Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Seminararbeit

Konzept zur Beschleunigung von Anwendungen durch FPGAs

Armin Jeyrani Mamegani

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Konzept zur Beschleunigung von Anwendungen durch FPGAs

Armin Jeyrani Mamegani

Seminararbeit
im Studiengang Informatik Master
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuer: Prof. Dr.-Ing. Bernd Schwarz

Abgegeben am 28. Februar 2009

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Bereits seit einigen Jahren hat sich bemerkbar gemacht, dass die Frequenz und damit die Geschwindigkeit von CPUs im PC-Segment aufgrund physikalischer Einschränkungen nicht weiter gesteigert werden kann [14]. Die Hersteller dieser Prozessoren steigern dennoch die Leistungsfähigkeit, indem sie auf die Parallelisierung ihrer Systeme setzen. Da die Frequenzsteigerung nicht mehr in hohem Maß zu erreichen ist [14], bauten die Hersteller Systeme mit zunächst zwei- und mittlerweile mehrkernigen Prozessoren. Im Grunde wurde das Paradigma geändert. Anstatt mit der Absicht voran zu gehen, eine Aufgabe immer schneller abzuarbeiten, wird nun die Absicht gepflegt, mehrere Aufgaben gleichzeitig abzuarbeiten [14]. Damit wurde das parallel programming wiedergeboren und für den PC-Markt lukrativ gemacht. Die CPU ist aber nicht die einzige Einheit in einem PC, die schnelle Berechnungen durchführen kann. In PCs operieren Grafikkarten und wurden bis vor kurzem lediglich für Grafikberechnungen eingesetzt. Somit verfolgen seit einigen Jahren die 2 größten Grafikchiphersteller, AMD (ehemals ATI) und NVIDIA die Idee, ihre Grafikprozessoren für andere Anwendungen zur Verfügung zu stellen. Allerdings unterscheidet sich die Programmierung von Grafikprozessoren (GPUs) von der Programmierung von CPUs [15]. Zum einen haben GPUs einen komplexeren Hardwareaufbau [14]. Zum anderen werden Entwickler, die ihre Programme für CPUs schreiben schon seit vielen Jahrzehnten durch etliche Compiler, APIs, Middleware und Betriebssysteme unterstützt.
OpenGL oder Direct3D sind zwar APIs, die eine Abstraktion der GPUs für Grafikentwickler bieten. Diese sind aber für Renderingoperationen vorgesehen. Andere Operationen wären mit diesen APIs entweder zu aufwändig oder sogar nicht realisierbar. [15]

Somit bieten NVIDIA und AMD ganze Frameworks an, die die Verwendung ihrer Grafikkarten für jegliche Anwendungen vereinfachen sollen. Diese Frameworks bieten eine Plattform, mit der ein Entwickler kaum Neues erlernen muss. NVIDIA setzt mit dem CUDA Framework komplett auf die Sprache C. Der HW-Aufbau der NVIDIA Grafikkarten und das CUDA-System werden im Abschnitt „**Beschleunigung von Anwendungen mit NVIDIA GPUs**“ vorgestellt. AMDs Stream Computing wird in dieser Ausarbeitung nicht vorgestellt, da sowohl der HW-Aufbau als auch das Vorgehen des Frameworks (Stream Computing) sehr ähnlich zu NVIDIA ist. [10] [1]

Dennoch sind diese Plattformen ausschließlich mit den jeweiligen Grafikkarten der o.g. Hersteller nutzbar. Eine von der Khronos Group definierte Sprache verspricht die herstellerunbahängige Nutzung jeglicher Recheneinheiten eines Rechners. Das Open Computing Language (OpenCL) soll als API Entwickler dabei unterstützen, rechenintensive Anwendungen unter Verwendung jeglicher PC-Ressourcen zu entwerfen, ohne die Kenntnis über die jeweilige HW-Konfiguration des einzelnen Nutzers zu haben. Ein kurzer Bericht über OpenCL wird in Abschnitt „**Beschleunigung von Anwendungen durch die Verwendung von OpenCL**“ gegeben. [9]

Die Verwendung von Grafikprozessoren eines PCs ist im Grunde genommen nichts anderes als eine Beschleunigung von rechenintensiven Anwendungen. Eine andere Möglichkeit, Anwendungen zu beschleunigen, ist die Verwendung von rekonfigurierbaren Logikbausteinen, den sogenannten FPGAs. Eine Beschleunigung von rechenintensiven Anwendungen im PC Segment durch ein FPGA wird noch nicht unterstützt. Hierbei liegt die Beschränkung nicht an der mangelnden HW für den PC-Bereich. Viel mehr liegen die Einschränkungen darin, dass zur Konfiguration dieser FPGAs das Lernen von bestimmten Hardwarebeschreibungssprachen notwendig ist und zusätzlich keinerlei Framework, Treiber oder API vorhanden ist, um einen einfacheren Zugang zur Nutzung der Vorteile von FPGAs zu bieten [7]. In Abschnitt „**Ziele dieser Recherche**“ wird ein kurzer Überblick zu FPGAs gegeben. Der Hintergrund dieser Arbeit ist die Idee zur Realisierung einer Plattform zur Verwendung von FPGAs als Beschleuniger für rechenintensive Anwendungen. Diese Idee wird in Anhang A beschrieben. In Abschnitt „**Konzept zur Beschleunigung von Anwendungen durch FPGAs**“ wird dann ein Konzept eines Frameworks vorgestellt, was aus den Kenntnissen über CUDA und OpenCL einen einfacheren Zugang zu FPGAs für Anwendungsentwickler darstellt.

1.2. Ziele dieser Recherche

Die in Handys eingebauten Prozessoren sind bei weitem nicht so leistungsstark, wie die Prozessoren in PCs. Zusätzlich besitzen Handys meist keine oder sehr schwache Grafikprozessoren. [11]
Dennoch sind rechenintensive Anwendungen wie Videobearbeitung(En-/Decodierung) im Handysegment sehr gefragt [12]. Am Ende des Vortrags zu [8] wurde die Idee einer Handyplattform vorgestellt, mit der rechenintensive Anwendungen durch den Einsatz von FPGAs beschleunigt werden können. Diese Idee wird im Anhang A genauer erläutert.

Die Funktionsweise eines FPGAs wird in Anhang B vereinfacht erklärt, um die Vorteile, die ein FPGA in solch einem Fall gegenüber einer CPU hat, näher zu bringen.

