operationen zur Adressberechnung der Matrixelemente verbergen sich im BlockAddr.

5.3.5.3. Verzögerungen in der Pipeline

Beim beliebigen Zusammensetzen der VHDL-Blöcke zu einer Pipeline können Datenpfade mit unterschiedlicher Latenz entstehen. Da am Ende der Pipeline nach der Durchlaufverzögerung jeder Takt ein Ergebnis liefern soll, müssen alle Datenpfade die gleiche Durchlaufverzögerung

aufweisen. Das Problem wird in Bild 5.12 verdeutlicht.

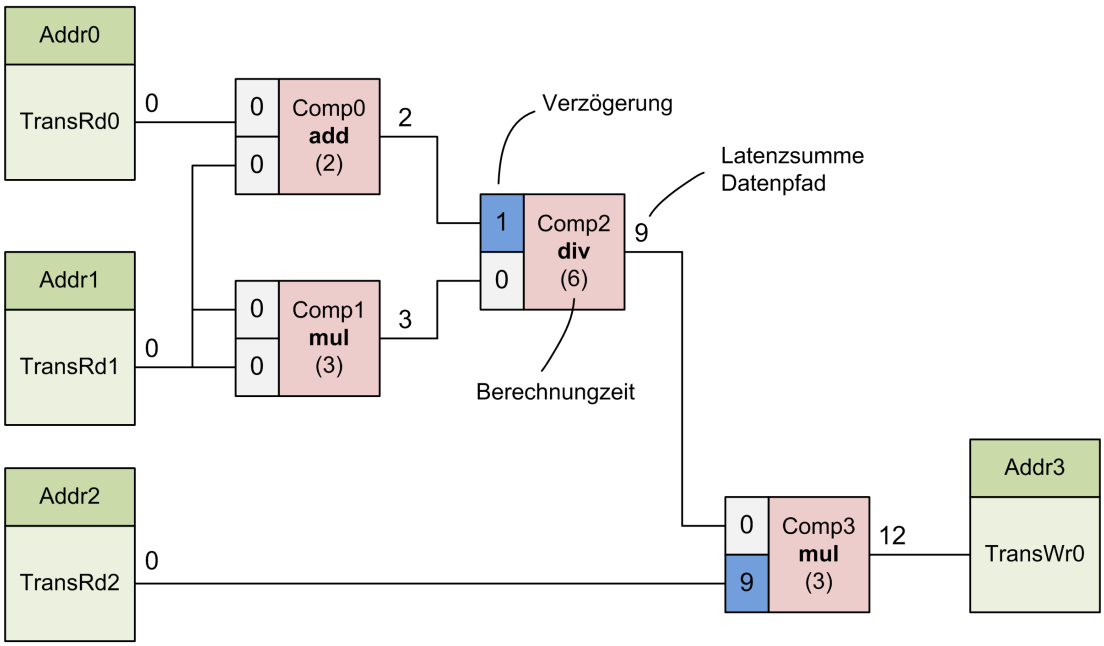


Bild 5.12.: VHDL-Pipeline mit synchronisierten Pfadlaufzeiten

Links befinden sich die nummerierten BlockTransRd- und BlockAddr-Komponenten, von denen der Datenstrom gelieft wird. An dieser Stelle haben sie in der Summe Null Taktverzögerung, zu sehen an der 0 an den Pfadausgängen der Komponenten. Die BlockComp-Komponenten 0 und 1 haben unterschiedliche Berechnungszeiten, erkennbar an der Taktzahl in den Klammern. BlockComp2 hat das Problem, dass an dessen Eingängen die Daten um einen Takt verzögert ankommen würden. Aus diesem Grund besitzen die Komponenten verzögerbare Dateneingänge, deren Daten sich um beliebige Takte verzögern lassen, in diesem Fall um einen Takt, siehe das blauen Kästchen. Nicht nur unterschiedliche Berechnungszeiten können unterschiedliche Latenzzeiten auf den Datenpfaden erzeugen, sondern auch eine verschachtelte Anordnung der VHDL-Blöcke, wie es mit dem BlockComp3 der Fall ist. Der obere Datenzweig weist bereits eine summierte Latenzzeit von neun Takten auf, wegen der erforderlichen Synchronität muss der untere Datenpfad um neun Takte verzögert werden.

Der Algorithmus, der die Latenzzeiten aller Datenpfade mit Verzögerungen synchronisiert, ist mit den Hilfsmethoden aus der `SourceCompile`-Klasse implementiert. Der Algorithmus beginnt in der Hilfsmethode `layerDetermineRec()` bei der BlockTransWr-Komponente und durchläuft den Block-AST „rückwärts“ mit rekursiven Aufrufen. Beim Durchlaufen wird jede Block-Komponente mit einer Ebenennummer versehen, auf der sie liegt, und es wird die größte Ebene

bestimmt. Im Bild liegt BlockComp3 auf der Ebene null, BlockComp2 auf der Ebene eins und die anderen auf der Ebene zwei, der größten. Die Hilfsmethode `latencyFillLatencyArray()` bestimmt für jede Ebene die größte Berechnungszeit und speichert sie in einem Array, durch die Ebene indiziert. Die Arraysumme entspricht dem Datenpfad mit der größten Latenz und die Teilsummen, Summe der Ebene 0 bis *i*, entsprechen der maximalen Latenz jeder Ebene für beliebige Datenpfade. In der letzten Hilfsmethode `latencySetInputDelay` werden für jeden Datenteilpfad die tatsächlichen Latenzzeiten für diesen Abschnitt summiert und mit den Teilsummen der entsprechenden Ebene verglichen. Die Differenz ergibt eine nötige Verzögerung der Eingänge.

5.3.6. Parsebaum VHDL-Wandlung

5.3.6.1. Generierung der VHDL-Pipeline

Dieser Abschnitt entspricht dem letzen Übersetzungsschritt 4 aus dem Blockschaubild 5.4. Übersetzungsschritt 3 wird noch erläutert. Die `VhdlArchitecture`-Klasse aus dem Bild 5.13 erzeugt aus dem Block-AST eine VHDL-Datei.

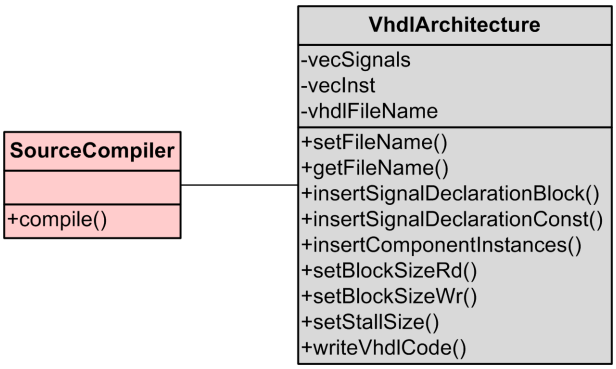


Bild 5.13.: Ausschnitt Klassendiagramm betreffend der VHDL-Generierung

Im Gegensatz zu den vorherigen Klassenbeziehungen, ist die Bindung dieser Klasse an die `SourceCompiler`-Klasse gering. Die `SourceCompiler.compile()`-Methode erzeugt eine `VhdlArchitecture`-Instanz, die ausschließlich mit `String`-Datentypen Textbausteine der Klasse hinzufügt. Ähnlich wie die `toString()`-Methoden in Javaklassen haben auch die VHDL-Block-Instanzen die Methode `getVhdlInstance()`, die eine VHDL-Textrepräsentation der Komponente zurück gibt. Der VHDL-Text jedes VHDL-Blocks im Block-AST wird über die `insertComponentInstances()` hinzugefügt. Mit der Methode `insertSignalDeclarationBlock()` werden der VHDL-Klasse alle VHDL-`Signale` mitgeteilt, die von den VHDL-Blöcken gebraucht werden. Es gibt mit der Methode `insertSignalDeclarationConst()` die Option, weitere Signale hinzuzufügen,

die von der zusätzlichen Logik, siehe folgenden Abschnitt, verwendet werden. Die gesammelten Textbausteine der Signaldeklarationen, der Blockinstanzen und der zusätzlichen Logik werden in der Methode `writeVhdlCode()` in eine VHDL-Datei geschrieben.

5.3.6.2. VHDL-Blöcke als Bausteine

Bild 5.14 zeigt den funktionalen Aufbau des Adressierungsblocks. Er hat die Aufgabe, die Adressen zu berechnen, die sich aus der Basisadresse und dem Adressoffset ergeben. Der Adressoffset wird aus den Parallelindizes hergeleitet.

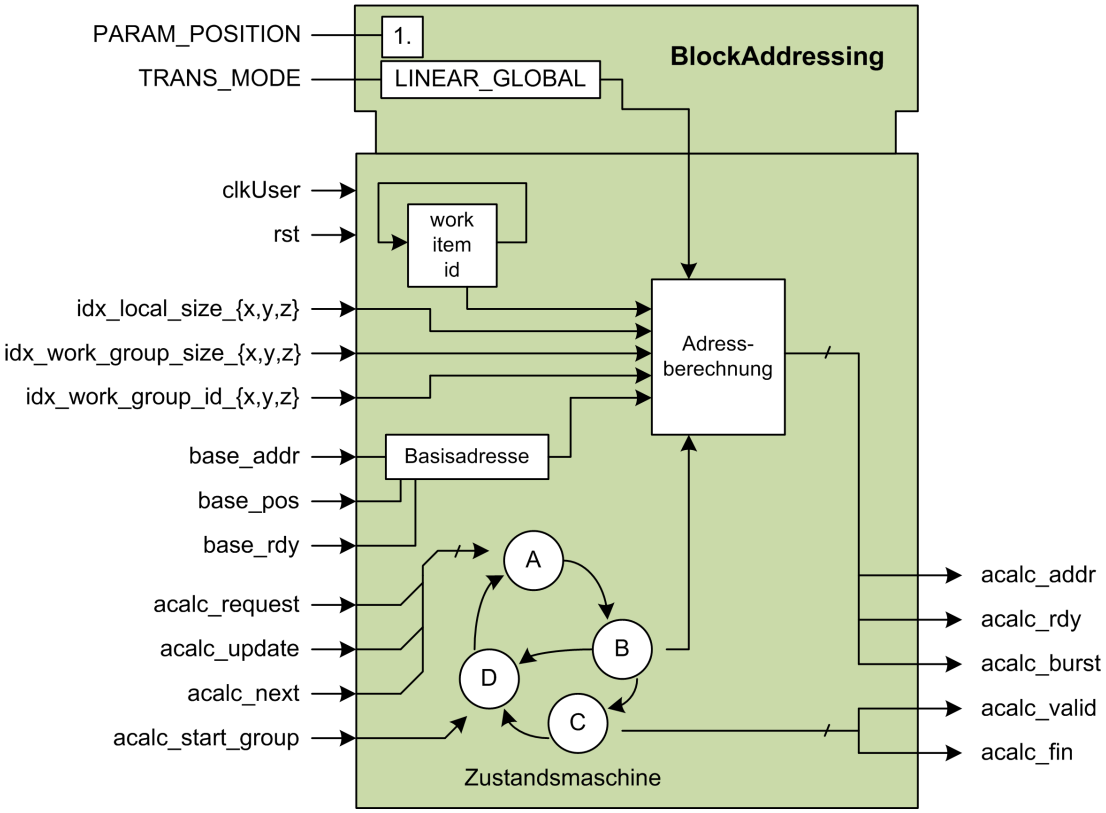


Bild 5.14.: Schematische Darstellung des VHDL-Adressblocks. Oben Generic-Ports und unten die Entity-Ports.

- Die Basisadresse ist erst zur Laufzeit bekannt und muss über die *base*-Schnittstelle mitgeteilt werden. Der Genericparameter gibt der Komponente eine eindeutige Nummer zur Identifikation.
- Die Parallelindizes werden in aufsteigender Reihenfolge abgearbeitet. In der Kompo-

nente wird der kleinste Parallelindex inkrementiert und mit den übrigen Indices der *idx*-Schnittstelle verrechnet. Zusammen mit dem Genericparameter, der das Speicherzugriffsmuster bestimmt, wird der Adressoffset bestimmt.

- Die Adressberechnung wird von einer Zustandsmaschine gesteuert. Sie regelt auch die Kommunikation mit der *acalc*-Schnittstelle, nimmt Anfragen entgegen und gibt Bescheid, wenn die Adressberechnung beendet ist.

Der funktionale Aufbau des Transferleseblocks ist in Bild 5.15 dargestellt. Dieser Block hat die Aufgabe, mit Hilfe des Adressblocks die Daten über den Speicherbus anzufordern, die als nächstes in der Pipeline benötigt werden. Die Daten werden in einem FiFo zwischengespeichert, aus dem die Pipeline taktweise Datenwerte für Berechnungen entnimmt.

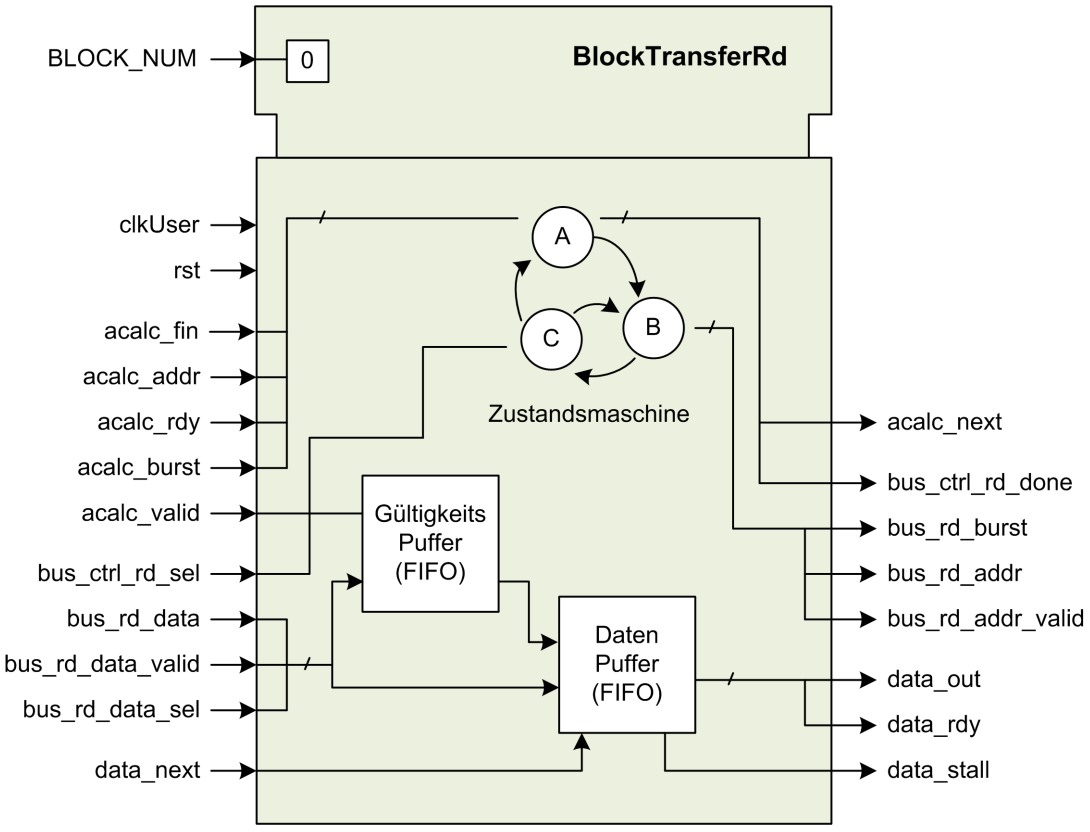


Bild 5.15.: Schematische Darstellung des VHDL-Transferleseblocks. Oben Generic-Ports und unten die Entity-Ports.

- Der Genericparameter *BLOCK_NUM* wird für den Datenbus zur Block-Identifikation benötigt.

- Die Zustandsmaschine verfolgt das Ziel, den Datenpuffer stets gefüllt zu halten. Über die *acalc*-Schnittstelle fordert sie eine Adresse an und initiiert über die *bus*-Schnittstelle einen Speicherzugriff.

- Nach einer gewissen Latenzzeit kommen die Daten, auf mehrere Takte verteilt, mit einer Wortbreite von 256-Bit an. Es kann oft vorkommen, dass nicht alle angeforderten Daten auch von der Pipeline benötigt werden. Deswegen gibt es den Gültigkeitspuffer, der alle 32-Bit-Werte als gültige oder als ungültige Daten maskiert. Ungültige Daten werden nicht in dem Datenpuffer aufgenommen.

- Die *data*-Schnittstelle ermöglicht der Pipeline, Daten aus dem Datenpuffer zu entnehmen. Das *stall*-Signal zeigt, dass der Datenpuffer leer ist und gibt der Pipeline den Befehl, die Berechnungen anzuhalten.

Der in Bild 5.16 dargestellte Transferschreibblock hat die Aufgabe, Ergebnisdaten aufzunehmen, in einem Datenpuffer zwischenzuspeichern und sie über den Bus in den Speicher zu transferieren.