Der Einsatz von FPGAs im Embedded-Bereich ist bereits weit verbreitet. Z.B. werden rechenintensive Teile von Anwendungen, die sich komplexer Bildbearbeitungsalgorithmen be-

dienen, auf FPGAs ausgeführt(z.B. Helms Sortierungsmaschinen, Basler Kameras [13]). Auch im Server-Segment gibt es Hersteller, die zu ihren Multiprozessorsystemen FPGAs einsetzen, die ihre Anwendungen beschleunigen [4]. Allerdings werden HW und SW in diesem Bereich komplett getrennt entwickelt [5]. Unter dem Aspekt, dass im Server-Segment höhere Preise für die Produkte verlangt werden, ist der Nachteil der kompletten Trennung bei der Entwicklung bezüglich der höheren Kosten geringer als die Vorteile, die eine FPGA Beschleunigung bringt.
Wie bereits am Anfang dieses Kapitels beschrieben, ist die Idee von einer Plattform für Handys ausgegangen. Bezug nehmend auf die großen Grafikchiphersteller, die auf eine Beschleunigung im PC-Segment setzen, wird die Idee, diese Plattform nur für Handys und Mobilgeräte auszulegen, verbessert. Diese Plattform sollte vielmehr so allgemein sein, dass es sowohl im PC-Segment oder auch in Embedded Computer eingesetzt werden kann. Eine Voraussetzung hierfür ist die in Anhang A beschriebene HW-Architektur (Abbildung A.1). Weiterhin besteht die Einschränkung, dass viele Entwickler FPGA-Hardware nicht kennen und diese auch nicht kennen möchten. Dem Entwickler geht es vielmehr darum, seine Algorithmen in einer höheren, abstrakten Sprache zu beschreiben. Dies würde aber wiederum bedeuten, dass Entwickler sich sowohl mit Programmiersprachen, als auch mit Hardwarebeschreibungssprachen (z.B. VHDL) auskennen müssen.
Eine getrennte Entwicklung bedeutet wiederum höhere Kosten (Einsatz von mehr Entwicklern) und macht zusätzlich die Realisierung von Algorithmen in verschiedenen Sprachen zu komplex. Eine einheitliche Sprache, die durch entsprechende Werkzeuge die Aufteilung in SW und HW vornimmt, wäre eine Beschleunigung der Entwicklung in diesem Bereich. [2]

Hintergrund dieser Arbeit ist ein Konzept für ein Framework zu erstellen, das aus einer Sprache Anwendungen erstellt, die sowohl als Software auf einem Prozessor ausgeführt werden können, als auch Teile davon in HW auf einem FPGA realisiert werden. Hierbei sollen die Kenntnisse, die von den Frameworks der Grafikchiphersteller und OpenCL gewonnen werden in das Konzept mit einfließen. Das Konzept wird in Kapitel 4 vorgestellt.

2. Beschleunigung von Anwendungen mit NVIDIA GPUs

Der Grund für den Einsatz von Grafikprozessoren im PC-Segment liegt darin, dass diese Art von Recheneinheiten in bestimmten Aufgabenbereichen Vorteile gegenüber herkömmlichen CPUs haben. Die Unterschiede zwischen GPU und CPU werden im Anhang E dargelegt. Hauptaugenmerk für diese Ausarbeitung bildet die Programmierbarkeit dieser GPUs.

In der Einleitung (Kapitel 1.2) bereits beschrieben, möchte sich ein Anwendungsentwickler wenig oder am Besten gar nicht mit den Details der verwendeten HW beschäftigen. Dies wird den Entwicklern durch die Bereitstellung von APIs vereinfacht. Bislang war die einzige Möglichkeit auf die Ressourcen eines Grafikprozessors zu zugreifen, die Verwendung von Grafik-APIs wie OpenGL oder Direct3D. Diese sind aber auf Grafikoperationen ausgelegt. Nicht jeder Entwickler ist aber Experte in der Grafikprogrammierung und kann mit Shadern, Texturen und Fragmenten etwas anfangen. [15]

Erfindungen wie die Sprache Brook wurden gemacht, um unter Einsatz dieser Grafik-APIs eine Abstraktion für Entwickler zu bieten. Brook wurde von den Entwicklern der Stanford University als „C with Streams“ vorgestellt. Brook soll den kompletten Verarbeitungsanteil der 3-D-API enthalten, so dass sich die GPU wie ein Koprozessor für parallele Berechnungen darstellt. Dennoch erfordert Brook die Verwendung von herstellerspezifischen Treibern. Brook hat sich bis heute nicht zum Standard durchsetzen können und ist damit für die Veränderungen in diesen Treibern nicht ausgelegt. [3]

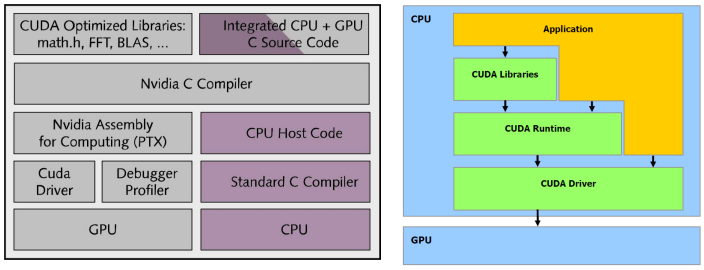


Abbildung 2.1.: Links: CUDA-Entwicklungsplattform. Rechts: CUDA-Softwarestack

Inspiriert von Brook kamen NVIDIA und AMD auf die Idee, ihren Markt zu erweitern und ihre Grafikkarten zur Beschleunigung von Anwendungen anzubieten. Dabei sollten die Entwickler durch ganze Frameworks unterstützt werden. Mit dem **C**ompute **U**nified **D**evice **A**rchitecture (CUDA) schuf NVIDIA ein Framework bestehend aus mehreren APIs, die die bei Brook entstandenen Probleme umgehen können. Wie in Abbildung 2.1 links veranschaulicht, basieren CUDA-Programme auf C und C++ Programmcode. Weiterhin bietet die CUDA-Plattform C/C++ Entwicklungstools, Funktionsbibliotheken und einen Hardwareabstraktionsmechanismus, das die GPU-HW für den Entwickler kapselt. Obwohl CUDA das Schreiben von speziellem Code für das parallele Programmieren erfordert, ist das explizite Verwalten von Threads im herkömmlichen Sinne nicht notwendig. Dies vereinfacht die Programmierung. CUDA Entwicklungswerkzeuge verwenden nebenher herkömmliche C/C++ Compiler, sodass Entwickler GPU-Code und CPU-Code vereinen können. NVIDIA hat schon immer die Architektur seiner GPUs unter APIs versteckt, was 2 Vorteile mit sich bringt. Zum einen werden Entwickler von den HW-Details der GPU isoliert. Zum anderen ermöglicht HW-Abstraktion durch APIs, dass die darunter liegende Architektur geändert werden kann ohne Einflüsse auf bestehenden Programmcode zu haben. Wie in Abbildung 2.1 rechts gezeigt, bietet CUDA Applikationsentwicklern den Zugriff auf die GPU über mehrere Abstraktionsebenen. Die oberste Ebene sind die mit CUDA mitgelieferten Bibliotheken für einfache mathematische Operationen (math.h), komplexere FFT-Berechnungen (FFT.h) und Operationen für lineare Algebra (BLAS.h). Es ist anzunehmen, dass bei erfolgreicher Verbreitung des CUDA-Systems auch die Anzahl an neuen Bibliotheken steigen wird. Weiterhin hat ein Entwickler die Möglichkeit, über die Funktionen der Runtime-API auf eine oder mehrere Recheneinheiten der GPU zu zugreifen und diese vom Host aus zu steuern. Die Driver-API ist in der Programmierung zwar aufwendiger, bietet aber einen besseren Zugriff auf die HW und eine bessere Kontrolle über die GPU. [10]

Abbildung 2.2 zeigt ein Programmierbeispiel in C, das eine SAXPY-Operation (Skalar Alpha * Y + X) einmal als serielle Ausführung auf einer herkömmlichen CPU und als parallele Ausführung auf einer NVIDIA GPU beschreibt. Mit der CUDA Erweiterung *__global__* wird dem C/C++ Parser mitgeteilt, dass *saxpy_parallel* für die Ausführung auf der GPU definiert ist und mit dem NVIDIA Compiler compiliert wird. Die andere CUDA Erweiterung in den drei spitzen Klammern spezifiziert die Ausführung und Verteilung der einzelnen Berechnungen auf den Ausführungseinheiten der GPU. Im Vergleich zur seriellen Ausführung auf einer CPU wird mit den NVIDIA GPUs die Berechnung auf die vielen Ausführungseinheiten parallelisiert und damit die Berechnungszeit je nach Anzahl Vektorelemente und Ausführungseinheiten verkürzt. Als Beispiel sei eine GPU mit 256 Ausführungseinheiten gegeben, mit der eine Vektorberechnung mit jeweils 512 Vektorelementen durchgeführt werden soll. Für die Berechnung mit einer herkömmlichen CPU beträgt die gesamte Ausführungszeit 512 Schleifendurchgänge. Die gesamte Ausführungszeit der Berechnung per GPU beträgt 2 Zyklen, die jeweils vergleichbar mit der Ausführungszeit eines Schleifendurchgangs sind. Selbst mit