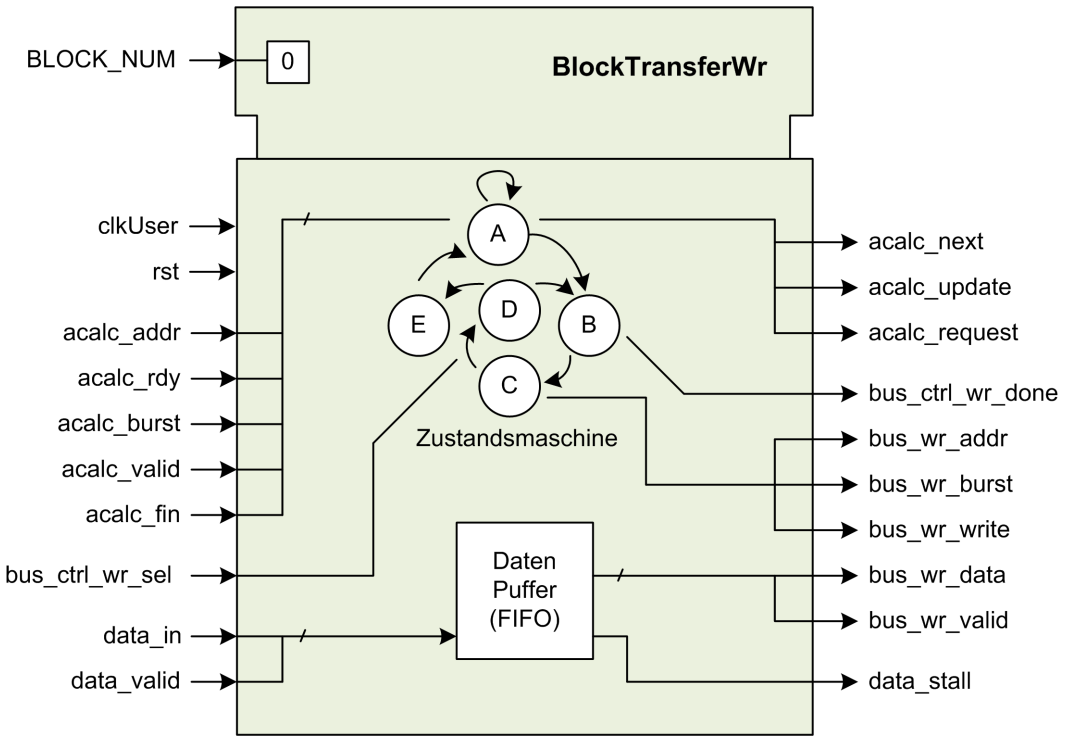


Bild 5.16.: Schematische Darstellung des VHDL-Transferschreibblocks. Oben Generic-Ports und unten die Entity-Ports.

- Die Ergebnisse kommen über die *data*-Schnittstelle im 32-Bit Format an und werden in den Datenpuffer geschrieben. Wenn er voll ist, gibt das *stall*-Signal der Pipeline die Anweisung, anzuhalten.
- Vergleichbar mit der Zustandsmaschine im Transferleseblock hat diese ebenfalls die Kontrolle über die *acalc*-Schnittstelle. Wenn der Datenpuffer Daten enthält, wird die Adresse angefordert, an der die Daten gespeichert werden sollen.
- Über die *bus*-Schnittstelle gibt die Zustandsmaschine den Schreibbefehl zusammen mit den Daten auf den Speicherbus. Der Datenpuffer ordnet die 32-Bit Ergebnisse in 256-Bit breite Datenworte um, markiert die gültigen 32-Bit Werte und füllt, wenn nötig, fehlende Werte für die Mindestmenge des Bustransfers auf.

Das Bild 5.17 zeigt die Bestandteile des Rechenblocks. Sie repräsentieren alle Rechenoperationen, die sich beliebig blockweise verschachteln lassen.

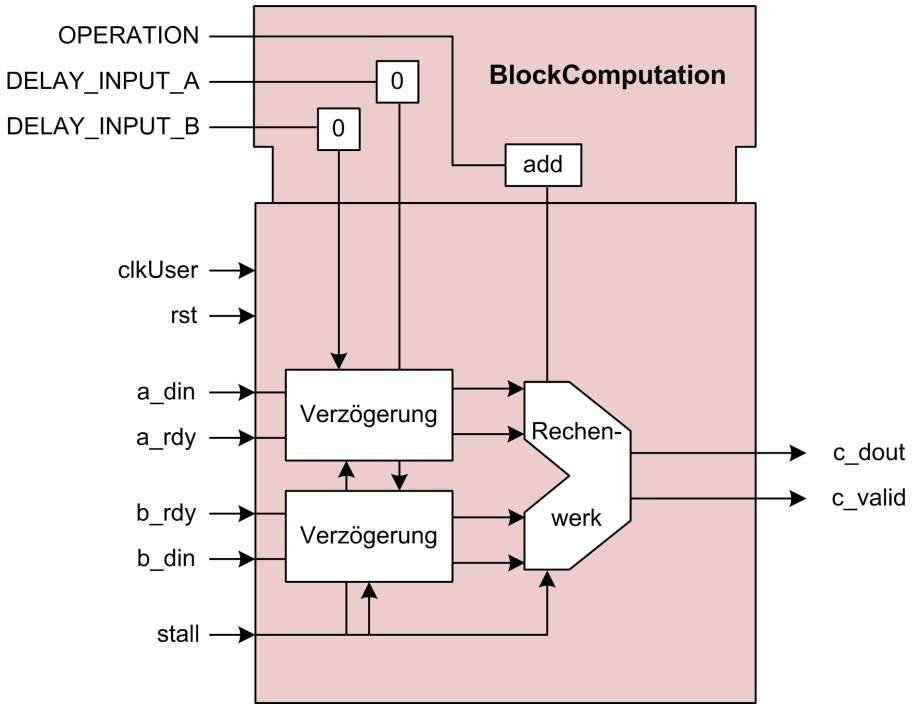


Bild 5.17.: Schematische Darstellung des VHDL-Berechnungsblocks. Oben Generic-Ports und unten die Entity-Ports.

- Welche Operation das Rechenwerk ausführen soll, wird vom Genericparameter bestimmt. Im VHDL-Quelltext gibt es für jede Rechenoperation eine Implementierung, die durch den *OPERATOR*-Parameter bestimmt wird.

- Die *DELAY_INPUT*-Parameter konfigurieren die Anzahl der Registerstufen an jedem Dateneingang, für deren Verzögerung.
- Das *stall*-Signal unterbricht die Berechnungen. D.h. alle Register halten ihren alten Wert.

Bild 5.18 zeigt den funktionalen Aufbau des VHDL-Blocks, der eine bedingte Zuweisung implementiert.

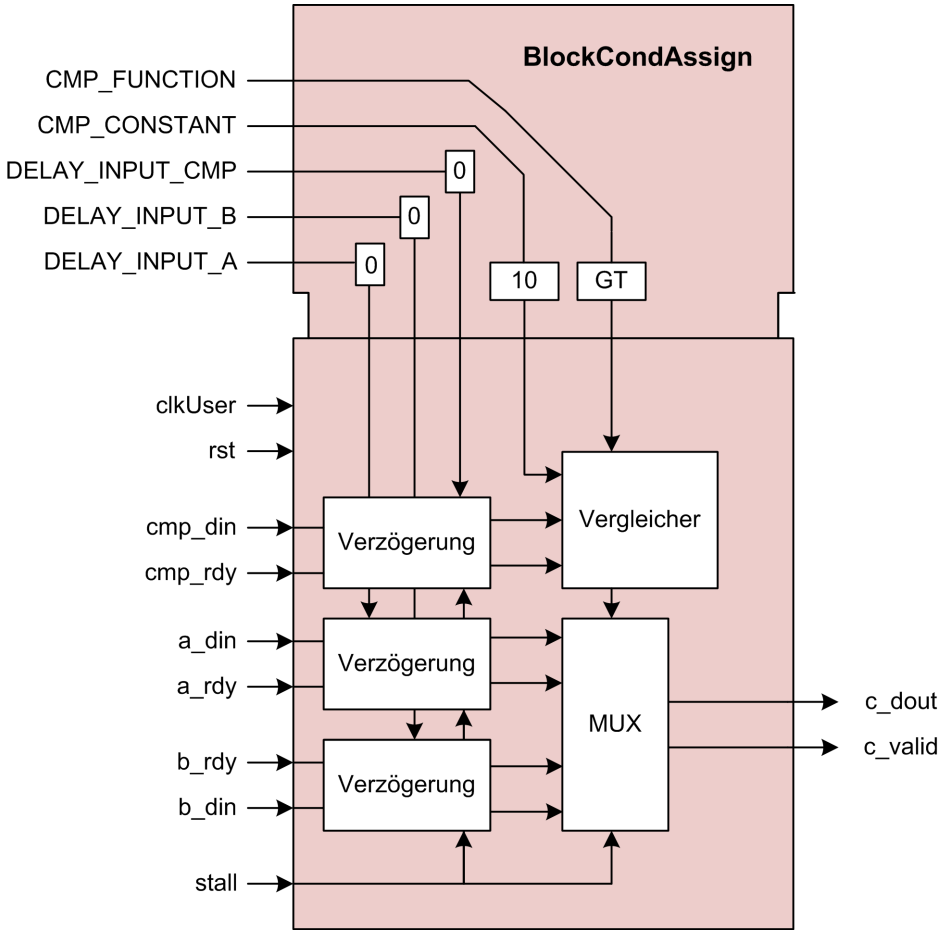


Bild 5.18.: Schematische Darstellung des VHDL-Bedingter-Zuweisungsblock. Oben Generic-Ports und unten die Entity-Ports.

- Der Genericparameter *CMP_FUNCTION* definiert die Vergleichsfunktion mit der die Daten der *cmp*-Schnittstelle mit dem konstanten Ausdruck *CMP_CONSTANT* verglichen werden. Implementierte Vergleichsfunktionen sind „gleich“, „ungleich“, „größer“, „größer gleich“, „kleiner“ und „kleiner gleich“.
- Das boolesche Vergleichsergebnis steuert einen Multiplexer an, der entweder den Datenkanal *a* oder *b* auf den Datenausgang *c* weiterleitet.

- Sämtliche Dateneingänge sind mit Verzögerungsgliedern ausgestattet, die sich mit den *DELAY*-Parametern konfigurieren lassen.

5.3.6.3. Zusätzliche Logik

Dieser Abschnitt entspricht dem fehlenden Übersetzungsschritt 3 aus dem Blockschaubild 5.4. Für die Pipeline ist zusätzliche Logik erforderlich, die mit der Übersetzung generiert werden muss. Ohne die zusätzliche Logik könnte mit der benachbarten Logik, dem Rahmendesign, keine ordnungsgemäße Funktion entstehen.

Speicherbus Oder-Logik. In der Pipeline können sich mehrere Transferblöcke befinden, die alle an den Speicherbus angeschlossen sein müssen. Es gibt mehrere Varianten, viele Quellsignale zu einem zu kombinieren. Eine Variante ist das Multiplexen, bei dem ein Kontrollsignal bestimmt, welches Quellsignal auf den Bus getrieben wird. Eine andere Variante ist die Tristatelogik, wobei alle Quellsignale elektrisch miteinander verbunden sind, aber immer nur eines senden darf, während die anderen einen hochohmigen Zustand einnehmen müssen. Die dritte Variante ist die Oderlogik, bei der alle Quellsignale mit Odergatter zu einem Signal kombiniert werden. Dabei müssen alle Quellsignale im inaktiven Zustand ein Low-Signal ausgeben und nur der aktive Teilnehmer darf beide Signalzustände verwenden. Die Oderlogik hat an dieser Stelle das beste Zeitverhalten und wurde für diese Arbeit gewählt.

Stall-Logik und Datenfluss-Kontrolle. Die Oderlogik kommt auch zum Einsatz, um die Stall-Signale zu einem zu kombinieren. Sofern kein VHDL-Block einen Datenmangel (Transferleseblock) bzw. einen Datenstau (Transferschreibblock) aufweist, kann das kombinierte Stall-Signal verwendet werden, um den Transferleseblöcken im nächsten Takt der Pipeline einen Wert zu liefern. Diese Logik steuert den Datenfluss der Pipeline.

Registerstufen im Speicherbus. Das Zeitverhalten erfordert Registerstufen zwischen dem Rahmendesign und dem Pipelinedesign, um die Speicherbussignale in beide Richtungen zeitlich zu entkoppeln. Diese Maßnahme ermöglicht den Einsatz einer dynamischen partiellen Rekonfiguration, um Pipelinemodule austauschen zu können.

Pipeline Fertigsignal. Die Pipeline muss zeigen, wann sie mit allen Berechnungen fertig ist. Dieses Signal wird dem Adressierungsblock entnommen, denn dieser kennt die Adresse vom letzten Datenelement und weiß, wann sie für einen Datentransfer verwendet wurde.

Anzahl der Transferinstanzen. Das Rahmendesign muss wissen, wie viele Instanzen von den Transferleseblöcken und den Transferschreibblöcken existieren, um sie selektieren zu können.

Pipeline Kennung. Es ist ebenso wichtig zu wissen, welche Pipeline sich gerade in der Hardware befindet. Deswegen muss jedes Design über eine ID verfügen. Eine Zufallszahl zwischen 0 und 255 ist ausreichend zufällig genug, um eine generierte Pipeline zu kennzeichnen.

5.3.7. Einschränkungen in der Übersetzung

Der Übersetzungsvorgang von einer OpenCL-Kernelfunktion in eine VHDL-Pipeline unterliegt Einschränkungen, die hier aufgelistet sind:

- Der *Shared*-Speicher kann nicht verwendet werden, deshalb ist eine Threadkommunikation nicht möglich.
- Schleifen werden nicht unterstützt. Es fehlt die Implementierung der Sprungbefehle und deren Übersetzungsvorschrift, die im Prototyp nicht vorgesehen sind.
- Komplexe `if`-Strukturen können ebenfalls wegen fehlender Sprungbefehle nicht übersetzt werden. Auch dieser Punkt ist Ziel der nächsten Version.
- Datenabhängige Speicherzugriffe und komplexere Speicherzugriffsmuster werden nicht unterstützt.
- Die Datenbreite der Pipelinearchitektur ist für 32 Bit entwickelt worden. Für andere Bitbreiten muss die Entwicklung angepasst werden.
- Bisher gibt es keine Implementierung der Fließkommaoperatoren in den Rechenblöcken.

Für zwei der Einschränkungen gibt es Behelfslösungen, die weniger effizient sind:

- Kleinere Schleifen können mit der `unroll`-Anweisung im Clang-Kompilierer entfaltet und aufgelöst werden.
- Durch die Existenz mehrerer Kernelfunktionen, die versetzt auf den Speicher zugreifen, kann eine Threadkommunikation erreicht werden.

Der bisherige Übersetzungsprozess ist in der Lage, Algorithmen zu übersetzen, die Operationen auf Werten ohne Interaktion ausführen. Ein Beispiel sind die Intensitätskorrekturen einzelner Pixel.

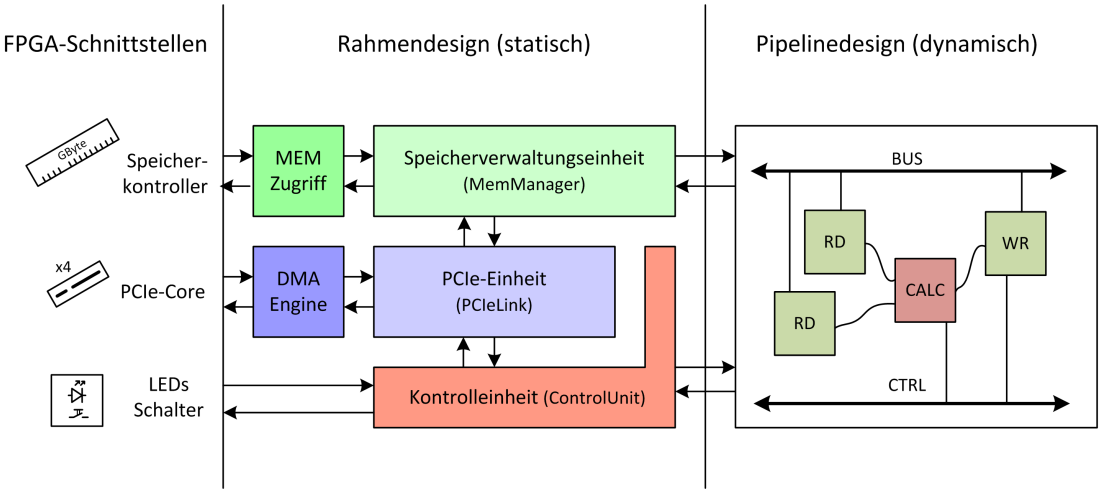


Bild 5.19.: Schnittstellen und Bestandteile des Rahmendesigns

5.4. Rahmendesign

5.4.1. Bestandteile

In der Mitte des Bildes 5.19 sind die Bestandteile des Rahmendesigns angeordnet. Rechts befindet sich die generierte Pipeline und links sind die FPGA-Schnittstellen bzw. die FPGA Xilinx-Cores, mit denen der FPGA verbunden ist. Das Rahmendesign übernimmt die Aufgabe, Knotenpunkt (zwischen Speicher, PCIe und Pipeline) zu sein und bietet der Software-Laufzeitumgebung eine Kommunikationsschnittstelle, über die sich die Pipeline steuern lässt. Die Bestandteile sind folgend weiter erläutert.

Für das Rahmendesign gibt es keine existierende Implementierung, die genutzt werden kann. Das Rahmendesign ist speziell für die FPGA-Karte und für den OpenCL-FPGA-Kompilierer zu entwickeln.