einem 4-Kern-Prozessor würde diese Berechnung immernoch die Ausführungszeit von 128 einzelnen Schleifendurchgängen beanspruchen. [6]

```
Computing y _ ax + y with a serial loop:
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

Computing y _ ax + y in parallel using CUDA:
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}
// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Abbildung 2.2.: CUDA Codebeispiel in C. Eine SAXPY-Operation ausgeführt als ein Thread auf einer herkömmlichen CPU (oben) und dieselbe Operation als multithreaded Version auf einer NVIDIA GPU (unten).

Trotz der vielen Erleichterungen, die NVIDIA Entwicklern zur Verfügung stellt, um NVIDIA GPUs zur Beschleunigung ihrer Anwendungen zu nutzen, sind da zwei Gesichtspunkte, die nicht außer Acht gelassen werden dürfen. Obwohl NVIDIA für ihr CUDA-System auf die Sprache C setzt, ist es nicht so, dass Entwickler einfach nur irgendwelche Bibliotheken verwenden. Spezielle Erweiterungen müssen angewandt werden, um die korrekte Handhabung des CUDA-Systems zu garantieren. Weiterhin bietet CUDA keinen Automatismus, mit der die beste Auslastung der HW bzw. Performanz einer Anwendung bestimmt werden kann. Diese Analyse ist immernoch dem Entwickler vorbehalten. D.h., dass eine geeignete Partition einer Anwendung vom Entwickler vorgenommen werden muss. NVIDIA kann mit der Architektur des CUDA-Systems garantieren, dass Anwendungen beschleunigt werden können und, dass die HW-Architektur ihrer GPUs keinen Einfluss auf den Programmcode hat. Dennoch ist CUDA kein Standard. Dies bedeutet, dass Anwendungen unter Benutzung von CUDA nur mit NVIDIA Grafikkarten funktionieren. D.h. eine Beschleunigung funktioniert nur dann, wenn eine NVIDIA Grafikkarte vorhanden ist. CUDA ist somit keine allgemein gültige Sprache oder System, mit dem eine Beschleunigung durch jegliche, zusätzlich zur CPU vorhandene Berechnungseinheit erreicht werden kann.

3. Beschleunigung von Anwendungen durch die Verwendung von OpenCL

In diesem Kapitel wird OpenCL vorgestellt. Da die letzte OpenCL-Spezifikation erst kürzlich verabschiedet wurde und die genauen Details den Rahmen dieser Ausarbeitung nicht sprengen sollen, wird hier ausschließlich auf die wichtigsten Eigenschaften sowie die Vorteile von OpenCL gegenüber NVIDIA CUDA eingegangen.

Das **Open C**omputing **L**anguage ist eine standardisierte API und wurde von der Khronos Group spezifiziert. Mit dieser API ist es möglich, alle Recheneinheiten eines Systems für eine Anwendung zu nutzen. Die API erlaubt dabei sowohl Taskparalleles Computing, als auch Datenparalleles Computing wie von der Grafikprogrammierung bekannt. Dabei ist OpenCL C-basiert und bietet Erweiterungen für das parallele Programmieren. Eine Abstraktion der HW-Details ähnlich des CUDA-Systems wird auch von OpenCL verfolgt. Ein Software-Stack bestehend aus Platfom-Layer, einer Runtime und einem Compiler gehört zum OpenCL Umfang. Mit dem Platform-Layer bietet OpenCL Funktionen, mit denen Berechnungseinheiten eines Systems entdeckt, ausgewählt und initialisiert werden können. Die Runtime ermöglicht das Handhaben der Ressourcen und das Ausführen von Code (Kernels) auf den Berechnungseinheiten. Der Compiler unterstützt eine Untermenge der ISO C99 Sprache und erzeugt Ausführungsprogramme aus dieser Sprache. [9]

Im Gegensatz zu CUDA kann mit OpenCL jegliche Recheneinheit eines Systems zur Beschleunigung von Anwendungen genutzt werden. Jedoch verwendet auch OpenCL darunter liegende spezifische Treiber. D.h., dass diese Treiber mit dem OpenCL Standard konform sein müssen. Ist einmal Programmcode unter Verwendung von OpenCL für eine Anwendung geschrieben, ist es nicht mehr von Bedeutung, ob die verwendete GPU von NVIDIA, AMD oder einem anderen Hersteller kommt. Jedoch müssen die Treiber dieser GPUs für eine Verwendung von OpenCL ausgelegt sein. Ein weiterer Vorteil ist, dass selbst wenn die im Programmcode eingeplante Berechnungseinheit nicht vorhanden ist, die Ausführung des Programmes auf der Hauptrecheneinheit eines Computersystems, also der CPU ausgeführt wird. [9]

In Kapitel 1.2 wurde die Idee der Beschleunigung durch ein FPGA erläutert. Die Verwendung von OpenCL für die Beschleunigung von Anwendungen mit einem FPGA wird nicht ohne weiteres funktionieren. Ein einfacher Treiber wird in diesem Fall nicht reichen. Neben

einem Treiber ist es notwendig, aus den Teilen einer Anwendung, die mit OpenCL geschrieben sind und für die Ausführung auf einem FPGA hervorgesehen sind, eine Konfigurationsdatei für das FPGA zu erzeugen, mit der dieselbe beabsichtigte Funktion auf einem FPGA realisiert werden kann (siehe Kapitel 1.2: Funktionsweise von FPGAs). Weiterhin ist über ein FPGA jegliche Kommunikationsform realisierbar. Dies bedeutet, dass auch ein einheitliches Kommunikationsmodell benötigt wird.

4. Konzept zur Beschleunigung von Anwendungen durch FPGAs

In diesem Kapitel wird das Konzept eines Frameworks vorgestellt, das aus einer Sprache Anwendungen erstellt, wovon einige Teile als Software auf einem Prozessor ausgeführt, und andere Teile davon in HW von einem FPGA beschleunigt werden. Im Folgenden werden diese Anwendungen FPGA-Anwendungen genannt.

In Abbildung A.1 wurde die grobe Architektur gezeigt, die notwendig ist, um ein Basissystem für eine FPGA-Beschleunigung zu garantieren. In Abbildung 4.1 ist eine Verfeinerung dieser Architektur abgebildet. FPGA und CPU kommunizieren über ein Peripheriebus. D.h., dass bei der Entwicklung einer HW-Plattform auf ein im PC-Segment verbreitetes Bussystem zurückgegriffen werden muss. Damit Komponenten in einem FPGA überhaupt die Möglichkeit haben, mit der CPU zu kommunizieren, muss ein Kommunikationskontroller implementiert werden, das die Kommunikation z.B. mit dem PCIe-Bus handhabt. Weiterhin werden auf dem FPGA Komponenten realisiert, die die einzelnen Operationen ausführen sollen. Diese müssen wiederum mit dem Kommunikationskontroller kommunizieren.

Da die Realisierung eines Busses auf einem FPGA mit herkömmlichen Leitungen nicht funktioniert, wird ein Pseudobus eingesetzt . Dieser Pseudobus besteht aus einer Schaltung von mehreren Multiplexern und Demultiplexern, die über eine Selektionsleitung festlegen, welche Komponente adressiert werden soll. [16]

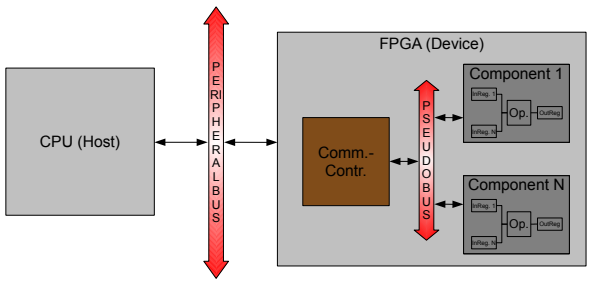


Abbildung 4.1.: Verfeinerung der CPU-FPGA-Verbindung

Die einzelnen Komponenten werden wie in Kapitel 1.2 am Beispiel der Vektoraddition gezeigt, realisiert. Detailiert bedeutet dies, dass zwischen je zwei Eingangsregister(Operanden) und einem Ausgangsregister(Ergebnis) eine Logik die jeweilige mathematische Operation ausführt. [17]

Eine FPGA-Anwendung besteht aus 2 Teilen (siehe Abbildung 4.2). Ein Teil dieser Anwendung ist die reine Softwarepartition, die auf der CPU ausgeführt wird. Der zweite Teil ist eine Konfigurationsdatei für das FPGA, also die Hardwarepartition dieser Anwendung.

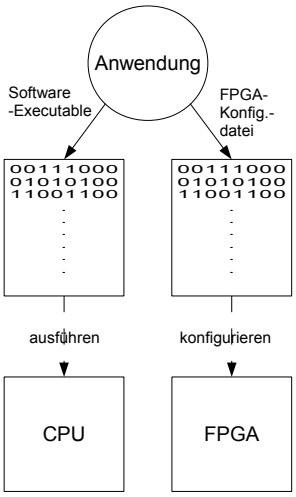


Abbildung 4.2.: Komponenten einer FPGA-Anwendung

Die Erzeugung dieser beiden Teile einer FPGA-Anwendung erfordert die Verwendung von verschiedenen Werkzeugen. Für die Erstellung eines Executables zur Ausführung auf einem Prozessor ist zuvor ein Design in einer Programmiersprache wie C, C++ oder ähnliches notwendig. Dieses Design muss dann von einem Compiler für das jeweilige System compiliert werden.

Für eine Hardwarekonfiguration wird zuvor ein Design in einer Hardwarebeschreibungssprache wie VHDL oder Verilog erstellt. Dieses muss dann mit dem für das verwendete FPGA geeignete Synthesewerkzeug des jeweiligen FPGA Herstellers synthetisiert werden (siehe Anhang C für die Erklärung von Synthese). [7]

Aus der Recherche zu CUDA und OpenCL resultiert, dass die Konstrukteure dieser beiden Systeme auf die Sprache C als einzige Modellierungssprache setzen, da diese Sprache bereits sehr weit verbreitet ist und von vielen Entwicklern beherrscht wird. Die auf GPUs ausgeführte Software wird nicht mit demselben Compiler wie die Software für eine CPU compiliert. CUDA teilt dabei den C-Code in Bereiche auf und compiliert die verschiedenen Bereiche mit den passenden Compilern. Für das hier vorgestellte Framework wird auch die Sprache C als

einzige Modellierungssprache dienen. Im Unterschied zu CUDA wird bei diesem Framework nicht auf eine explizite Aufteilung des Codes gesetzt. Dies macht das Lernen von speziellen Erweiterung überflüssig. Vielmehr ist ein essentieller Bestandteil dieses Frameworks eine Bibliothek für einfache mathematische Operationen. Diese Bibliothek wird in der ersten Phase einfache Additions- und Subtraktionsoperationen anbieten. Dennoch ist eine Aufteilung bzw. die Erzeugung von 2 Codeeinheiten unumgänglich. Wie bereits oben erwähnt, wird für eine FPGA-Anwendung jeweils eine Datei für CPU und eine für das FPGA benötigt. In Anhang D wird die Umsetzungsstrategie zur Erzeugung der jeweiligen Dateien für CPU und FPGA erläutert.

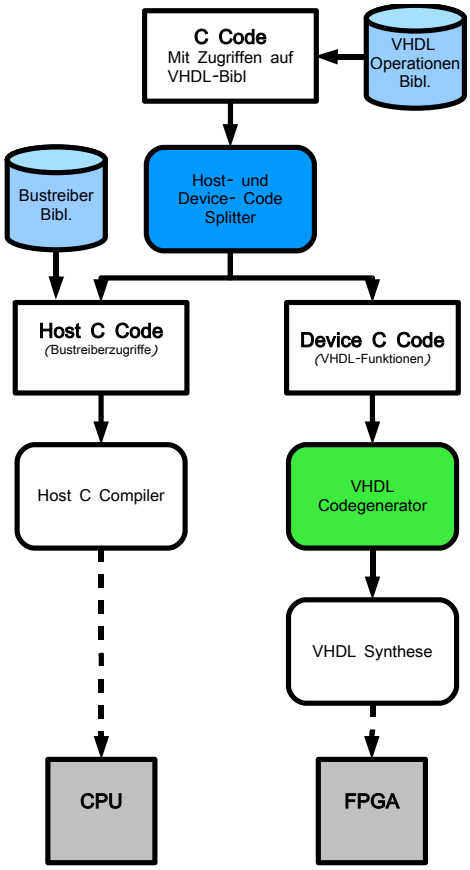


Abbildung 4.3.: Ablauf der Anwendungserzeugung

Abbildung 4.3 zeigt den Ablauf zur Erzeugung einer FPGA-Anwendung. Tabelle 4.1 listet alle Komponenten des in Abbildung 4.3 dargestellten Ablaufs auf und beschreibt diese.

Die beschriebenen Komponenten sind Bestandteile der Entwicklungsphase einer FPGA-Anwendung. Diese sollen gestartet und ausgeführt werden und haben daher ein Laufzeitverhalten.

Komponente	Funktion
C-Code	Eigentliches Design. Ausschließlich dieser Code wird vom Entwickler modelliert.
VHDL-Bibliothek	C-Bibliothek mit Funktionen für einfache mathematische Operationen, die auf einem FPGA realisiert werden sollen.
Host- und Device-Code Splitter	Ein Programm oder Skript, das aus dem C-Code ein Host C-Code mit den jeweiligen Bustreiberzugriffen und ein Device C-Code erzeugt.
Bustreiber-Bibliothek	Funktionen für die Zugriffe auf den jeweilig verwendeten Peripheriebus.
Host C-Code	Aus dem vom Entwickler entworfenem C-Code generierter Code zur Ausführung auf der CPU.
Host C-Compiler	C-Compiler für das Instruktionsset der CPU.
Device C-Code	Aus dem vom Entwickler entworfenem C-Code extrahierte FPGA-Funktionen.
VHDL-Codegenerator	Setzt die verwendeten Funktionen aus dem Device C-Code in VHDL Komponenten um.
VHDL-Synthese	Synthesewerkzeug zur Erzeugung der Konfigurationsdatei für das FPGA.

Tabelle 4.1.: Komponenten des Frameworks

Der Ablauf zur Laufzeit sieht folgendermaßen aus:

1. FPGA-Anwendung wird vom Benutzer gestartet.
2. FPGA wird automatisch mit der bereits vorhandenen Konfigurationsdatei geladen.
3. Sobald das FPGA bereit ist, startet das Hostprogramm und das FPGA ist in der Lage Daten zu empfangen.
4. Kommunikation zw. Hostprogramm und FPGA erfolgt durch Lesen und Beschreiben von Registern.
5. Wird die FPGA-Anwendung beendet, steht das FPGA anderen FPGA-Anwendungen zur Verfügung.

Um diesen Laufzeitanforderungen zu genügen, ist zusätzlich eine FPGA-Runtime-Komponente zu realisieren, das dieses Laufzeitverhalten ermöglicht. Diese Runtime-Komponente ist als Software zur Ausführung auf dem Prozessor zu implementieren.

5. Zusammenfassung und Ausblick

In dieser Ausarbeitung wurde ein Konzept zur Realisierung einer Plattform zur Verwendung von FPGAs als Beschleuniger für rechenintensive Anwendungen vorgestellt. Zunächst wurde in Kapitel 1.1 ein Überblick über den aktuellen Stand der Anwendungsbeschleunigung durch Grafikprozessoren und die bei der Entwicklung unterstützenden Frameworks gegeben. In Bezug auf diese Frameworks wurde im selben Kapitel ein Konzept für ein Framework zur Anwendungsbeschleunigung durch FPGAs als Hintergrund für diese Arbeit gegeben.

In Kapitel 1.2 wurde ein Konzept für ein Framework, das aus einer Sprache Anwendungen erstellt, die sowohl als Software auf einem Prozessor ausgeführt werden können, als auch Teile davon in HW auf einem FPGA realisiert werden, als Ziel dieser Recherche definiert. Die Frameworks NVIDIA CUDA und OpenCL wurden als Referenz für dieses Konzept genannt.

In Kapitel 2 wurde eine Übersicht zur Funktionsweise von NVIDIA GPUs und des CUDA-Frameworks gegeben. Die Vorteile, wie die Verwendung von APIs und der Sprache C, sowie der Beschleunigung von Anwendungen durch Parallelisierung wurden hervorgehoben und als Bestandteile des o.g. Konzeptes berücksichtigt. Die Aufteilung des Codes durch spezielle Erweiterungen der Sprache C, wie es von CUDA vorgenommen wird, wurde als nachteilig bewertet und resultiert in eine andere Strukturierung für das Framework zur FPGA-Beschleunigung.

OpenCL wurde als herstellerunabhängiges Framework für die Beschleunigung von Anwendungen durch jegliche Recheneinheiten eines PCs in Kapitel 3 präsentiert. Es wurde festgestellt, dass die Verwendung von OpenCL zur Beschleunigung von Anwendungen durch FPGAs zwar nicht ausgeschlossen ist, hierfür aber zusätzliche Werkzeuge notwendig sind, um eine OpenCL-Spezifikation in eine Hardwarebeschreibung umzusetzen.