5.4.2. PCIe-Core und DMA-Engine

Mit dem PCIe-Core von Xilinx [81] wird ein Kommunikationskanal zwischen der Host-Software und dem Hardware-Design über den PCI-Express-Bus hergestellt. Mit dem Coregenerator von Xilinx wird der vorhandene Hardware-PCIe-Core auf dem Virtex6 Chip konfiguriert. Die wesentlichen Eigenschaften der Konfiguration sind:

- Der PCIe-Core wird in der Xilinx-Version 1.3 verwendet. Aktuellere Versionen laufen nicht auf der verwendeten Entwicklungssteckkarte, da dessen FPGA ein Prototyp (Engineering Sample) ist und in diesem Fall eine eingeschränkte Funktionalität besitzt.

- Es werden vier von möglichen acht Lanes verwendet.
- Für die Bus-Kommunikation wird der PCIe-Standard der Generation 1.1 zugrunde gelegt. Jede Lane hat eine maximale Übertragungsrate von 250MB/s, 1000MB/s entsprechend für vier Lanes.
- Die Schnittstelle des PCIe-Cores soll mit einer Frequenz von 100 MHz angesprochen werden.
- Der PCIe-Core benötigt eine Lizenz, die der Universität gehört.

Wenn die CPU Daten über den PCIe-Bus sendet oder liest (PIO-Mode), kann eine Datenrate in der Größenordnung von zehn Megabytes pro Sekunde erwartet werden. Aus diesem Grund wird eine DMA-Engine an den PCIe-Core geheftet, die Datenraten von bis zu 790MB/s lesend und 507MB/s schreibend unterstützt. Der Unterschied zur theoretisch maximalen Bandbreite besteht in den notwendigerweise zu übertragenden Protokollinformationen und einer nicht optimalen Ausnutzung aller Takte aufgrund von Latenzzeiten. Die DMA-Engine steht auf der Internetseite von OpenCores [29] bzw. den Institutsseiten der Universität Heidelberg [47] zum Herunterladen zur Verfügung. Details der DMA-Engine sind im CBM-Bericht [30] nachzulesen. Aus Designsicht bietet der PCIe-Core zwei einfache Benutzerschnittstellen zum Lesen und Schreiben an, die mit der PCIe-Einheit im Rahmendesign verbunden sind.

5.4.3. PCIe-Einheit

In Bild 5.20 ist der funktionale Aufbau der PCIe-Einheit, aus dem Rahmendesign, schematisch dargestellt.

- Die PCIe-Einheit verfügt über mehrere Dualport-FIFOs, die unterschiedlich getaktet sind und somit, neben der Datenspeicherung, auch die Aufgabe der Taktumsetzung haben. Die Signale von und zur DMA-Engine (clkPCIe) sind mit 125 MHz getaktet, die Signale von und zu den benachbarten Einheiten (clkUser) sind mit 133 MHz getaktet.
- Es gibt zwei DMA-Schnittstellen. Eine, um Daten aus dem FPGA zum Host zu transferieren (Lesen) und eine, die Daten vom Host zum FPGA transferiert (Schreiben). Die Signalnamen beider Schnittstellen im Bild beginnen mit dem Präfix *pcie*.
- Beim Schreib-DMA wird über die Adresse das Ziel-FIFO zum Speicher (RAM) oder zu den Registern (REG) bestimmt, und die Daten werden dahin geleitet.
- Der DMA-Lesevorgang wird von einer Zustandsmaschine gesteuert. Sie bestimmt über die Adresse, von welcher Schnittstelle (RAM oder REG) gelesen wird und wie viele

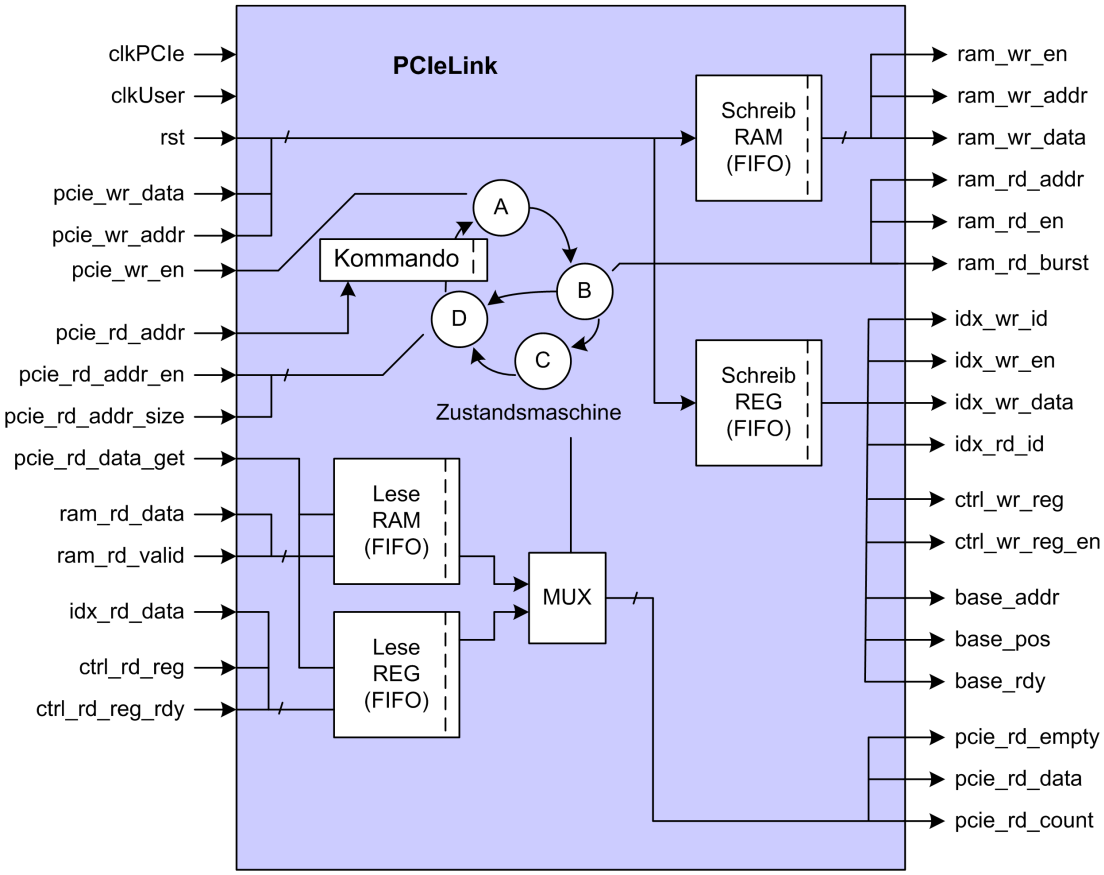


Bild 5.20.: Schematisches Blockschaltbild der PCIe-Einheit

Worte die Leselänge umfasst. Danach ist die Zustandsmaschine in der Lage, nacheinander mehrere Leseoperationen auszuführen, bis die Leselänge ausgeschöpft ist. Die Lesedaten werden von den Schnittstellen geliefert und von der Logik gezählt. Ist die entsprechende Leselänge an Worten an die DMA-Schnittstelle gesendet worden, ist die Zustandsmaschine für die nächste Lesetransaktion bereit.

- Der PCIe-Benutzer-Adressraum ist auf die Schnittstellen RAM, Index (idx), Register (ctrl) und Base aufgeteilt, um vom Host aus auf diese zugreifen zu können.
- Die *ram*-Schnittstelle hat 256 Bit breite Datenworte. Aus diesem Grund existiert im Lesepfad ein separates FIFO für den Speicher und die Register. Das Lese-RAM-FIFO setzt die Wortbreite auf 32 Bit breite Datenworte für die *pcie*-Schnittstelle um.
- Die *idx*-Schnittstelle ist Teil der Register-Schnittstelle, auf der die Parallelindizes in Register der Kontrolleinheit geschrieben und gelesen werden können.
- Die *base*-Schnittstelle ist Teil der Register-Schnittstelle. Sie existiert, um die 26 Bit brei-

ten Basisadressen in die Transferblöcke zu schreiben, da diese erst zur Laufzeit bekannt werden. Es existiert kein Lesepfad für die Basisadressen.

- Die *ctrl*-Schnittstelle ist Teil der Register-Schnittstelle. Sie führt zu einem 32 Bit breiten Kontrollregister in der Kontrolleinheit, mit einem Lese- und einem Schreibpfad.

5.4.4. Datenflusskonzept

In der Pipeline gibt es VHDL-Transfer-Blöcke, die als Dateneinspeisepunkt und als Datenausgabepunkt dienen. Das Konzept, einen Datenfluss in der Pipeline aufrecht zu erhalten, sieht vor, ein Bussystem mit dem Speicherkontroller zu verbinden, über das die Transferblöcke Zugriff zum Speicher bekommen. Wenn in der Pipeline an einem Punkt keine Daten bereitstehen, muss sie angehalten werden, bis wieder Daten geliefert wurden. Um die Pipeline am laufen zu halten, ist eine kontinuierliche Versorgung mit Daten unerlässlich. Aus diesem Grund besitzt jeder Transferblock einen Datenpuffer, vergleichbar mit kleinen Caches. Bei jedem Speicherzugriff wird ein Burstzugriff, bestehend aus 128 Werten, initiiert, um möglichst effizient viele Werte zu lesen und in den Caches abzulegen.

Ein anderer Ansatz wäre, einen großen Cache an den Datenbus anzuschließen, dafür keine Caches in den Transfer-Blöcken zu installieren, ähnlich wie bei einer CPU. Gibt es viele Transferblöcke in der Pipeline wird der Datenbus zum „Flaschenhals“, da jeder Transferblock gleichzeitig Daten aus dem Cache anfordert. Aus diesem Grund werden viele Caches eingesetzt um jedem Transferblock Zeit zu geben, Daten aus dem Speicher anzufordern, während die Caches die Pipeline mit Daten versorgen.

Der Busmaster, der den Datenbus kontrolliert, wird in der Speicherverwaltungseinheit des Rahmendesigns implementiert.

5.4.5. Speicherkontroller und vereinfachtes Ansprechen

Um Zugriff auf das 512 MByte große DDR3-Speichermodul zu bekommen, das auf der Entwicklungssteckkarte gesteckt ist, benötigt der FPGA einen Speicherkontroller. Der Virtex6 besitzt keinen Hardcore-Speicherkontroller, somit muss ein Softcore beschrieben werden. Diese Arbeit nutzt das Xilinx-MIG [83] (Memory Interface Generator), um eine Hardwarebeschreibung eines Speicherkontrollers [85] erzeugen zu lassen.

Die Benutzerschnittstelle des erzeugten Speicherkontrollers lässt sich logisch in drei Teile glie-

dern.

1. Kommando-Schnittstelle. Mit ihr kann man zusammen mit einer Adresse Lese- und Schreib-Kommandos absetzen. Gleichzeitig gibt es Signale, die mitteilen, dass der Speicherkontroller gerade keine Kommandos akzeptiert, beispielsweise wenn gerade ein Refresh-Zyklus passiert.

2. Schreib-Schnittstelle. Sie besteht aus Datenleitungen, Maskierungsleitungen und Kontrollleitungen. Die minimale Datenmenge von 512 Bit muss immer in zwei aufeinander folgenden Takten mit 256 Bit Datenworten gesendet werden. Ebenso kann die Schnittstelle mitteilen, dass sie gerade keine Daten akzeptiert, beispielsweise wenn der interne Schreibpuffer voll ist.

3. Lese-Schnittstelle. Sie besteht aus den Datenleitungen und Kontrollleitungen. Die angeforderten Daten werden, ebenso wie beim Beschreiben, als zwei aufeinander folgende 256 Bit Worte geliefert.

Für den Zweck dieser Arbeit besitzt die Benutzerschnittstelle des Speicherkontroller mehrere Nachteile:

- Die Benutzerschnittstelle arbeitet mit fest eingestellten 200MHz.
- Die Kommandoschnittstelle und die Schreibschnittstelle müssen synchronisiert angesprochen werden. Es existiert ein spezifisches Zeitverhalten beim Schreiben, was einen Schreibvorgang kompliziert gestaltet und eine Zustandsmaschine bedarf.
- Beim Lesen einer Adresse mit Byteoffset werden die Daten innerhalb der 256 Bit Worte umsortiert, was eventuell gar nicht gewünscht ist.
- Die selben Adressleitungen müssen für das Lesen und das Schreiben verwendet werden.

Eine zwischengeschaltete Zugriffs-Komponente besitzt vereinfachte Schnittstellen, logisch in zwei Teile aufgeteilt: eine zum Lesen und eine zum Schreiben. Beide habe ihre eigenen Adressleitungen und Wartesignale, die einer FIFO-Schnittstelle ähneln. Die vereinfachte Schnittstelle kann mit beliebigen Taktraten angesprochen werden. Dualport-FIFO verbinden die zwei Taktdomänen und machen den Datentransfer zwischen ihnen möglich. Intern werden die Kommandos generiert und abgesetzt, ohne dass man es an der vereinfachten Schnittstelle merkt.

5.4.6. Speicherverwaltungseinheit

Die Speicherverwaltungseinheit (Bild 5.21) aus dem Rahmendesign hat die Kontrolle über den DDR3-Speicher und bietet den anderen Schnittstellen einen Zugriff auf den Speicher an.

Die *ddr3*-Verbindung ist über die Speicherzugriffskomponente an den Speicherkontroller angeschlossen, auf die alle Speicherzugriffe geroutet werden. Die Signale zur PCIe-Einheit, um dem Host das Lesen und das Beschreiben des Speichers zu ermöglichen, haben den Präfix *dma*. Die Transferblöcke aus der Pipeline sind über einen Lesebus (*bus_rd* bzw. *bus_ctrl_rd*) und über einen Schreibbus (*bus_wr* bzw. *bus_ctrl_wr*) an die Speicherverwaltungseinheit angeschlossen.

- Die Signale *ctrl_block_start* und *end* bestimmen den Transfermodus der Speicherverwaltungseinheit. Die Signalquellen liegen in der Kontrolleinheit bzw. in der Pipeline, die darüber Auskunft geben, ob die Pipeline gerade am Rechnen ist. Es gibt zwei Transfermodi, den DMA-Transfer und den BUS-Transfer.

- Die Hauptaufgabe der Speicherverwaltungseinheit ist, den Transferblöcken und der PCIe-Einheit Speicherzugriffe zu ermöglichen. In Hardware wurde dies mit Multiplexer und Demultiplexer realisiert, kontrolliert vom Transfermodus. Im DMA-Transfermodus bekommt die PCIe-Einheit (Signale mit *dma*-Präfix) den exklusiven Zugriff auf die Speicherschnittstelle. Die Speicherzugriffe werden direkt an die Speicherschnittstelle weitergeleitet. Im BUS-Transfermodus erhalten die Transferblöcke (Signale mit *bus*-Präfix) die Kontrolle über den Speicher. Aufgrund der Einfachheit der Implementierung wurde darauf verzichtet, beiden Schnittstellen gleichzeitig Speicherzugriffe zu ermöglichen. D.h. der Host kann erst auf den Speicher zugreifen, wenn die Pipeline nicht rechnet.

- Am Lese-Bus und am Schreib-Bus können mehrere Transferblöcke, die Slaves, angeschlossen sein. Die Anzahl der an den Bussen angeschlossenen Slaves wird von den Signalen *bus_def_rd_size* und *bus_def_wr_size* aus der Pipeline, von der zusätzlichen Logik, geliefert. In der Speicherverwaltungseinheit sitzt der Bus-Master, der über die *sel*-Signale alle Slaves nacheinander selektiert, die ihrerseits mit den *done*-Signalen entweder einen Speicherzugriff anfordern oder mitteilen, dass sie keinen Bedarf eines Speicherzugriffs haben. Der Bus-Master selektiert alle Slaves abwechselnd auf dem Lese-Bus und dann auf dem Schreib-Bus, da es keine Notwendigkeit des Full-Duplex-Betriebs gibt, obwohl die Möglichkeit mit zwei getrennten Bussen bestünde.

- Wenn ein Slave des Schreib-Busses selektiert wird, kann dieser mit dem *done*-Signal die Adresse, die Daten und die Information, ob es sich um einen Burst-Zugriff handelt, senden. Das *valid*-Signal zeigt, ob der Bus für einen Transfer beansprucht wird.