Am Anfang von Kapitel 4 wurde eine schematische HW-Architektur mit den notwendigen Komponenten aufgezeigt, die eine Basis für die Kommunikation zwischen CPU und FPGA darstellt. Weiterhin wurden die Komponenten einer FPGA-Anwendung und die Grobstrategie zur Erzeugung dieser Komponenten erläutert. Die einzelnen Komponenten des Frameworks für FPGA-Anwendungen wurden in Tabelle 4.1 aufgelistet und ihre jeweilige Funktion erklärt.

Zusätzlich wurden die einzelnen Schritte des Ablaufs des Laufzeitverhaltens einer FPGA-Anwendung benannt. Die Notwendigkeit einer Runtime-Komponente zur Realisierung dieses Laufzeitverhaltens wurde dargelegt.

Das in dieser Ausarbeitung beschriebene Konzept wird als Fundament für die Masterarbeit dienen. Dabei wird mit der Realisierung der VHDL-Bibliothek begonnen. Die Erstellung eines Codesplitters und eines Codeumsetzers soll den nächsten Schritt der Masterarbeit bilden. Diese sollen dann mit der VHDL-Bibliothek als Input und einer Beispielapplikation validiert werden. Schließlich soll die FPGA-Runtime verwirklicht und zur Ausführung gebracht werden. Eine Umgebung, die den in Abbildung 4.3 gezeigten Ablauf automatisiert, bildet den letzten Teil dieses Frameworks. Durch die Wahl und Realisierung einer geeigneten Referenzanwendung soll dieses Framework analysiert werden. Die Unterschiede sowie die Vor- und Nachteile zu CUDA sollen das Ergebnis dieser Analyse bilden und innovative Kenntnisse im Bereich der Beschleunigung durch FPGAs liefern.

Literaturverzeichnis

[1] AMD. Amd stream computing. User Guide Version 1.2.1, 10 2008.

[2] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification*. Morgan Kaufmann, 2007.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. White Paper, Computer Science Department, Stanford University, 2004.

[4] DRC Computer Corporation. Drc reconfigurable processor unit. Technical Specification, 04 2007.

[5] Michael D'Amour. Standards-based reconfigurable computing for hpc. *HPC wire*.

[6] Tom R. Halfhill. Parallel processing with cuda. *Microprocessor Report*, 2008.

[7] Steve Kilts. *Advanced FPGA Design*. Wiley & Sons, 2007.

[8] Armin Jeyrani Mamegani. Framework für den soc entwurf. Vortrag zu Anwendungen 1 im Studiengang Master Informatik an der HAW Hamburg, 06 2008.

[9] Aaftab Munshi. The opencl specification. Specification Version 1.0, 12 2008.

[10] NVIDIA. Nvidia cuda: Compute unified device architecture. Programming Guide Version 1.1, 11 2007.

[11] Chris Rowen. *Engineering the Complex SoC*. Pearson Education, 2004.

[12] Thomas C. Schmidt and Matthias Wählisch. MOVIECAST - Mobile Video Lösungen für SIP–initiierte Multicast Gruppenkonferenzen. In J. Sieck and M. Herzog, editors, *Wireless Communication and Information*, pages 75–100, Boizenburg, 2008. Verlag Werner Hülsbusch.

[13] Basler Vision Technologies. Basler ip kamera bip-1600c. Technical Specification, 01 2009.

[14] Thomas Rauber und Gudula Rünger. *Parallele Programmierung*. Springer, 2007.

[15] Bernhard Eberhardt und Jens-Uwe Hahn. *Kompendium Medieninformatik: Medienpraxis*. Springer, 2007.

[16] Prof. Dr.-Ing. Bernd Schwarz und Prof. Dr. Jürgen Reichardt. *VHDL Synthese*. Oldenbourg, 2007.

[17] Wayne Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, 2008.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A.

Idee zur Beschleunigung rechenintensiver Anwendungen

In diesem Anhang wird die Idee aus [8] genauer erläutert. Diese Idee sieht einen HW-Aufbau vor, bestehend aus einem Prozessor, einer rekonfigurierbaren Einheit, Speicher und einem Bussystem, worüber diese Komponenten kommunizieren können (Abb. A.1).

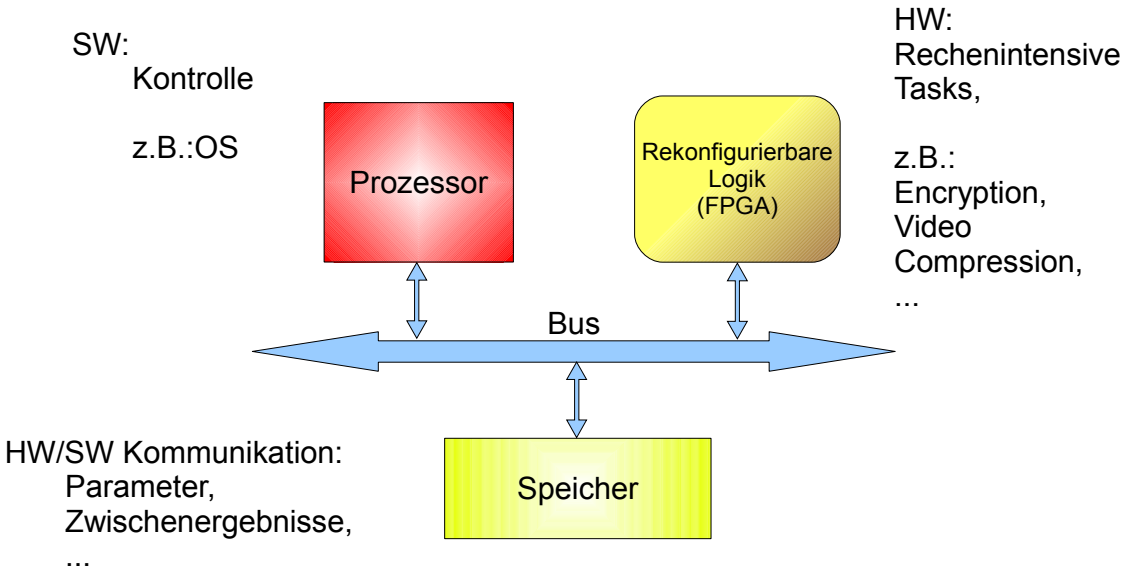


Abbildung A.1.: Schematischer HW-Aufbau einer CPU in Verbindung mit einem FPGA über ein Bussystem

Dabei ist vorgesehen, dass die eigentliche Anwendung auf dem Prozessor ausgeführt wird. Teile dieser Anwendung, die durch eine Realisierung in HW in einer Beschleunigung ei-

ner Anwendung resultieren, werden im FPGA ausgeführt. Es ist notwendig, dass diese rechenintensiven Teile in einer Hardwarebeschreibungssprache implementiert werden. Dieses muss dann in Abhängigkeit vom verwendeten FPGA Baustein von einem Synthesewerkzeug (Compiler für FPGAs, z.B. Xilinx ISE) in eine Konfigurationsdatei für das FPGA umgesetzt werden und in das FPGA geladen werden. Zusätzlich bietet ein am Bussystem angeschlossener Speicher eine Ablagemöglichkeit für Parameter oder größere Datenpakete zwischen CPU und FPGA.

Anhang B.

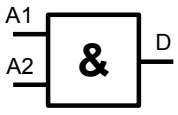
Funktionsweise von FPGAs

FPGAs sind im Grunde genommen nichts anderes als Speicher. Der Speicher ist in Speicherbereiche aufgeteilt, die mit Adressleitungen versehen sind und eine Stelle mit einer Adresse beschrieben bzw. ausgelesen werden kann. Diese Speicherbereiche können als sogenannte **L**ook **U**p **T**ables (LUT) angesehen werden. Als Beispiel wird hier eine 2-zu-1-LUT gezeigt (Abb. B.1).

A1	A2	D
0	0	
0	1	
1	0	
1	1	

Abbildung B.1.: Look Up Table mit leeren Speicherstellen

Diese LUT hat also 4 Speicherstellen, wo insgesamt 4 Bit abgespeichert werden können. Jedes Bit kann mit 2 Adressleitungen adressiert werden. Möchte man nun im FPGA ein logisches UND konfigurieren, dann wird die LUT wie in Abbildung B.2 geladen.



A1	A2	D
0	0	0
0	1	0
1	0	0
1	1	1

Abbildung B.2.: Look Up Table als UND Funktion

Eine LUT wird hierbei einmal beschrieben (konfiguriert). Danach wird nur noch gelesen. Je nachdem, welche Bitkombination an den Adressen anliegt, wird der für eine UND Operation richtige Wert ausgegeben. Diese LUTs werden je nach Kapazität des FPGAs durch ein komplexes Routingverfahren zusammengeschaltet und ergeben damit eine große digitale Schaltung. Es ist zu beachten, dass der Zeitbedarf einer solchen UND Operation mit dem von einem einzigen Prozessorzyklus vergleichbar ist. Die Vorteile ergeben sich aber dadurch, dass mit einem FPGA mehrere solcher LUT-Ketten gebildet werden können. Damit sind keine Register- oder Speicheroperationen notwendig. Noch mehr auf die Geschwindigkeit wirkt sich aus, dass in einem FPGA parallele Ketten gebildet und somit viele Berechnungen gleichzeitig gemacht werden können. Als Beispiel soll eine Vektoraddition dienen.

$$x_1,...,x_n + y_1,...,y_n$$

Der Programmcode würde als Beispiel in der Sprache C folgendermaßen aussehen:

```
for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
```

In einem System mit einem einzigen CPU-Kern würde der gesamte Zeitbedarf für die Berechnung dem n-fachen der Ausführungszeit eines einzigen Schleifendurchganges betragen. Im FPGA könnte diese Berechnung komplett parallelisiert werden. Da alle Operanden voneinander unabhängig sind, würde dies schematisch wie in Abbildung B.3 aussehen.

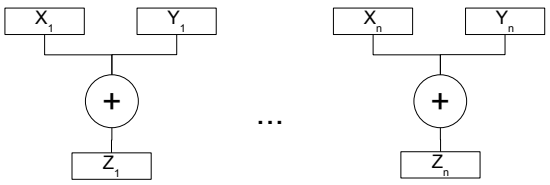


Abbildung B.3.: Parallelisierte Vektoraddition

Anhang C.

VHDL-Synthese

Abbildung C.1 links zeigt die einzelnen Schritte zur Erzeugung eines ausführbaren Programmes aus der Programmiersprache C. Rechts sind die einzelnen Schritte beim Hardwareentwurfsprozess aufgezeigt. Die Spezifikation eines Entwurfs in der Hardwarebeschreibungssprache VHDL wird anhand eines Synthesewerkzeuges (z.B. Xilinx ISE) durch eine Verhaltenssynthese in die Register-Transfer-Level Beschreibung umgesetzt. Aus dieser RT-Beschreibung werden durch die RT-Synthese Logikgleichungen erzeugt. Aus diesen wird wiederum anhand der Logiksynthese eine Beschreibung auf Logikgatter-Ebene errechnet. Diese Logikgatter-Beschreibung wird dann schließlich zu einer Konfigurationsdatei für das jeweilige FPGA umgesetzt.

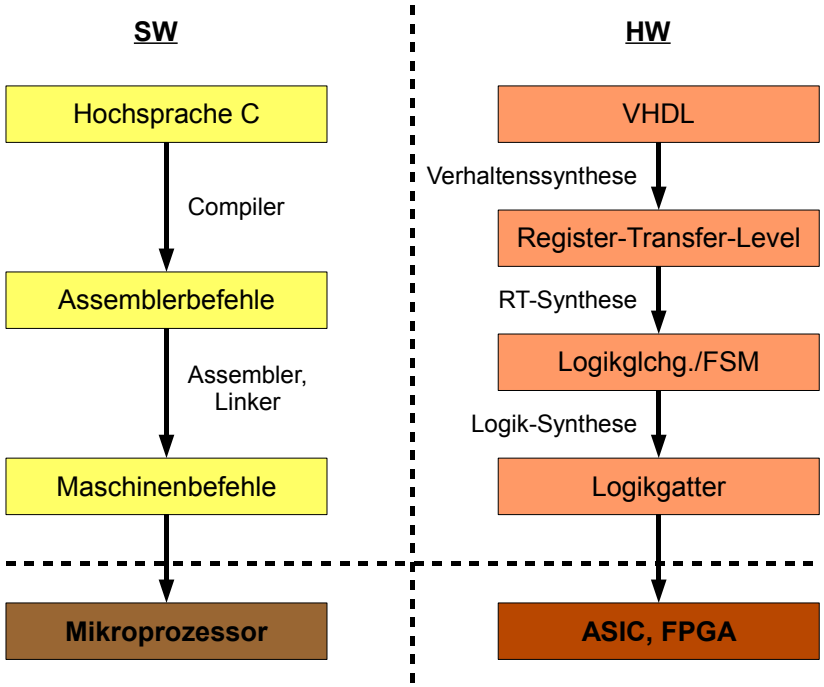


Abbildung C.1.: Entwurfsprozess. Links: Software. Rechts: Hardware

Anhang D.

Strategie zur Umsetzung in Host C-Code und Device Code

An dieser Stelle wird anhand eines Pseudocodebeispiels dargestellt, wie eine Funktion zur Implementierung auf einem FPGA aussehen soll und die die beiden umgesetzten Codeeinheiten.

Unten aufgeführt ist ein Pseudocode, dass aus dem einheitlichen Design eine FPGA-Funktion aufruft.