- Auf dem Lese-Bus gibt es die Problematik, dass der Slave, der Daten angefordert hat, wieder identifiziert werden muss, um die Daten richtig zuzuordnen. Gelöst wird das Problem mit einem Job-Puffer und einer Zustandsmaschine, die Speicheranfragen mit der Slave-ID zwischenspeichert. Die Anfrage besteht aus der Adresse, einem Burst-Signal

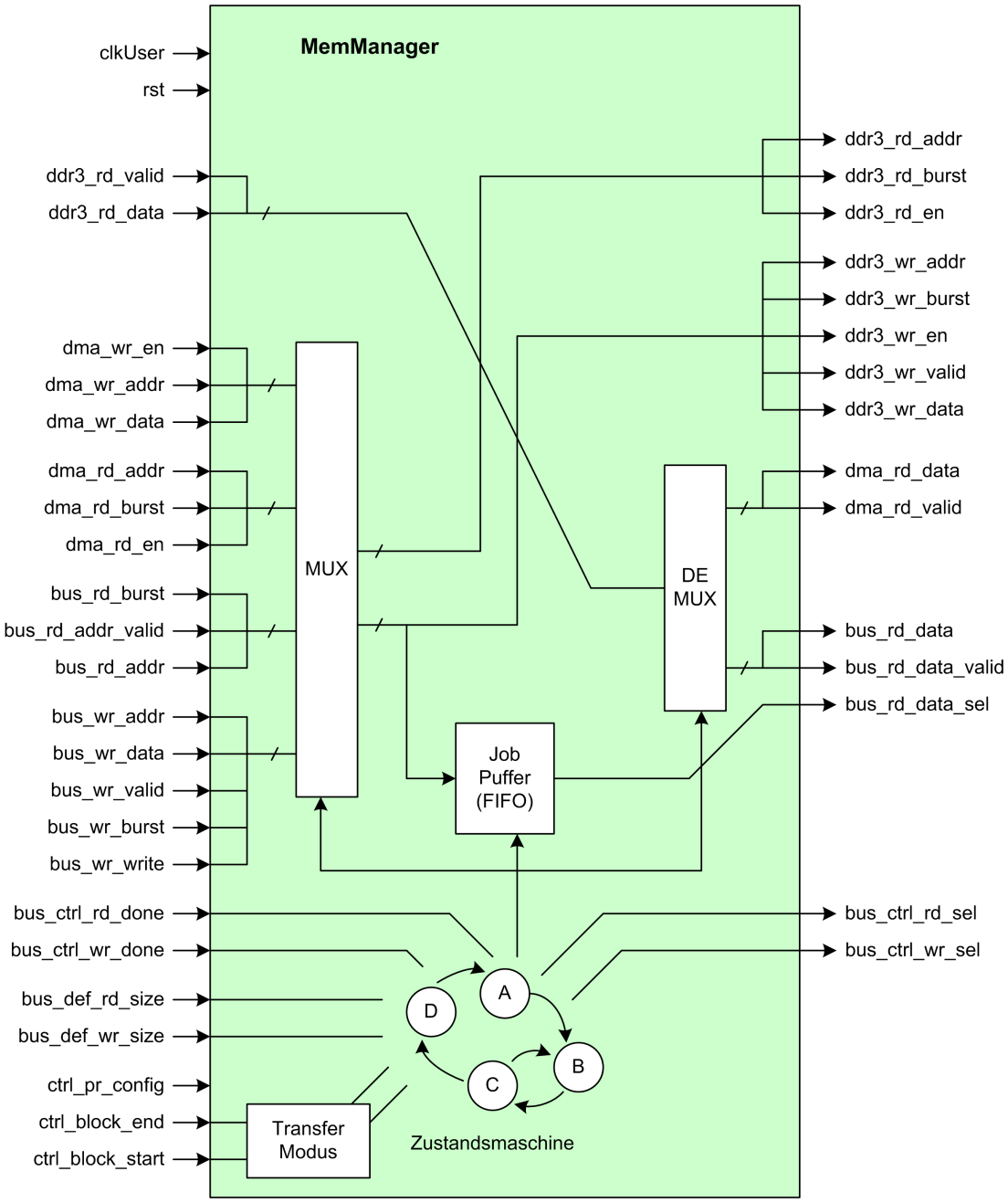


Bild 5.21.: Schematisches Blockschaltbild der Speicherverwaltungseinheit

und einem Gültigkeitssignal, ob es eine Leseanforderung gibt. Kommen nach der Latenzzeit des Speicherkontrollers die Daten, wird aus dem Job-Puffer die Slave-ID zum Selektieren ermittelt. Das Selektionssignal wird synchron mit den Daten und dem Gültigkeitssignal auf den Bus gesendet. Die Daten werden von den Datenpuffern des selektierten Transferblocks aufgenommen.

5.4.7. Kontrolleinheit

Bild 5.22 zeigt den schematischen Aufbau der Kontrolleinheit aus dem Rahmendesign. Die Aufgaben der Einheit sind, die Berechnungen der Pipeline zu steuern, dem Host Zugriff auf die Register zu geben und die aktuellen Parallelindizes der Pipeline mitzuteilen.

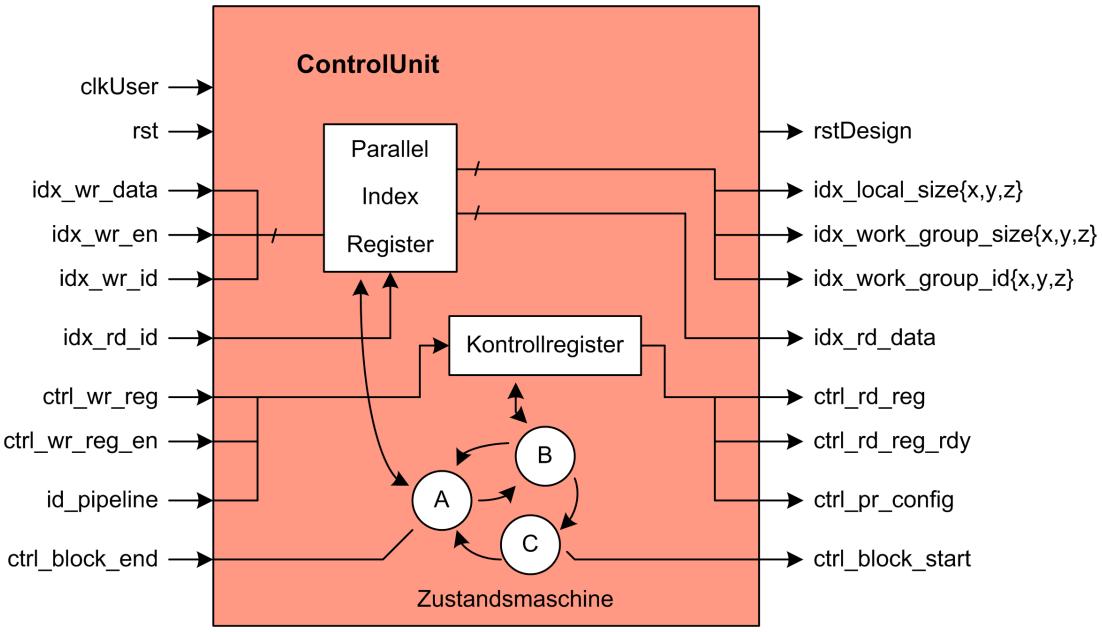


Bild 5.22.: Schematisches Blockschaltbild der Kontrolleinheit

- Die Parallelindizes werden in neun Registern gespeichert, die über eine Lese- und eine Schreibschnittstelle zur PCIe-Einheit verbunden sind. Die Registerinhalte können über die *idx*-Signale von den Adressblöcken innerhalb der Pipeline ausgelesen werden.

- Das 32 Bit breite Kontrollregister ist in Kontrollbits (Bit 0 bis 15) und in Informationsbits (Bit 16 bis 31) aufgeteilt. Nur die Kontrollbits können vom Host beschrieben werden. Die Kontrollbits steuern die Berechnung, den Partieller Rekonfigurations Modus und

den Pipelinereset. Die Informationsbits geben Auskunft über einen FIFO-Überlauf und die Pipeline-ID.

- Eine Zustandsmaschine verwaltet die Berechnung. Sie verharrt in einem Ruhezustand, in dem die Pipeline nicht rechnet, bis das nullte Bit des Kontrollregisters gesetzt wird. Der Berechnungsstart wird der Speicherverwaltungseinheit und der Pipeline mitgeteilt. Im nächsten Zustand wird gewartet, bis die Pipeline das Ende der Berechnung für die aktuelle Kombination der Parallelindizes meldet. Dann wird die Pipeline zurückgesetzt und geprüft, ob alle Parallelindizes-Kombinationen ausgeschöpft wurden. An dieser Stelle verzweigt sich die Zustandsmaschine, entweder startet die Berechnung mit neuen Parallelindizes-Kombinationen, oder alle Berechnungen sind fertig und die Zustandsmaschine geht in ihren Ruhezustand zurück. Das nullte Bit des Kontrollwortes wird zurückgesetzt.

- Die Parallelindizes-Kombination wird zwischen jeder Berechnugnsphase ermittelt. Welche gerade berechnet wird, bestimmen die Register *work group id x* bis *z*. In den Berechnungspausen wird die *work group id* erhöht, wie bei einer dreifach verschachtelten Schleife die ein Volumen elementweise durchläuft. Die *work group size* entspricht der Schleifengröße von jeder Dimension, mit der sich die Anzahl aller Pipelineberechnungen ermitteln lässt. Die Pipeline berechnet alle Threads einer *work-group*. Die Anzahl der Threads lässt sich von den Registern *work item size x* bis z bestimmen. Da jeder Takt einem Ergebnis eines Threads entspricht, benötigt die Kernelfunktion in Form einer Pipeline entsprechend viele Takte plus der Latenzzeit, um alle Threads einer *work group* auszuführen.

5.4.8. Taktnetz

Wie bereits erwähnt, gibt es drei Taktnetze mit unterschiedlichen Frequenzen:

clkDDR3 200 MHz Benutzerschnittstelle Speicher
clkUser 133 MHz Rahmendesign und Pipeline
clkPCIe 125 MHz Benutzerschnittstelle PCIe

Die Taktrate der Pipeline und weite Teile des Rahmendesigns müssen schneller getaktet sein als *clkPCIe*, weil es sonst zum Überlauf der Datenpuffer in der PCIe-Einheit kommen könnte. Ebenso sollte sie wegen eines möglichen Pufferüberlaufs in der Speicherverwaltungseinheit nicht schneller als 200 MHz sein. Die Komponenten der Pipeline sind für eine Frequenz mit 200MHz entwickelt worden. Beliebige Kombinationen der Pipeline-Komponenten, hoher Verdrahtungsaufwand der Stall-Logik, der Datenflusskontrolle und der Speicherbus-Oder-Logik können die Frequenz einschränken. Die Pipelinefrequenz wird auf vorsichtige 133 MHz ge-

setzt. Mit dieser Frequenz konnten eine Vielzahl unterschiedlicher Kernelfunktionen in synthesefähige und lauffähige Pipelines übersetzt werden und garantieren so eine fehlerfreie Übersetzung im Zeitverhalten.

5.5. OpenCL - Laufzeitumgebung

5.5.1. FPGA Kommunikation

5.5.1.1. Entwicklungssteckkarte ML605

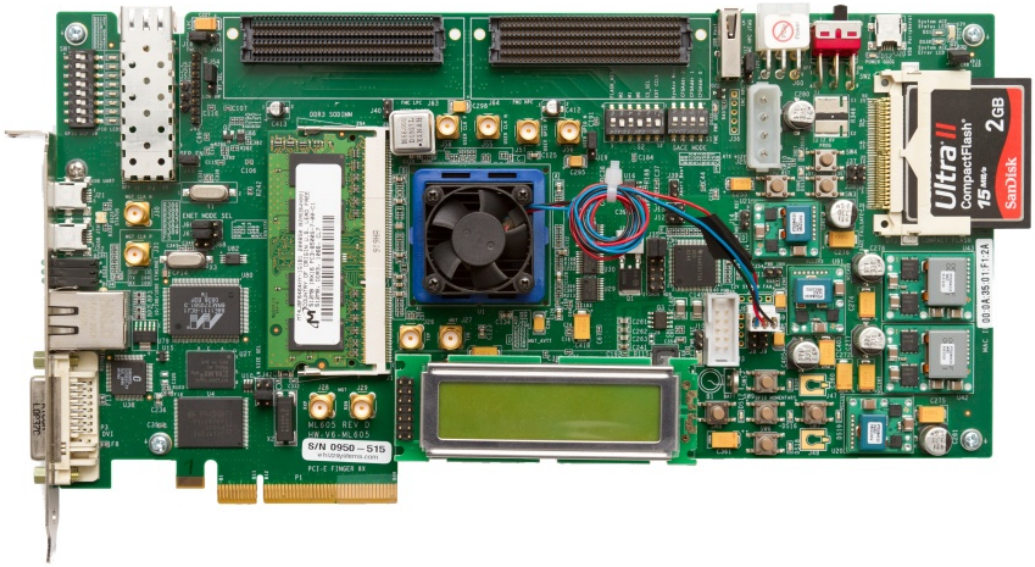


Bild 5.23.: ML605 Entwicklungsplatine als PCIe-Steckkarte verwendet.

Das Bild 5.23 zeigt die Entwicklungssteckkarte von Xilinx [82], die in dieser Arbeit für den Kompilierer verwendet wird. Die Karte ist mit einer Vielzahl von Bausteinen und Schnittstellen ausgestattet, um möglichst viele unterschiedliche Beispieldesigns und Entwicklungsanwendungen demonstrieren zu können. Für die Kompiliererentwicklung unwichtige Merkmale sind: Gigabit-Transceiver, Gigabit-Ethernet-Schnittstelle, USB 2.0 Schnittstelle, ein DVI-Monitoranschluss und ein 16 mal 2 Zeichen LCD-Monitor. Die wichtigen Hauptmerkmale in Bezug auf den Kompilierer, die teilweise bereits erwähnt wurden, sind hier aufgelistet:

DDR3 SO-DIMM Speicher (512 MB). Das 512 MB große Speichermodul wird als globaler Speicher verwendet. Es ist möglich, es durch ein größeres zu ersetzen, womit der Speicherkontroller neu konfiguriert werden muss.

PCI Express x8 Stecker. Die PCIe-Schnittstelle wird für die Kommunikation zwischen Karte und Host-PC verwendet. Die Anbindung der DMA-Engine liegt bei vier Lanes der ersten PCIe Generation (PCIe x4 v1.0).

200 MHz Differentialoszillator. Dieser Oszillator ist die Taktquelle der Taktnetze *clkUser* und *clkDDR3*. Das Taktnetz *clkPCIe* wird von der PCIe-Schnittstelle gespeist.

LEDs und DIP-Schalter. Die Leuchtdioden werden für die Fehleranzeige und der Betriebsanzeige verwendet. Mit den Schaltern kann Testlogik aktiviert werden, um die ordnungsgemäße Funktion einzelner Teile zu ermitteln.

BPI Linear Flash (32 MB). Der Flash-Speicher hält das Rahmendesign, das mit dem Einschalten des PCs und der Versorgung der Karte automatisch geladen wird.

5.5.1.2. PCIe-Treiber und MPRACE-Bibliothek

Die Entwicklungssteckkarte ML605 als Beschleunigerkarte kommuniziert mit dem Host über den PCIe-Bus. Die Kommunikation wird mit Hilfe eines Linux Treibers (PCIe-Treiber) und einer C/C++ Bibliothek (MPRACE) bewältigt, siehe [56]. Der Puffermanager [57] aus der MPRACE-Bibliothek, der Transfers zwischen dem virtuellen und dem physikalischen Adressraum koordiniert, arbeitet mit der bereits beschriebene DMA-Engine zusammen. Die MPRACE-Bibliothek umfasst einfache Funktionen für einen peripheren Transfer (PIO), einen direkten Speichertransfer (DMA) und Registerzugriffe in der DMA-Engine, die die Kommunikation steuert. Die Kommunikation des Kompilierers basiert auf DMA-Zugriffen und Registerzugriffen. Quelltextabschnitt 5.1 zeigt die verwendeten Lese- und Schreibfunktionen.

```
1  virtual void readDMA(
2    const unsigned int address,      // FPGA Adresse
3    DMABuffer& buf,                  // DMA Puffer
4    const unsigned int count,        // Anzahl zu lesender 32Bit Werte
5    const unsigned int offset = 0,   // Versatz des ersten Wertes
6    const bool inc = true,           // Adresse im FPGA inkrementieren
7    const bool lock = true,          // Warte bis der Transfer fertig ist
8    const float timeout = 0.0);      // Abbruchzeit in Millisekunden
9
10 virtual void writeDMA(
```

```
11   const unsigned int address,      // FPGA Adresse
12   const DMABuffer& buf,            // DMA Puffer
13   const unsigned int count,        // Anzahl zu lesender 32Bit Werte
14   const unsigned int offset = 0,   // Versatz des ersten Wertes
15   const bool inc = true,           // Adresse im FPGA inkrementieren
16   const bool lock = true,          // Warte bis der Transfer fertig ist
17   const float timeout = 0.0);      // Abbruchzeit in Millisekunden
18
19 virtual void setReg(
20   const unsigned int address,      // FPGA Adresse
21   const unsigned int value);       // Wert der geschrieben wird
22
23 virtual unsigned int getReg(       // Wert der gelesen wird
24   const unsigned int address);     // FPGA Adresse
```

Quelltext 5.1: Ausschnitt aus der „board.h“ mit den verwendeten Funktionen

5.5.1.3. Kommunikations-Klasse

Die Kommunikations-Klasse hat spezialisierte Methoden, die auf den Adressraum des Rahmendesigns (siehe Abschnitt 5.4.3) zugeschnitten sind. Sie abstrahieren Kommunikations-Transaktionen mit mehreren Aufrufen von Zugriffsfunktionen und verwalten die Klassen-Instanzen aus der MPRACE-Bibliothek. Bild 5.24 zeigt die Kommunikations-Klasse mit den abstrahieren Zugriffsmethoden.