```
// Codeausschnitt aus dem einheitlichen Design
unsigned char x, y, z;
x = 21;
y = 140;
// Funktionsaufruf aus der VHDL-Bibliothek
z = fpga_add8(x, y);
...
```

Aus diesem Codeabschnitt wird zunächst ein Code für die CPU erzeugt.

```
// Aus dem einheitlichen Design erzeugtes C-Code für die CPU
unsigned char x, y, z;
x = 21;
y = 140;
// Senden der Variablen per PCIe an das FPGA
PCIDriver pciDriver("PortXY");
pciDriver.write(adr, x);
pciDriver.write(adr + 1, y);
pciDriver.read(adr + 2, &z);
...
```

Der nächste Schritt ist dann das Erzeugen einer VHDL-Komponente, die eine Additionsoperation mit zwei Operanden und einem Ergebnis durchführt. Unten wird der Pseudocode dafür aufgeführt.

```
ENTITY add8 IS
  PORT (
    a           : IN  std_logic_vector(7 downto 0);
    b           : IN  std_logic_vector(7 downto 0);
    c           : OUT std_logic_vector(7 downto 0));
END ENTITY add8;

ARCHITECTURE adding OF add8 IS
BEGIN
// interne Signal- und Registerdeklarationen
...
// Prozesse zur Registerrealisierung
...
// nebenläufige Berechnung
c <= a + b;

END adding;
```

Es ist zu beachten, dass bei der VHDL-Generierung die aufrufbaren Funktionen zu VHDL-Vorlagen zugeordnet sind. D.h. für eine C-Funktion add... wird ein bereits vordefinierter VHDL-Code substituiert.

Anhang E.

Unterschiede zwischen GPU und CPU

In diesem Anhang wird veranschaulicht, wo sich GPU und CPU unterscheiden.
CPUs operieren auf einem Befehlssatz. D.h. in einem Takt sollen je nach Anzahl Kernen und Parallelisierungsgrad ein oder mehrere Befehle abgearbeitet werden. Dieser Befehlssatz kann auf unterschiedlichen Daten operieren. Der Trend geht bei CPUs also dahin, dass bei Erhöhung der Kerne immer mehr Instruktionsparallelismus entsteht. Dennoch ist die Koordination von Befehlen auf die verschiedenen Verarbeitungskomponenten sehr komplex. Daher macht die blinde Erhöhung dieser Komponenten keinen Sinn. GPUs hingegen sind darauf ausgelegt, auf einem Pixelsatz zu operieren. Diese Pixel sind Daten, die von einander unabhängig sind und daher von parallelen Komponenten verarbeitet werden können. Dabei ist es meistens so, dass auf allen Daten in einem Datensatz dieselbe Operation ausgeführt werden muss. D.h. alle Recheneinheiten führen dieselbe Operation auf verschiedenen Daten aus. Eine Koordination dieser Operationen ist demnach weitaus weniger komplex. Daher kann bei einer GPU ein erheblicher Teil der Chipfläche für die Recheneinheiten verwendet werden und nicht für die Koordinationseinheit. Abbildung E.1 stellt schematisch dar, wie die Aufteilung der Chipfläche zu verstehen ist.

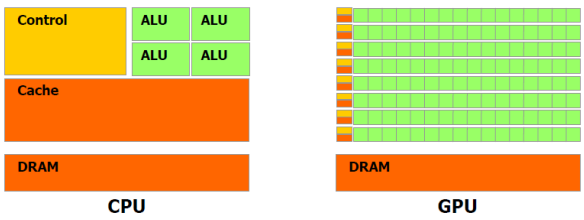


Abbildung E.1.: Schematischer HW-Aufbau einer CPU und einer GPU

Dabei ist das parallele Berechnungsverfahren wie die Erklärung in Anhang B über die Parallelisierung mit FPGAs zu verstehen.