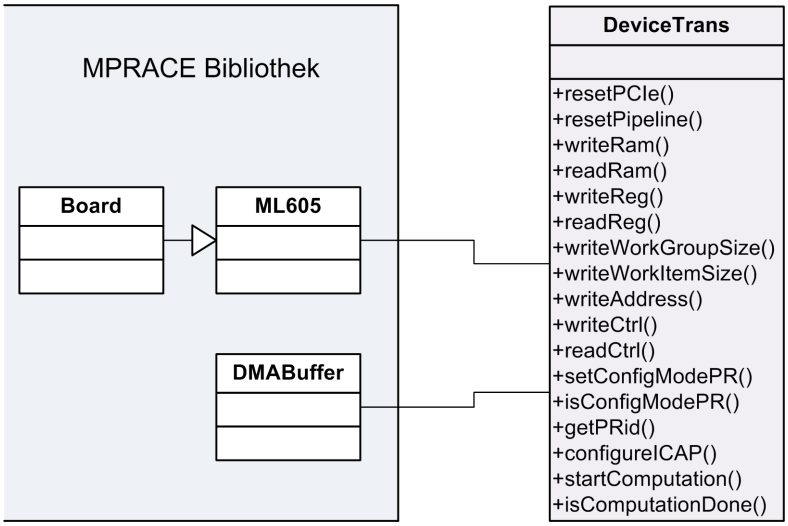


Bild 5.24.: DeviceTrans

Lese-Schreibtransfers auf den Speicher. Es gibt Anforderungen an die Basisadresse und die Zugriffsblockgröße beim Lesen bzw. beim Schreiben auf den globalen Speicher. Die kleinste Zugriffsgröße besteht aus einem zusammenhängenden Block von 64 Bytes, der mit einem

Vielfachen von 64 Bytes und der genauen Bytegrenze von allen 64 Bytes adressiert werden muss. Der DMA-Transfer besitzt die Anforderung einer konfigurierbaren maximalen Datenlänge. Die Methoden `readRam()` und `writeRam()` berücksichtigen alle drei Anforderungen und setzen beliebige Adressen und Datenlängen, wenn nötig, in mehrere DMA-Transfers um.

Reset-Funktionen. Vor jedem DMA-Transfer muss die entsprechende Lese- bzw. Schreib-DMA-Engine rückgesetzt und die Abbruchzeiten neu gesetzt werden. Die Methode `resetPCIe()` übernimmt diese Aufgabe. Sie wird von den Transferfunktionen vor jedem DMA-Transfer aufgerufen. Die Methode `resetPipeline()` setzt ein Bit im Kontrollregister, das für wenige Taktzyklen ein Reset in der Pipeline durchführt.

Lese-Schreibtransfers auf die Register. Beim Lesen und Schreiben der Register in der Kontrolleinheit des Rahmendesigns, implizieren die Methoden `readReg()` und `writeReg()` die Registeradresse im Adressraum, so dass zur Adressierung lediglich die gewünschte Registernummer angegeben werden muss.

Größe der parallelen Indizes setzen. Mit den Methoden `writeWorkGroupSize()` und `writeWorkItemSize()` wird die Dimension der *work-group* und der *work-items* zur Programmlaufzeit in der Kontrolleinheit des Rahmendesigns gespeichert bzw. verändert.

Kontrollregister lesen und schreiben. Auf das 32-Bit Kontrollregister im Rahmendesign kann mit den Methoden `readCtrl()` und `writeCtrl()` zugegriffen werden. (Die einzelnen Bits sind im Abschnitt 5.4.7 erläutert.)

Basisadressen setzen. Mit der Methode `writeAddress()` wird zur Programmlaufzeit die Basisadresse eines Zeigers in ein Register der Adresseinheiten geschrieben. Der Zeiger wird der Kernelfunktion übergeben und anhand der Parameterposition einer Adresseinheit in der Pipeline zugeordnet.

Berechnung steuern. Die Methode `startComputation()` liest und schreibt das Kontrollregister, um das Rechenbit zu setzen. Die Berechnung startet und das Rechenbit bleibt gesetzt solange die Berechnung läuft, was mit der Methode `isComputationDone()` abgefragt werden kann. Der Hostthread pollt auf das Rechenbit, bis es sich selbstständig rücksetzt, dann, wenn die Berechnung fertig ist.

Pipeline Austauschfunktionen. Die Methoden `setConfigModePR()`, `isConfigModePR()`, `configureICAP()` und `getPRid()` sind für den Austausch einer Pipeline verantwortlich. Auf die Funktionen wird später im Abschnitt 5.5.3.2, bei der Erläuterung der Laufzeitumgebung, genauer

eingegangen.

5.5.1.4. Speichertabellen-Klasse

Der globale Speicher wird in der Software verwaltet. Soll heißen, die Software bestimmt beim Anlegen eines Speicherbereichs welche Adresse vergeben wird. Die Singleton-Klasse `DeviceMem`, von der es nur eine Instanz gibt, speichert alle verwendeten Speicherbereiche mit Adresse und Größe in einer Liste. Mit der Methode

```
bool allocate(void** pointer, unsigned int length);
```

wird innerhalb der Liste ein ungenutzter Speicherbereich der entsprechenden Länge gesucht, und im Erfolgsfall wird der Zeigerparameter mit einer Adresse gesetzt. Die Speicherbereiche werden in der Liste nach der Adresse sortiert, um nicht verwendete Speicherbereiche leichter berechnen zu können. Der Algorithmus sieht im Fall eines fragmentierten Speichers vor, die erste passende Lücke zu verwenden. Falls für die angeforderte Speichergröße kein freier Bereich zu finden ist, wird ein Nullzeiger und `false` zurückgegeben. Die Methode

```
bool free(void** pointer);
```

löscht den zugewiesenen Speicherbereich aus der Liste, so dass dieser wieder zur Verfügung steht. Weiter besitzt die Klasse `DeviceMem` eine statische öffentliche Zugriffsmethode zur Instanz, eine Methode um die Liste zu löschen, um den gesamten Speicher freizugeben und für Debugzwecke eine Ausgabemethode, um alle aktuell belegten Speicherbereiche darzustellen.

5.5.2. OpenCL-Funktionen

5.5.2.1. Implementierung der OpenCL-Funktionen

Eine OpenCL-Implementierung sieht vor, alle Funktionen aus dem Standard [36] zu unterstützen. In dieser Arbeit wurde eine Untermenge aller OpenCL-Funktionen in einer Bibliothek implementiert, genug um eine Demonstration zu ermöglichen. Die nicht implementierten Funktionen geben einen Hinweis aus und beenden das Programm, falls sie von einer OpenCL-Anwendung verwendet werden.

Die OpenCL-Funktionen der Laufzeitumgebung sind ANSI C, während die Kommunikationsklasse und die MPRACE-Bibliothek in C++ entwickelt sind. Das bedeutet, dass aus den C-

Funktionen der Laufzeitumgebung Klassenmethoden aufgerufen werden müssen. Abhilfe schaffen Rückruffunktionen (*call back functions*), die in der C-Umgebung deklariert aber in C++ definiert werden. Die Implementierung der *Call-Back*-Funktionen erstellt die nötigen Klassen-Instanzen und setzt die C-Funktionsaufrufe in Klassen-Methodenaufrufe um. Es existiert für jede Methode der Kommunikations-Klasse (`DeviceTrans`) und der Speichertabellen-Klasse (`DeviceMem`) eine entsprechend bezeichnete *Call-Back*-Funktion.

Die Funktion `systemCall()` ist eine selbst entwickelte Variante der Bibliotheksfunktion `system()`, die Linux-Programmaufrufe ausführt. Die Bibliotheksfunktion ließ sich nicht korrekt ausführen, weil deren `fork()` zur Erzeugung eines Kind-Prozesses eine Kopie der DMA-Puffer erstellt, der physikalische Speicheradressen enthält. Sobald der Kind-Prozess beendet wird, werden alle Speicherbereiche inklusive DMA-Puffer freigegeben, auch der physikalische Speicher, der jedoch weiter von der ersten Instanz benötigt wird. In der Funktion `systemCall()` wird ein `vfork()` verwendet, der den Speicherbereich des Vater-Prozesses nicht kopiert.

In den folgenden Abschnitten sind die wichtigsten Passagen der Laufzeitumgebung erläutert, wie sie mit dem Rahmendesign, der Kommunikationsklasse und letztendlich mit der MPRACE Bibliothek zusammenarbeiten. Auf die Fehlerbehandlung wird mit Blick auf das Wesentliche verzichtet. Eine vollständige und lauffähige OpenCL-Anwendung, die den Einsatz des Kompilierers und die Ausführung auf dem FPGA demonstriert, befindet sich im Anhang A.

5.5.2.2. Verwaltung der OpenCL-Geräte

Der Quelltextausschnitt 5.2 aus der Demonstrationsanwendung im Anhang, zeigt wie die OpenCL-Funktionen den FPGA findet und für die weitere Verwendung nutzbar macht.

```
1  cl_int        status   = CL_SUCCESS;
2  cl_uint       number   = 0;
3  cl_device_id  device   = NULL;
4  cl_context    context  = NULL;
5
6  status  = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_ACCELERATOR, 1, &device, &number);
7
8  context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_ACCELERATOR, NULL, NULL, &
       status);
```

Quelltext 5.2: OpenCL Initialisierung des FPGA als Beschleunigerkarte

Die OpenCL-Funktion `clGetDeviceIDs()` sucht im Computer ein OpenCL-Gerät des Types Beschleuniger (`ACCELERATOR`) und gibt eine Liste mit den gefundenen Geräten zurück. Die Implementierung ruft die *Call-Back*-Funktion `hasDevice()` auf, die ihrerseits eine Instanz von `ML605`

anlegt. Erfolgt beim Anlegen keine Exception, existiert die Beschleunigerkarte.

Beim `clCreateContextFromType()`-Aufruf wird eine Kontext-Struktur angelegt, die den Bezug zum OpenCL-Gerät hält und bei der Verwendung der meisten OpenCL-Funktionen mit angegeben werden muss. Der Standard sieht vor, viele OpenCL-Geräte in einen Kontext zusammenzufassen. Die vereinfachte Implementierung dieser Arbeit stützt sich auf einem Kontext mit einem Gerät.

5.5.2.3. Kernelfunktion Übersetzen

In OpenCL wird entweder der Kernelfunktion-Quelltext zur Programmlaufzeit übersetzt oder es wird eine übersetzte Kernelfunktions-Repräsentation aus einer Datei geladen. Der Quelltextabschnitt 5.3 zeigt diesen Vorgang.

```
1  const unsigned char*  fileContent;
2  const size_t          fileSize;
3  bool                  binary;
4
5  if (binary)
6    program = clCreateProgramWithBinary(context, 1, &device, &fileSize, &fileContent,
         NULL, &status);
7  else
8    program = clCreateProgramWithSource(context, 1, (const char**)&fileContent, &
       fileSize, &status);
9
10 status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

Quelltext 5.3: Ausschnitt aus der OpenCL Demonstrationsanwendung des Anhangs, Kompilierung

Die Implementierung der Funktion `clCreateProgramWithBinary()` liest eine bereits übersetzte Kernelfunktion ein, wenn diese beim Programmstart übergeben wurde. Wird dem Programm Quelltext einer Kernefunktion übergeben, wird diese von der Funktion `clCreateProgramWithSource()` eingelesen und einer Programm-Struktur zugeordnet. Die Übersetzung passiert erst mit dem Aufruf von `clBuildProgram()`. Befindet sich in der Programm-Struktur die übersetzte Repräsentation, wird diese in den FPGA geladen. Handelt es sich um Quelltext, wird dieser mit der Ausführung

1. eines Bash-Skripts,
2. des VHDL-Kompilierers und
3. eines TCL-Skripts

in die übersetzte Repräsentation gebracht und dann in den FPGA geladen.

Bash-Skript. Es beinhaltet zwei Programmaufrufe und mehrere Prüfungen. Als Parameter erwartet das Skript eine OpenCL-Datei mit einer Kernelfunktion und der Dateierweitung „cl“. Das Skript prüft die Existenz des Parameters und die Existenz der Datei mit der cl-Endung. Das C-Frontend wird aufgerufen (`clang -c -m32 -emit-llvm "$1"-o "$file".bc`), um in die SSA-Zwischensprache LLVM-IR zu übersetzen. Die Ausgabedatei hat eine bc-Endung. Der Optimierer reduziert die IR-Befehle in ein kompaktes Programm (`opt -O3 -S -o "$file".ll $file.bc`).

VHDL-Kompilierer. Dieser wurde im Abschnitt 5.3 ausführlich beschrieben. Er übersetzt eine ll-Datei in eine „pipeline.vhdl“-Datei (`ssaCompiler build/pipeline.ll build/pipeline.vhd`)

TCL-Skript. Die „pipeline.vhdl“ wird vom einem TCL-Xilinx-Skript (Tool Command Language) in eine bit-Datei, ein Hardwaredesign, synthetisiert (`pipelineTCL.sh build/pipeline.vhd`). Das Skript generiert lediglich die Hardware-Pipeline als dynamischen Teil und wird zum Übersetzen statischen Rahmendesigns verknüpft. Der nächste Schritt ist, die Pipeline auf dem Chip auszutauschen. Weitere Details sind im Abschnitt „Pipeline Module mit DPR austauschen“ 5.5.3.2 erläutert.

5.5.2.4. Datenübertragung und Pipeline starten

Befindet sich die Kernelfunktion als Pipeline im FPGA, kann sie für eine Berechnung herangezogen werden. Der Quelltextausschnitt 5.4 zeigt, welche Strukturen benötigt werden, um die Berechnung auszuführen.

```
1  kernel = clCreateKernel(program, "kernel", &status);
2  queue  = clCreateCommandQueue(context, device, 0, &status);
3
4  mem1= clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned int) * MEGA, NULL, &
       status);
5  mem2= clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned int) * MEGA, NULL, &
       status);
6  h_mem1 = new unsigned int[MEGA];
7  h_mem2 = new unsigned int[MEGA];
8  status = clEnqueueWriteBuffer(queue, mem1, true, 0, sizeof(unsigned int) * MEGA,
       h_mem1, 0, NULL, NULL);
9
10 status = clSetKernelArg(kernel, 0, sizeof(mem1), mem1);
11 status = clSetKernelArg(kernel, 1, sizeof(mem1), mem1);
12 status = clSetKernelArg(kernel, 2, sizeof(mem2), mem2);
13
```

```
14 const size_t global_work_size[3] = { MEGA, 1, 1 };
15 const size_t local_work_size[3]  = { 1, 1, 1 };
16 status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
       local_work_size, 0, NULL, NULL);
17
18 status = clEnqueueReadBuffer(queue, mem2, true, 0, sizeof(unsigned int) * MEGA,
       h_mem2, 0, NULL, NULL);
```

Quelltext 5.4: Ausschnitt aus der OpenCL Demonstrationsanwendung des Anhangs, Kompilierung

In der ersten und in der zweiten Zeile wird in der Programm-Struktur die einzige Kernelfunktion extrahiert und eine Kommando-Warteschlange erstellt.
Zeile vier bis acht legt Speicher im Host an und kopiert die Daten in den externen (globalen) Speicher der ML605-Steckkarte. Die Funktion `clCreateBuffer()` greift über die *Call-Back*-Funktion auf die Methode `DeviceMem.memAllocate()` zu, um einen Speicherbereich zu reservieren. Die Implementierung der Funktion `clEnqueueWriteBuffer()` verwendet die *Call-Back*-Funktion `devWriteRam()`, um über einen DMA-Kanal die Daten in den externen Speicher zu kopieren.
Mit Zeile zehn bis zwölf werden die Zeiger der reservierten Speicherbereiche als Parameter der Kernelfunktion übergeben. In der `clSetKernelArg`-Implementierung wird die Zeigeradresse mit der *Call-Back*-Funktion `devWriteAddress()` in die Adressblock-Komponenten als Basisadresse übertragen. Jetzt weiß die Pipeline, an welcher Adresse die Daten der jeweiligen Kernel-Funktionsparameter liegen und sie können vom Adressblock fortlaufend adressiert werden, um einen Datenfluss zu generieren.
In Zeile 14 und 15 wird die Größe und Dimension der *work-items* definiert, was der Threadanzahl entspricht. Die Größen werden in die Kontrolleinheit in Register kopiert. Die Adressblöcke benötigen die Größen (Parallelindizes), um das Zugriffsmuster und die letzte Adresse im Datenfluss zu bestimmen. Nach der Bekanntgabe der Parallelindizes wird die Berechnung in Zeile 16 gestartet. Die Implementierung setzt die Parallelindizes mit der *Call-Back*-Funktion `devWriteWorkGroupSize()` und `devWriteWorkItemSize()`. Die Berechnung wird mit der *Call-Back*-Funktion `devStartComputation()` gestartet, womit das Bit Null im Kontrollregister gesetzt wird. In einer Schleife wird das Bit Null mit `devComputationDone()` kontinuierlich ausgelesen und geprüft, ob die Berechnung fertig ist. Um die CPU-Last des aktiven Wartens zu reduzieren, wird innerhalb der Schleife der Hostthread für wenige Mikrosekunden nach jeder Prüfung schlafen gelegt.
Die letzte Zeile zeigt, wie die Ergebnisse der Berechnung vom externen Speicher des ML605 in den Arbeitsspeicher des Hosts kopiert werden.

5.5.3. Austausch Pipeline Modul

5.5.3.1. Programmierschnittstellen und DPR

Der PCIe-Bus hat die Anforderung, beim Bootvorgang des PCs zu allen PCIe-Geräte einen Link herstellen zu müssen, und zwar innerhalb einer Zeitspanne von hundertstel Millisekunden [81]. Die JTAG-Schnittstelle ist für diese Anforderung unpraktisch. Es besteht die Möglichkeit, den FPGA mit einem Design inklusive PCIe-Core zu beschreiben und anschließend den PC neu zu starten, um den Link zwischen Bus und PCIe-Core herzustellen. Eine andere Möglichkeit besteht in der Verwendung eines Bootflashs, das einmal mit einem Design beschrieben wird und beim Bootvorgang das Design in den FPGA lädt. Auch dann wird der PCIe-Core innerhalb der Zeitspanne richtig initialisiert.
Ein weiteres Problem bezüglich der Programmierung besteht, wenn eine Kernelfunktion in ein Design übersetzt wurde und das FPGA damit programmiert werden soll. Während des Programmierens wird der PCIe-Core vom Bus getrennt und die Initialisierung geht verloren. Wenn der PC dabei nicht abstürzt, ist es zumindest nicht mehr möglich, den neu geladenen PCIe-Core zu verwenden. Die einzige Option, den FPGA mit einem neuen Design zu laden, ist ein Neustart des PCs.
Die Lösung des Problems besteht darin, den initialisierten PCIe-Core bei einer Programmierung nicht zu überschreiben und zu halten, damit der Link bestehen bleibt. Im Allgemeinen ist die Neuprogrammierung lediglich für den FPGA-Bereich notwendig, in dem die Pipeline platziert wird. Der PCIe-Core, die Logik des Speicherkontrollers und des Rahmendesigns bleiben für jede übersetze Pipeline konstant und müssen somit nicht neu programmiert werden. Mit DPR kann dieses Verhalten erzielt werden, indem die Pipeline in einem dynamischen Modul platziert wird und das restliche Design statisch übersetzt wird. Im FPGA wird ein Bereich gewählt, in dem das dynamische Pipelinemodul platziert wird. Außerhalb des Bereichs findet das statische Design die nötigen Ressourcen. Bei der Wahl des Bereichs ist es das Ziel, dem dynamischen Modul so viel FPGA-Chip-Fläche wie möglich zu bieten.
Die Verwendung von DPR bietet an, die ICAP-Schnittstelle (Internal Configuration Access Port) zu verwenden. Mit ihr kann direkt über den PCIe-Bus ein dynamisches Modul in den FPGA geladen werden. ICAP ist mit einer Programmierzeit von Millisekunden schneller als JTAG, variabler als der Bootflash und benötigt kein (USB-)Programmierkabel.

5.5.3.2. Pipeline Module mit DPR austauschen

Zwischen der dynamischen Pipeline und dem statischen Rahmendesign müssen Registerstufen existieren, damit die Schnittstellensignale einen kurzen Pfad aufweisen. Über Modulgrenzen hinweg kann das Zeitverhalten nicht optimiert werden. Die betroffenen Signale liegen zwi-

schen der Pipeline und der Kontrolleinheit im Speicherbus. Im Abschnitt 5.3.6.3 wurde bereits darauf hingewiesen, welche „zusätzliche Logik“ zur Pipeline benötigt wird. Gemeint sind die Registerstufen im Speicherbus.

Bevor ein dynamisches Modul im FPGA ausgetauscht wird, muss sichergestellt werden, dass keine Logik im statischen Design in einen undefinierten Zustand kommen kann. Während des Austauschs sind die Ausgabesignale des dynamischen Moduls für kurze Zeit undefiniert und können ein Fehlverhalten im statischen Design auslösen. Die Ausgabesignale müssen während des Programmierens von der statischen Logik entkoppelt werden. Der einfachste Weg der Entkoppelung ist, alle Signale mit der Quelle aus dem dynamischen Modul im statischen Design mit einem UND-Gatter zu verknüpfen. Mit einem Steuersignal kann der Ausgang des UND-Gatters und somit das Signal aus dem dynamischen Design gezielt auf logisch Null gesetzt werden. Das UND-Gatter bezeichnen wir als Torschaltung, das von einem Kontrollsignal gesteuert wird und das Tor öffnen und schließen kann.

Wie die OpenCL-Funktion `clBuildProgram()` die Kernelfunktion übersetzt, wurde bereits beschrieben. Es fehlt, wie das generierte (dynamsiche) Pipelinemodul in den FPGA geladen wird:

1. Die Bit-Datei mit dem Pipelinemodul wird vom Dateisystem gelesen, die Dateilänge bestimmt und in eine OpenCL Programm-Struktur gespeichert.

2. Mit der *Call-Back*-Funktion `devSetConfigModePR(1)` wird ein Bit im Kontrollregister gesetzt und die Torschaltung schließt. Die Signale aus dem dynamischen Modul sind im statischen Design logisch entkoppelt.

3. Das Pipelinemodul wird mit `devConfigurePR(program->bitfile, program->bitlength)` über die ICAP-Schnittstelle in den FPGA geladen.

4. Danach wird mit `devSetConfigModePR(0)` die Torschaltung geöffnet und die Signale aus der Pipeline haben wieder Einfluss auf die statische Logik.

5. Ein Pipelinereset mit der *Call-Back*-Funktion `devResetPipeline()` beendet den Austausch der Pipeline.

6. Optional kann die acht Bit große Pipeline-Modul-ID mit `devGetPRid()` aus dem Kontrollregister ausgelesen werden, um den Austausch zu verifizieren.

6. Ergebnisse und Diskussion

Nachdem die Implementierungen der GPU-Beschleunigung (offline Prozessierung) im vierten Kapitel dargelegt und die Implementierung des OpenCL-Kompilierers (online Prozessierung) im fünften Kapitel erläutert wurde, werden in diesem Kapitel die Ergebnisse zu den Implementierungen zusammengetragen.

6.1. GPU-Beschleunigung

6.1.1. Geschwindigkeitsgewinn

Folgende Tabelle 6.1 zeigt die Ausführungszeiten des Haralick Texturen Algorithmus, die alle das gleiche Multizellbild für ihre Berechnung verwendet haben. Verglichen wird die Originalsoftware mit einer optimierten Software Version, einer ersten GPU Version I und einer optimierten GPU Version II. Mit enthalten sind die Ausführungszeiten dreier unterschiedlicher Grafikkarten, die im Zeitraum dieser Arbeit erschienen sind.

	Ausführungs-Zeit [s]	Faktor zu 1.	Faktor zu 2.	Faktor zu 3.
1. Original Software Version	2378	-	-	-
2. Optimierte Software Version	214	11x	-	-
3. GPU Version I (8800 GTX)	11.1	214x	19x	-
4. GPU Version I (GTX 280)	6,6	360x	32x	1,7x
5. GPU Version II (GTX 280)	4,65	511x	46x	2,4x
6. GPU Version II (GTX 480)	2,55	933x	84x	4,3x

Tabelle 6.1.: Ausführungszeiten und Beschleunigungsfaktoren im Vergleich aller vorgestellten Implementierungen.

Der Test-PC, auf dem die Messung durchgeführt wurde, besitzt ein Intel Core 2 Quad CPU (Q6600) mit 2.4 GHz Taktrate, 4 GBytes DDR2 Speicher mit 1066MHz Speichertakt. Die Leistung der Grafikkarten werden im nächsten Abschnitt vorgestellt.

6.1.2. Skalierung des Algorithmusses

Die theoretischen Spitzenleistungen der drei Grafikkarten und der CPU sowie die erreichten Effizienzen der Haralick Bildmerkmalsimplementierung sind in Tabelle 6.2 aufgelistet.

	Rechenleistung [GFLOPS]	Speicherdurchsatz [GBytes/s]	Effizienz Rechnen-leistung	Effizienz Speicher-durchsatz
1. CPU (Q6600)				
theor. Maximal	9,6	17		
Implementierung	0.18	-	1,9%	-
2. GPU (8800GTX)				
theor. Maximal	345	86		
Implementierung	3,36	10,6	0,9%	12,4%
3. GPU (GTX280)				
theor. Maximal	622	141		
Implementierung	8	20,8	1,3%	14,8%
4. GPU (GTX480)				
theor. Maximal	1345	177		
Implementierung	15	37,7	1,1%	21,3%

Tabelle 6.2.: Leistungsvergleich der Architektur mit der Implementierung

Die Spalte Rechenleistung zeigt die theoretisch maximale Rechenleistung der jeweiligen Architektur und die gemessenen Werte der Implementierung. Ebenso wurde der maximale und der gemessene Speicherdurchsatz angegeben. Die Messungen sind Durchschnittswerte für den gesamten Algorithmus, inklusive des Datentransfers zwischen GPU und CPU. D.h. es gibt einzelne Kernelfunktionen, die sehr effizient rechnen bzw. effiziente Speichertransfers haben, und andere, die nur einen Bruchteil der maximalen Leistung verwenden. In den Spalten Effizienz wurden die erzielte Leistungen im Verhältnis zu theoretisch maximalen Leistungen dargestellt. Es ist zu erkennen, dass die Rechengeschwindigkeit von den Speichertransfers gebremst wird, da sie eine weitaus höhere Ausnutzung hat als die Rechenleistung. Weiter lässt sich erkennen, dass die Implementierung mit besseren Architekturen skaliert. Die Effizienz der Rechenleistung bleibt im Schnitt bei einem Prozent. Der Anstieg auf 1,3% der Recheneffizienz von der GTX280 liegt an der verbesserten Implementierung, siehe nächster Abschnitt.

6.1.3. Optimierungsergebnisse der zweiten Version

Die Verbesserungen, die im Abschnitt „Profiler“ 4.3.3 auf der Seite 63 beschrieben wurden, ergaben ein beschleunigtes Verhalten. Die Optimierungskonzepte konnten auf die meisten Kernelfunktionen angewendet werden. In der Tabelle 6.3 sind die Ausführungszeiten der zweiten GPU-Versionen aufgelistet sowie deren Beschleunigungsfaktor errechnet. Im Fokus der Optimierung standen hauptsächlich diejenigen Kernelfunktionen mit den längsten Ausführungszeiten. Die Funktionen 2A, 2B, 2C, 3A, 3B, 3C und 3D haben jeweils eine Ausführungszeit von $4ms - 6ms$. Die Funktionen A4, 4B und 4C haben Ausführungszeiten um die $10ms$. Diese Funktionen sind bereits sehr effizient, wurden nicht optimiert und in die Tabelle nicht aufgenommen.

Funktionsname	GPU Version I	GPU Version II	Faktor
Funktion 0A	276,3 ms	275,9 ms	1
Funktion 0B	242,4 ms	86,8 ms	2,8
Funktion 0C	466,4 ms	367,6 ms	1,3
Funktion 0D	224,4 ms	141,0 ms	1,6
Funktion 1A	221,8 ms	221,2 ms	1
Funktion 1B	416,8 ms	373,6 ms	1,1
Funktion 1C	202,5 ms	203,3 ms	1
Funktion 1D	929,2 ms	177,0 ms	5,2
Funktion 1E	310,1 ms	288,2 ms	1,1
Funktion 1F	602,6 ms	418,2 ms	1,4
Funktion 5A	418,6 ms	300,5 ms	1,4
Funktion 5B	269,3 ms	270,7 ms	1
Funktion 5C	309,9 ms	273,0 ms	1,1
Funktion 5D	225,1 ms	226,5 ms	1
Gesamtausführungszeit inklusive nicht gelisteter Funktionen	6600 ms	4650 ms	1,42

Tabelle 6.3.: Ausführungszeiten und Beschleunigungsfaktoren der GPU Version I und II mit ausschließlich relevanten Funktionen

6.2. OpenCL-Kompilier

6.2.1. Nutzen der OpenCL-Implementierung

Mit dieser OpenCL-Implementierung ist es wesentlich einfacher, den FPGA mit einer Funktion zu programmieren. Beispiel: Handelt es sich um eine Vektor-Addition-Funktion, kann

diese im FPGA mit einem Addierer und einem Register verwirklicht werden. Das ist für einen hardwarekundigen Programmierer eine leichte Aufgabe und Programmierer ohne Hardwarekenntnisse können das lernen. Was fehlt, ist die Logik, die einen Datenfluss aufrecht erhält und weitere Kontrolllogik, die die Berechnungen starten lässt und das Berechnungsende signalisiert. Die Kommunikation zwischen Host, FPGA-Beschleunigerkarte und dem externen Speicher ist sehr komplex im Vergleich zur eben gezeigten Vektor-Additions-Funktion. Für die Aufgaben, einen Datenfluss herzustellen, benötigen Programmierer sehr viel Zeit und umfangreiche Hardwarekenntnisse. Die OpenCL-Implementierung bietet eine Umgebung (OpenCL-Laufzeitumgebung), in der die Kommunikation mit der FPGA-Beschleunigerkarte und die Generierung eines Datenflusses bereits implementiert ist. Der nötige Progammquelltext, der die Laufzeitumgebung verwendet und die Kommunikation mit der Beschleunigerkarte regelt, ist für alle OpenCL-Implementierungen sehr ähnlich, weit verbreitet und leicht zu erlernen. Weiter bietet die OpenCL-Implementierung einen Übersetzer an (OpenCL zu FPGA Kompilierer), eine C-ähnliche parallele Funktion in eine Hardwarebeschreibungssprache zu kompilieren. Mit der Laufzeitumgebung und dem Übersetzer ermöglicht diese OpenCL-Implementierung Programmierern ohne Hardwarekenntnisse, eine FPGA-Beschleunigerkarte einzusetzen. Übersetzte Funktionen können zur Laufzeit im FPGA geladen und zur Ausführung gebracht werden. Die Übersetzungszeit einer OpenCL-Kernelfunktion in ein Pipelinemodul mit weniger als zehn Operatoren beträgt ca. 40 Minuten. Die Übersetzungszeit mit der Datenfluss- und Kontrolllogik (PCIe-Core, DMA-Engine, Speicherkontroller und Rahmendesign) benötigt mehr Zeit. Bereits übersetzte Pipelinemodule können im Bruchteil einer Sekunde in den FPGA geladen werden.

6.2.2. Nutzen des Pipelinekonzepts

Da OpenCL eine parallele Sprache ist, ist es einfach eine Kernelfunktion auf den FPGA parallel abzubilden. Viele parallel arbeitende Threads durchlaufen die gleichen Operationen eines Programms. Es liegt nahe, die Operationen eines Threads in eine Pipelinestruktur zu ordnen und sie von vielen Threads durchlaufen zu lassen, wie es in dieser Arbeit gemacht wurde. Jeder Takt liefert ein Ergebnis für einen Thread. Genau darin besteht der Vorteil des Pipelinekonzepts, alle Operatoren in der Pipeline parallel arbeiten zu lassen. Die Pipeline ist mit 133MHz getaktet. Gibt es in der Pipeline N Operatoren, dann ist der Rechendurchsatz R

$$R\,[GigaOperationen/s] = 0,133\,[GHz] * N\,[Operationen]. \tag{6.1}$$

Mit steigendem N und höherer Taktrate wird die Pipeline effizienter. Folgende Faktoren lassen den Rechendurchsatz sinken:

- Die Kernelfunktion besitzt nur wenig Operationen und die Pipeline arbeitet wenig paral-

lel.

- Die Pipeline benötigt zu viele FPGA-Ressourcen, so dass das Zeitverhalten nicht erfüllt werden kann und die Taktrate gesenkt werden muss.
- Bei Pipelines mit vielen Speicherzugriffen könnte die Speicherbandbreite nicht ausreichen und die Pipeline muss zeitweise angehalten werden, bis alle Teile (Transferblöcke) wieder mit Daten versorgt sind.

Die in der Einführung vorgestellten theoretisch maximalen FPGA-Rechenleistungen von ca. 100 GFLOPs, können aus diesen Gründen nicht erreicht werden. Unter optimistischen Voraussetzungen wird in der Praxis eine Rechenleistung von ca. 10 GFLOPs, 10% von der theoretischen Rechenleistung, erreicht. Der Unterschied liegt in der geringenen Taktrate und der fehlenden Möglichkeit, alle Ressourcen für die Rechnungen einzusetzten.

6.2.3. Bandbreite Speicherkontroller

Die Speicherbandbreite des DDR3-1066 Speichermoduls hat eine maximale Datenrate von $8,5\ GBytes/s$. In diesem Design wird die Benutzerschnittstelle über die vereinfachte Schnittstelle mit 133MHz statt 200MHz betrieben. In jedem Takt können 32 Bytes übertragen werden. Die errechnete Speicherbandbreite liegt bei rund $4\ GBytes/s$.
Damit die Pipeline nicht unterversorgt wird, muss jede Transferleseeinheit mit Daten vom Speicher versorgt sein. Jede Transferleseeinheit T_{rd} liefert taktweise 4 Bytes an die Pipeline. Die maximale Anzahl von Transferleseeinheiten bestimmt sich zu

$$T_{rd} = \frac{32\ Bytes\ pro\ Takt\ vom\ Speicher}{4\ Bytes\ pro\ Transferleseeinheit\ und\ Takt}. \tag{6.2}$$

Damit die Pipeline kontinuierlich ein Ergebnis pro Takt zu liefern kann, dürfen maximal acht Transferleseeinheiten in der Pipeline verbaut sein. Es dürfen auch mehr sein, aber dann arbeitet die Pipeline nicht mehr optimal und die Berechnung setzt taktweise aus, um auf Daten zu warten. Die gleiche Rechnung gilt ebenso für die Transferschreibeinheiten, um einen Datenstau zu vermeiden.

6.2.4. Ressourcenbedarf des Designs

Der Ressourcenbedarf wird für das statische und das dynamischen Design getrennt behandelt. Das statische Design umfasst die Logik des Speicherkontrollers, der vereinfachten Speicherschnittstelle, der PCIe-Logik, der DMA-Engine und dem Rahmendesign. Das dynamische Design entspricht der Pipeline mit der zusätzlichen Logik. Da die Pipeline selbst aus Blöcken

besteht, die unterschiedlich zusammengesetzt werden können, werden sie in der Ressourcentabelle 6.4 einzeln aufgelistet.

	Slice Registers		Slice LUTs		Block RAM/FIFO		Slice DSP48E1s	
statisches Design	17083	5,7%	17245	11,4%	34	8,2%	-	-
BlockAddressing	374	0,1%	925	0,6%	-	-	4	0,5%
BlockComputing (ADD)	99	0,0%	35	0,0%	-	-	-	-
BlockComputing (MUL)	101	0,0%	3	0,0%	-	-	4	0,5%
BlockCondAssign	132	0,0%	43	0,0%	-	-	-	-
BlockTransferRd	2350	0,8%	1427	0,9%	6	1,4%	-	-
BlockTransferWr	986	0,3%	1562	1,0%	4	1,0%	-	-

Tabelle 6.4.: Ressourcenverbrauch des statischen Designs und der Pipelinebausteine.

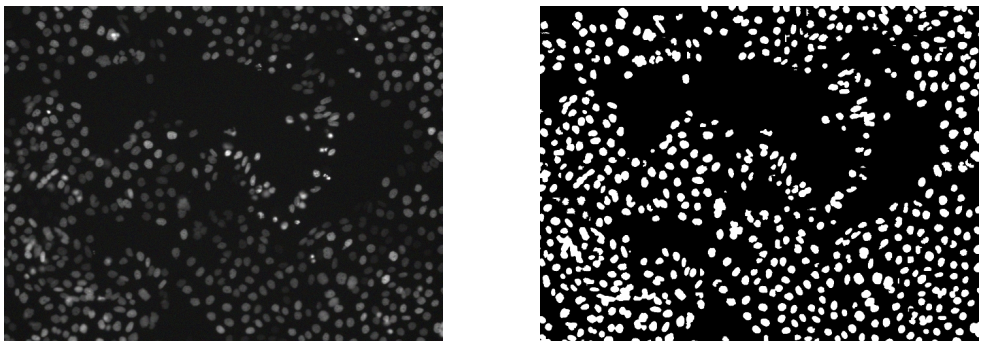
Anhand der Ressourcentabelle lässt sich abschätzen, wie viele Pipelinebausteine zu einer Pipeline zusammengefügt werden können. Für eine hohe Ausnutzung der FPGA-Ressourcen muss entsprechend ein großer Bereich für das dynamische Design reserviert werden. Unter Ausnutzung des gesamten FPGAs, einschließlich des statischen Designs, könnten ca. 130 Pipelinebausteine hinein passen. Die Schätzung berücksichtigt ausschließlich Integer-Operatoren und eine durchschnittliche Nutzung aller existierender Pipelinebausteine. Sofern eine Fließkommabibliothek in den Kompilierer einbezogen wird, wäre die Bausteindichte mit Fließkommaoperatoren wesentlich kleiner.

6.2.5. Beispiel-Applikationen

Ein Multizellbild, das vom Haralick Bildmerkmalsalgorithmus offline auf der GPU prozessiert wird, muss einen planen Hintergrund, der Bereich zwischen den Zellen, mit Nullwerten haben. Eine Variante, den Hintergrund zu entfernen, ist, das Multizellbild 6.1 (a) mit einem segmentierten Binärbild 6.1 (b) zu multiplizieren. Das Binärbild hat den Hintergrundwert Null und innerhalb der Zellen den Wert Eins.

Die Aufgabe, den Hintergrund zu entfernen, lässt sich auf dem FPGA online prozessieren. Folgendes Quelltextbeispiel 6.1 zeigt die Kernelfunktion, die dazu imstande ist und sich für den FPGA übersetzen und ausführen lässt.

```
1 __kernel void removeBGwithSementImg(int* multicell, int* segment, int* Z, int w)
2 {
3   int idx   = get_global_id(0);
```



(a) (b)

Bild 6.1.: Multizellbilder, (a) verrauscht im Hintergrund und (b) binär segmentiert

```
4   int idy   = get_global_id(1);
5   int index = idy * w + idx;
6
7   Z[index]  = multicell[index] * segment[index];
8 }
```

Quelltext 6.1: Kernelfunktion, die ein Multizellbild mit einem segmentierten Binärbild multipliziert

Existiert zur Zeit der Aufnahme kein binäres Segmentbild aus einer anderen Quelle der Online-Prozessierung, kann ein Algorithmus der den Hintergrund entfernt, für den FPGA übersetzt werden. Der vorgestellte Algorithmus 6.2 verwendet einen Schwellwert, der über dem Rauschen liegt, um den Hintergrund auf Null zu setzen.

```
1 __kernel void removeBGwithSementImg(int* multicell, int* Z, int w)
2 {
3   int idx   = get_global_id(0);
4   int idy   = get_global_id(1);
5   int index = idy * w + idx;
6
7   Z[index] = (multicell[index] > 10) ? multicell[index] : 0;
8 }
```

Quelltext 6.2: Kernelfunktion, die ein Multizellbild einem Schwellwert unterzieht

Die letzte Methode, die alle Pixel unterhalb eines Schwellwertes auf Null setzt, manipuliert sowohl die Pixel, die sich im Hintergrund befinden, als auch diejenigen, die sich innerhalb der Zellen befinden. Werden zu viele Pixel innerhalb der Zellen verändert, bleibt die Möglichkeit, einen komplexeren Segementierungsalgorithmus in OpenCL zu implementieren. Kann die komplexe Kernelfunktion nicht mit dem FPGA-Kompilierer übersetzt werden, dann kann sie für GPUs übersetzt werden, ohne dass diese verändert werden müssen.

7. Fazit und Ausblick

7.1. Ziele der Arbeit

Diese Arbeit bietet Lösungen zur Online- und Offline-Prozessierung aus dem ViroQuant-Projekt an.

Bei der Offline-Prozessierung können GPUs eingesetzt werden, um einen Rechencluster zu beschleunigen. Die Verarbeitung der Mikroskopbilder lässt sich im allgemeinen sehr gut auf den Grafikkarten, sowohl in der Pixel-Ebene, als auch in der Bild-Ebene, parallelisieren bzw. beschleunigen. Eine Anwendung für die beschleunigte Offline-Prozessierung ist der Haralick Bildmerkmalsalgorithmus, der den Biologen gute Ergebnisse liefert, dafür aber einen hohen Rechenaufwand besitzt. Die GPU-basierte Beschleunigung dieser Arbeit reduzierte die Rechenzeiten von Monaten auf Stunden. Aufgrund des Skalierungsverhaltens der letzten GPU-Generationswechsel versprechen zukünftige Grafikkarten mit einem einfachen Austausch eine weitere Beschleunigung, ohne den Algorithmus anpassen zu müssen. Die zu erwartende Beschleunigung mit der bereits erzielten ergeben keinen Bedarf, den Algorithmus weiter verbessern zu müssen. Eine weitere Beschleunigung würde für die Gesamtausführungszeit der Algorithmenkette aus der automatischen Bildanalyse keinen nennenswerten Vorteil bieten.

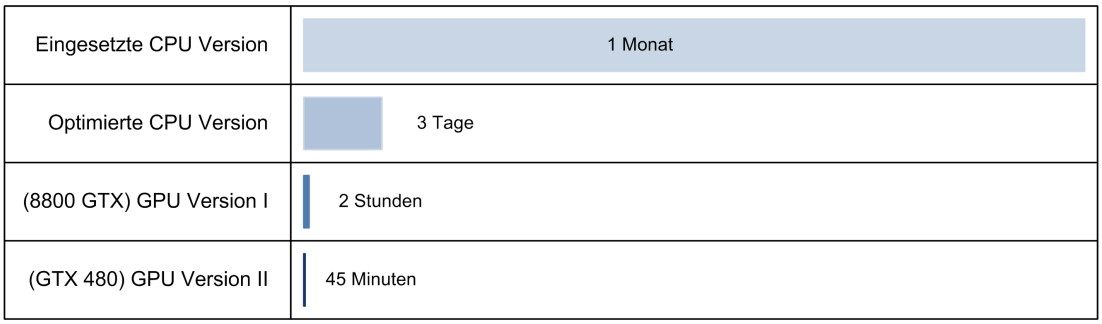


Bild 7.1.: Vergleich der Ausführugszeiten der unterschiedlichen Versionen

Bild 7.1 illustriert die Ausführungszeiten des Haralick Algorithmuses mit unterschiedlichen Implementierungen. Als Referenz dient die Originalversion, die eingesetzt wurde, um die Bildmerkmale zu berechnen. Sie benötigt für einen Datensatz, einer bestimmten Größe einen Monat

Rechenzeit. Die Illustration zeigt, dass die optimierte Version mit drei Tagen Rechenzeit bereits einen wichtigen Teil der Beschleunigung erreicht. Mit dieser Beschleunigung ist es den Biologen möglich, mehrere Datensätze in einer Woche zu analysieren. Der Einsatz der Grafikkarte reduziert die Wartezeit von Tagen auf zwei Stunden bzw. 45 Minuten. Diese Beschleunigung ist maßgebend dafür, dass mehrere Tests am selben Tag durchgeführt werden können. In der jungen Disziplin der Systembiologie mit ihren vielen unerforschten Bereichen ist es außerordentlich wichtig, schnell Ideen und Ansätze auf deren Ergebnisse prüfen zu können, um die besten Ansätze aufzuspüren. Der Einsatz der GPU trägt maßgebend zu einem schnelleren Erkenntnissgewinn bei.

Die Online-Prozessierung hat die Anforderung, gestreamte Daten in Echtzeit verarbeiten zu können, an neue Gegebenheiten anpassbar zu machen und leicht programmierbar zu sein. OpenCL kombiniert mit FPGAs auf einer Beschleunigerkarte erfüllen alle Anforderungen. OpenCL ist eine anerkannte, weit verbreitete und leicht zu erlernende Sprache. FPGAs werden häufig für Echtzeitanwendungen eingesetzt und lassen sich durch Neuprogrammieren mit anderen Anwendungen ersetzen.

Die in dieser Arbeit demonstrierte OpenCL-Entwicklung erfüllt den Zweck der Online-Prozessierung aus den Anforderungen. Einfache Bildverarbeitungsalgorithmen können in Kernelfunktionen umgesetzt werden. Diese wiederum können für den FPGA in Pipelinemodule übersetzt werden. Dynamisch zur Laufzeit können die Pipelinemodule im FPGA ausgetauscht und ausgeführt werden. Pipeline und Rahmendesign zusammen lassen den Bildverarbeitungsalgorithmus auf dem FPGA ausführen. Das komplexe Zusammenspiel aller Einzelkomponenten liefert gute Ergebnisse. Diese Arbeit ermöglicht den Biologen in Bezug auf die Online-Prozessierung,

- die bekannte und weit verbreitete parallele Sprache OpenCL zur Programmierung der FPGA-basierten Beschleunigerkarte einzusetzen,
- ohne Hardwarekenntnisse Kernelfunktionen entwickeln zu können,
- diese in eine gestreamte Pipeline für den FPGA zu übersetzen,
- und die FPGA-Beschleunigerkarte mit einfachen Funktionen aus der standardisierten OpenCL-Laufzeitumgebung ansteuern zu können.

Weder FCUDA, noch OpenRCL bzw. keine bisherige Entwicklung vereint alle aufgelisteten Vorteile.

Eine zu dieser Arbeit parallele, kürzlich erschienene, OpenCL-Entwicklung für Altera-FPGAs [16] zeigt, dass es einen reellen Bedarf gibt, FPGAs mit OpenCL programmieren zu kön-

nen. FPGAs sind sparsamer im Energiebedarf und mit der Pipelinetechnik rechnen sie eine bis zwei Größenordnungen schneller als Einkern-CPUs. Der Trend zeigt, dass die Taktfrequenz der CPUs kaum noch ansteigt, hingegen steigt die Anzahl der Kerne, um leistungsfähigere CPUs herzustellen. Die höhere Integrationsdichte der zukünftigen Chips steigert bei FPGAs die Anzahl und die Komplexität der Logikzellen, um einen höheren Grad der Parallelisierung zu erreichen. Die Frequenzen der FPGAs steigen nur zweitrangig an. OpenCL ist heute und in der Zukunft eine ideale Sprache, Algorithmen wahlweise parallel auf Vielkern-CPUs, GPUs oder FPGAs ausführen zu lassen.

Die Ergebnisse dieser Arbeit konnten teilweise in Journals, einem Konferenzbeitrag und in einem Poster veröffentlicht werden. Die Veröffentlichung zur Beschleunigung der Haralick Bildmerkmale auf der GPU wurde auf der ICPDC'08 zum „*Best Paper*“ normiert [32]. Auf eine Einladung hin wurde der Konferenzbeitrag zu einem Journal erweitert [33] und im *IAENG International Journal of Computer Science* veröffentlicht. Die Ergebnisse der zweiten GPU-Version konnten im *Proceedings of the Fourth International Conference on High Performance Scientific Computing* vorgestellt werden [34]. Zuletzt wurden technische Details zur GPU-Beschleunigung auf der NVIDIA Konferenz GTC'09 mit einem Poster dem Fachpublikum präsentiert [31].

7.2. Verbesserungen für die Zukunft

Wie bereits erwähnt, lohnt sich eine weitere Beschleunigung der Haralick Bildmerkmalserkennung nicht und es besteht auch kein Bedarf einer Weiterentwicklung.

Der OpenCL FPGA Kompilierer besitzt den Stand eines Prototyps, der eine gezielte Aufgabe erfüllt. Für eine allgemeinerer Verwendung wurde explizit auf eine modulare Entwicklung geachtet, um Funktionalität hinzufügen zu können. Folgende Vorschläge würden den OpenCL FPGA Kompilierer verbessern:

- Die Verwendung eines OpenCL-Frontends würde die Nutzung des *Shared*-Speichers ermöglichen, wofür es ein Konzept im FPGA-Design zu entwickeln gilt.

- Das Hinzufügen weiterer Sprachelemente wie `if-else`-Strukturen und Schleifen würde die Übersetzung komplexerer Programme ermöglichen.

- Die Erweiterung der Speicherzugriffsanalyse und der entsprechenden Adresskomponente löst die Einschränkung bestimmter Speicherzugriffsmuster.

- Zur Unterstützung von `float`-Datentypen muss eine Fließkommabibliothek dem Kompilierer und dem VHDL-Block-Vorrat hinzugefügt werden.
- Komplexere Kernelfunktionen erhöhen den Ressourcenbedarf der Pipelineregion auf dem FPGA, der vergrößert werden muss.
- Sofern die FPGA-Ressourcen ausreichen, könnte die Pipeline mehrfach implementiert werden, um unterschiedliche *work-groups* parallel abarbeiten zu lassen, was den Rechendurchsatz erhöhen würde.
- Das Rahmendesign könnte angepasst werden, um gleichzeitig Rechnungen und Datentransfer zwischen Host und OpenCL-Gerät zuzulassen.
- Die vereinfachte Implementierung der Laufzeitumgebung müsste erweitert werden, um dem vollen OpenCL-Standard zu entsprechen.
- Es gibt einen ICP-Treiber, um unterschiedliche OpenCL-Laufzeitumgebungen nebeneinander anzubieten. Sie müsste implementiert werden, damit ein Programm unterschiedliche OpenCL-Geräte (z.B. GPU und FPGA) verwenden kann.

A. OpenCL FPGA Beispielanwendung

```
1  void TestProgramKernelExecVecAdd(const unsigned char* fileContent, const size_t
       fileSize, bool binary)
2  {
3    cl_int                  status   = CL_SUCCESS;
4    cl_uint                 number   = 0;
5    cl_device_id            device   = NULL;
6    cl_context              context  = NULL;
7    cl_program              program  = NULL;
8    cl_kernel               kernel   = NULL;
9    cl_command_queue        queue    = NULL;
10   cl_mem                  mem1     = NULL;
11   cl_mem                  mem2     = NULL;
12   unsigned int*           h_mem1   = NULL;
13   unsigned int*           h_mem2   = NULL;
14   unsigned int            i        = 0;
15   unsigned int            Z        = 0;
16
17   status = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_ACCELERATOR, 1, &device, &number);
18   if (status != CL_SUCCESS) coutAndExit("Could not find a device within platform. ",
         status);
19
20   context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_ACCELERATOR, NULL, NULL, &
         status);
21   if (status != CL_SUCCESS) coutAndExit("Could not create context from Type. ",
         status);
22
23   if (binary)
24   {
25     program = clCreateProgramWithBinary(context, 1, &device, &fileSize, &fileContent,
           NULL, &status);
26     if (status != CL_SUCCESS) coutAndExit("Could not create program with binary. ",
           status);
27   }
28   else
29   {
30     program = clCreateProgramWithSource(context, 1, (const char**)&fileContent, &
           fileSize, &status);
31     if (status != CL_SUCCESS) coutAndExit("Could not create program with source. ",
           status);
32   }
33
34   status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
35   if (status != CL_SUCCESS) coutAndExit("Could not build program. ", status);
36
```

```
37  kernel = clCreateKernel(program, "kernel", &status);
38  if (status != CL_SUCCESS) coutAndExit("Could not create kernel. ", status);
39
40  queue = clCreateCommandQueue(context, device, 0, &status);
41  if (status != CL_SUCCESS) coutAndExit("Could not create a command queue. ", status)
        ;
42
43  mem1 = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned int) * MEGA, NULL
        , &status);
44  if (status != CL_SUCCESS) coutAndExit("Could not create a memory object. ", status)
        ;
45
46  mem2 = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(unsigned int) * MEGA, NULL
        , &status);
47  if (status != CL_SUCCESS) coutAndExit("Could not create a memory object. ", status)
        ;
48
49  h_mem1 = new unsigned int[MEGA];
50  h_mem2 = new unsigned int[MEGA];
51  for (i = 0; i < MEGA; ++i)
52  {
53    h_mem1[i] = i;
54    h_mem2[i] = 0;
55  }
56
57  status = clEnqueueWriteBuffer(queue, mem1, true, 0, sizeof(unsigned int) * MEGA,
        h_mem1, 0, NULL, NULL);
58  if (status != CL_SUCCESS) coutAndExit("Could not transfer memory", status);
59
60  status = clSetKernelArg(kernel, 0, sizeof(mem1), mem1);
61  if (status != CL_SUCCESS) coutAndExit("Could not set kernel argument", status);
62
63  status = clSetKernelArg(kernel, 1, sizeof(mem1), mem1);
64  if (status != CL_SUCCESS) coutAndExit("Could not set kernel argument", status);
65
66  status = clSetKernelArg(kernel, 2, sizeof(mem2), mem2);
67  if (status != CL_SUCCESS) coutAndExit("Could not set kernel argument", status);
68
69  const size_t global_work_size[3] = { 1, 1, 1 };
70  const size_t local_work_size[3]  = { MEGA, 1, 1 };
71
72  status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
        local_work_size, 0, NULL, NULL);
73  if (status != CL_SUCCESS) coutAndExit("Could not execute kernel", status);
74
75  clEnqueueReadBuffer(queue, mem2, true, 0, sizeof(unsigned int) * MEGA, h_mem2, 0,
        NULL, NULL);
76  if (status != CL_SUCCESS) coutAndExit("Could not transfer memory", status);
77
78  for (i = 0; i < MEGA; ++i)
79  {
80    Z = h_mem1[i] + h_mem1[i];
81
82    if (Z != h_mem2[i])
```

```
83        break;
84    }
85
86    if (i == MEGA)
87      cout << "TestProgramKernelExec success." << endl;
88    else
89      cout << "There is a computational miss match at position: " << i << endl;
90
91    clReleaseMemObject(mem1);
92    clReleaseMemObject(mem2);
93    clReleaseCommandQueue(queue);
94    clReleaseKernel(kernel);
95    clReleaseProgram(program);
96    clReleaseContext(context);
97  }
```

Quelltext A.1: Lauffähige OpenCL-Anwendung auf dem FPGA

Literatur

[1] *Khronos Group OpenGL Standard*. URL: http://www.khronos.org/opengl (1992).

[2] *Microsoft DirectX Developer Center*. URL: http://msdn.microsoft.com/de-de/directx (1995).

[3] *Just-Another Hardware Description Language*. URL: http://www.jhdl.org (1997).

[4] *Open SystemC Initative - Defining and Advancing SystemC Standards*. URL http://www.systemc.org/home/ (1999).

[5] *Impuls Accelerated Technologies*. URL: http://www.impulseaccelerated.com/ (2003).

[6] *Microsoft Programming Guide for HLSL*. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635%28v=VS.85%29.aspx (2003).

[7] *AMD's Close-to-the-Metal*. URL: http://sourceforge.net/projects/amdctm/ (2007).

[8] *clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org/ (2007).

[9] *AMD Developer Centrale*. URL: http://developer.amd.com/sdks/amdappsdk/downloads/pages/default.aspx (2009).

[10] *Apple - OS X Lion - Technical specifications*. URL: http://www.apple.com/macosx/specs.html (2009).

[11] *NVIDIA Developer Zone*. URL: http://developer.nvidia.com/opencl (2009).

[12] *IBM developerWorks - OpenCL lounge*. URL: https:

//www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1 (2010).

[13] *Intel® OpenCL SDK - Intel® Software Network*. URL http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/ (2010).

[14] *Conformt OpenCL Products*. URL http://www.khronos.org/conformance/adopters/conformant-products/ (2011).

[15] *Hardware description language*. URL: http://de.wikipedia.org/wiki/HDLC (2011).

[16] *Learn About Altera's OpenCL Program for FPGAs*. URL: http://www.altera.com/b/opencl.html (2011).

[17] *LLVM Users*. URL: http://llvm.org/Users.html (2011).

[18] *Intel® microprocessor export compliance metrics*. URL: http://www.intel.com/support/processors/xeon/sb/CS-020863.htm (5. Dez. 2008).

[19] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, und Robert Wahbe. *Efficient and language-independent mobile programs*. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, Seite 127–136. ACM, New York, NY, USA (1996). doi:10.1145/231379.231402.

[20] Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), Seite 483–485. ACM, New York, NY, USA (1967). doi:10.1145/1465482.1465560.

[21] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, und R. Uribe. *Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design*. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, Seite 263 – 264 (april 2003). doi:10.1109/FPGA.2003.1227262.

[22] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, und K. Scott Hemmert. *Embedded floating-point units in FPGAs*. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, Seite 12–20. ACM, New York, NY, USA (2006). doi:10.1145/1117201.1117204.

[23] Pavle Belanovic und Miriam Leeser. *A Library of Parameterized Floating-Point Modules and Their Use*. In *Proceedings of the Reconfigurable Computing Is Going Mainstream,*

12th International Conference on Field-Programmable Logic and Applications, FPL '02, Seite 657–666. Springer-Verlag, London, UK, UK (2002).

[24] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, und Pat Hanrahan. *Brook for GPUs: stream computing on graphics hardware*. ACM Trans. Graph., Volume 23:777–786 (August 2004). doi:10.1145/1015706.1015800.

[25] C. Conrad, H. Erfle, P. Warnat, N. Daigle, T. Lörch, J. Ellenberg, R. Pepperkok, und R. Eils. *Automatic identification of subcellular phenotypes on human cell arrays.* Genome Res, Volume 14(6):1130–6 (2004). doi:10.1101/gr.2383804.

[26] Qualis Design Corporation. *VHDL Quick Reference Card*. URL: `http://www.eda.org/rassp/vhdl/guidelines/vhdlqrc.pdf` (1995).

[27] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, und F. Kenneth Zadeck. *Efficiently computing static single assignment form and the control dependence graph*. ACM Trans. Program. Lang. Syst., Volume 13:451–490 (October 1991). doi:10.1145/115372.115320.

[28] M. Flynn. *Some Computer Organizations and Their Effectiveness*. IEEE Trans. Comput., Volume C-21:948+ (1972). doi:10.1109/TC.1972.5009071.

[29] W. Gao und Z. Han. *PCIe SG DMA controller*. URL: `http://opencores.org/project,pcie_sg_dma` (2011).

[30] Wenxue Gao, Andreas Kugel, Reinhard Männer, und Guillermo Marcus. *PCI Express DMA Engine Design*. Technischer Report, CBM Progress Report (2007).

[31] M. Gipp, G. Marcus, N. Harder, A. Suratanee, K. Rohr, R. König, und R. Männer. *Haralick's Texture Features Computations Accelerated by GPUs in Biological Applications*. In *GPU Technology Conference*, GTC '09. NVIDIA, San Joes, California, USA (2009). Poster Research Summit.

[32] Markus Gipp, Guillermo Marcus, Nathalie Harder, Apichat Suratanee, Karl Rohr, Rainer König, und Reinhard Männer. *Haralick's Texture Features using Graphics Processing Units (GPUs)*. In *Proceedings of The World Congress on Engineering 2008*, Volume I von *ICPDC '08*, Seite 587–592. International Association of Engineers, Newswood Limited, London, UK, UK (2008).

[33] Markus Gipp, Guillermo Marcus, Nathalie Harder, Apichat Suratanee, Karl Rohr, Rainer König, und Reinhard Männer. *Haralick's Texture Features Computed by GPUs for Biological Applications*. IAENG International Journal of Computer Science, Volume 36

(2009).

[34] Markus Gipp, Guillermo Marcus, Nathalie Harder, Apichat Suratanee, Karl Rohr, Rainer Koenig, und Reinhard Maenner. *Haralick's Texture Features Computation Accelerated by GPUs for Biological Applications*. In *To appear in Modeling, Simulation and Optimization of Complex Processes, Proceedings of the Fourth International Conference on High Performance Scientific Computing, March 2-6, 2009, Hanoi, Vietnam*, Seite 127–138. Springer-Verlag Berlin / Heidelberg (2011). doi:10.1007/978-3-642-25707-0_11.

[35] Khronos Group. *OpenCL Quick Reference Card*. URL: http://www.khronos.org/files/opencl-quick-reference-card.pdf (2009).

[36] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.48* (6 October 2009).

[37] R. M. Haralick. *Statistical and structural approaches to texture*. Proceedings of the IEEE, Volume 67(5):786–804 (1979). doi:10.1109/PROC.1979.11328.

[38] R. M. Haralick und K. Shanmugam. *Computer Classification of Reservoir Sandstones*. Geoscience Electronics, IEEE Transactions on, Volume 11(4):171–177 (1973). doi:10.1109/TGE.1973.294312.

[39] Robert M. Haralick, K. Shanmugam, und Its'Hak Dinstein. *Textural Features for Image Classification*. Systems, Man and Cybernetics, IEEE Transactions on, Volume 3(6):610–621 (1973).

[40] John Hennessy und David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann (2007).

[41] Allen Holub. *Compiler design in C*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990).

[42] Agility Design Solutions Inc. *Handel-C Language Reference Manual* (2009).

[43] Justin Richardson Kunal Gosrani Siddarth Suresh Jason Williams, Alan D. George. *Computational Density of Fixed and Reconfigurable Multi-Core Devices for Application Acceleration*. RSSI (2008).

[44] Randi Rost John Kessenich, Dave Baldwin. *The OpenGL Shading Language*. URL: http://www.opengl.org/documentation/glsl/ (2002).

[45] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Technischer Report (1979).

[46] Bernd Schwarz Jürgen Reichardt. *VHDL-Synthese, Entwurf digitaler Schaltungen und*

Systeme. Oldenburg Verlag München (2009).

[47] Andreas Kugel, Guillermo Marcus, und Wenxue Gao. *The MPRACE Framework*. URL: http://li5.ziti.uni-heidelberg.de/mprace/.

[48] Chris Lattner und Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, Seite 75. IEEE Computer Society, Washington, DC, USA (2004). doi:10.1109/CGO.2004.1281665.

[49] Chris Lattner und Vikram Adve. *LLVM Language Reference Manual* (2011).

[50] G. Lienhart, G. Marcus Martinez, A. Kugel, und R. Manner. *Rapid Design of Special-Purpose Pipeline Processors with FPGAs and its Application to Computational Fluid Dynamics*. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, Seite 301 –302 (april 2006). doi:10.1109/FCCM.2006.60.

[51] Gerhard Lienhart. *Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren*. Doktorarbeit, Universität Heidelberg (2004).

[52] Mingjie Lin, Ilia Lebedev, und John Wawrzynek. *OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices*. International Conference on Field Programmable Logic and Applications, Volume 0:458–463 (2010). doi:10.1109/FPL.2010.93.

[53] LLVM. *LLVM Demo-Übersetzer Webanwendung*. URL: http://llvm.org (2004).

[54] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, und Roger A. Bringmann. *Effective compiler support for predicated execution using the hyperblock*. SIGMICRO Newsl., Volume 23:45–54 (December 1992). doi:10.1145/144965.144998.

[55] Guillermo Marcus. *Acceleration of Astrophysical Simulations with Special Hardware*. Doktorarbeit, University Heidelberg (2011).

[56] Guillermo Marcus, Wenxue Gao, Andreas Kugel, und Reinhard Manner. *The MPRACE framework: An open source stack for communication with custom FPGA-based accelerators*. In *2011 VII Southern Conference on Programmable Logic (SPL)*, Volume 605, Seite 155–160. IEEE (April 2011). doi:10.1109/SPL.2011.5782641.

[57] Guillermo Marcus, Gerhard Lienhart, Andreas Kugel, und Reinhard Männer. *On Buffer Management Strategies for High Performance Computing with Reconfigurable Hardware*. In *2006 International Conference on Field Programmable Logic and Applications*, i,

Seite 1–6. IEEE (2006). doi:10.1109/FPL.2006.311235.

[58] William R. Mark, R. Steven Glanville, Kurt Akeley, und Mark J. Kilgard. *Cg: a system for programming graphics hardware in a C-like language*. ACM Trans. Graph., Volume 22:896–907 (July 2003). doi:10.1145/882262.882362.

[59] Michael McCool. *Sh-Lib*. URL: http://libsh.org (2003).

[60] Microsoft, Editor. *Phoenix Compiler and Shared Source Common Language Infrastructure*. URL: https://connect.microsoft.com/Phoenix.

[61] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley, Upper Saddle River, NJ, USA (2007).

[62] NVIDIA. *Whitepaper NVIDIA GF100*. URL: http://www.nvidia.com/object/IO_86775.html (2010).

[63] NVIDIA. *Whitepaper NVIDIAs Next Generation CUDA™Compute Architecture Fermi*. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (2010).

[64] NVIDIA. *CUDA™C Best Practices Guide* (Mai 2011).

[65] NVIDIA. *NVIDIA CUDA™C Programming Guide* (Mai 2011).

[66] NVIDIA. *CUDA Occupancy Calculator*. URL: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/tools/CUDA_Occupancy_Calculator.xls (2012).

[67] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, und Wen-Mei W. Hwu. *FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs*. Application Specific Processors, Symposium on, Volume 0:35–42 (2009). doi:10.1109/SASP.2009.5226333.

[68] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, und S. Eggers. *CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures*. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Seite 173 –178 (sept. 2008). doi:10.1109/FPL.2008.4629927.

[69] Daniel J. Quinlan. *ROSE: Compiler Support for Object-Oriented Frameworks*. Parallel Processing Letters, Volume 10(2/3):215–226 (2000).

[70] Satnam Singh. *Computing without Processors*. Queue, Volume 9:50:50–50:63 (June 2011). doi:10.1145/1978542.1978558.

[71] M. A. Tahir, A. Bouridane, F. Kurugollu, und A. A. Amira A. Amira. *Accelerating the computation of GLCM and Haralick texture features on reconfigurable hardware*. In A. Bouridane, Editor, *Image Processing, 2004. ICIP '04. 2004 International Conference on*, Volume 5, Seite 2857–2860 Vol. 5 (2004). doi:10.1109/ICIP.2004.1421708.

[72] David Tarditi, Sidd Puri, und Jose Oglesby. *Accelerator: using data parallelism to program GPUs for general-purpose uses*. SIGOPS Oper. Syst. Rev., Volume 40:325–335 (October 2006). doi:10.1145/1168917.1168898.

[73] Sergios Theodoridis und Konstantinous Koutroumbas. *Pattern Recognition Third Edition*. Academic Press An imprint of Elsevier, San Diego, CA, USA (2006).

[74] J.L. Tripp, K.D. Peterson, C. Ahrens, J.D. Poznanovic, und M.B. Gokhale. *Trident: an FPGA compiler framework for floating-point algorithms*. In *Field Programmable Logic and Applications, 2005. International Conference on*, Seite 317 – 322 (aug. 2005). doi: 10.1109/FPL.2005.1515741.

[75] Justin Tripp, Preston Jackson, und Brad Hutchings. *Sea Cucumber: A Synthesizing Compiler for FPGAs*. In Manfred Glesner, Peter Zipf, und Michel Renovell, Editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, Volume 2438 von *Lecture Notes in Computer Science*, Seite 51–72. Springer Berlin / Heidelberg (2002). doi:10.1007/3-540-46117-5_90.

[76] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, Christine Kitchen, Cheshire Wa Ad, Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, und Christine Kitchen. *An overview of FPGAs and FPGA programming; Initial experiences at Daresbury.* Technischer Report (2006). doi:DL-TR-2006-010.

[77] Xiaojun Wang und Miriam Leeser. *VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware*. ACM Trans. Reconfigurable Technol. Syst., Volume 3:16:1–16:34 (September 2010). doi:10.1145/1839480.1839486.

[78] Markus Wannemacher. *Das FPGA-Kochbuch*. MITP-Verlag (1998).

[79] Xilinx®. *Development System Reference Guide*, v10.1 Edition (2008).

[80] Xilinx®. *Partial Reconfiguration User Guide*, ug702 v12.3 Edition (Oktober 2010).

[81] Xilinx®. *Virtex-6 FPGA Integrated Block for PCI Express User Guide* (2010).

[82] Xilinx®. *Virtex-6 FPGA ML605 Evaluation Kit*. URL: `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm` (2010).

[83] Xilinx®. *DDR2 and DDR3 SDRAM Memory Interface Solution* (2011).

[84] Xilinx®. *LogiCORE IP Floating-Point Operator v5.0* (2011).

[85] Xilinx®. *Virtex-6 FPGA Memory Interface Solutions* (2011).

[86] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, und Jason Cong. *AutoPilot: A Platform-Based ESL Synthesis System*. Seite 99–112 (2008). doi:10.1007/978-1-4020-8588-8_6.