Institut für Technische Informatik
Universität Stuttgart
Pfaffenwaldring 47
70569 Stuttgart

Fachstudie Nr. 111

Implikationen moderner Many-Core-Architekturen auf die Abbildung von Algorithmen aus dem EDA-CAD-Bereich

Matthias Müller, Jochen Puff, Mark Silberberger

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inform. Claus Braun, Dipl.-Inform. Stefan Holst
begonnen am:	01. Mai 2009
beendet am:	30. November 2009
CR-Klassifikation:	C.1.2, B.7.2

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Überblick

Diese Arbeit beginnt in *Kapitel 1* mit einer generellen Motivation und Einleitung in die Problemstellung. Danach werden in *Kapitel 2* verschiedene Hardwarearchitekturen von skalaren Prozessoren bis hin zu Stream-Prozessoren aktueller Grafikkarten vorgestellt, diese in Klassen eingeteilt und deren Vor- und Nachteile erläutert. In *Kapitel 3* folgt dann ein kurzer Überblick über, zum Zeitpunkt dieser Arbeit aktuell vorhandene, Grafikhardware von NVIDIA und AMD/ATI. *Kapitel 4* zählt bedeutende Programmierumgebungen auf, die Grafikkarten als generalisierte Rechenmaschinen nutzbar machen. Dabei liegt der Schwerpunkt auf NVIDIA CUDA. Darauf aufbauend zeigt *Kapitel 5* die Besonderheiten von Many-Core-Architekturen im Hinblick auf den eigentlichen Entwurf von Algorithmen für GPGPU (*General-Purpose Computing on Graphics Processing Units*). Dabei wird insbesondere auf die Eigenschaften von CUDA eingegangen. Um einen Praxisbezug herzustellen wird dann letztendlich in *Kapitel 6* ein Ansatz einer Fehlersimulation von Schaltungen auf GPUs (*Graphics Processing Units*) bewertet, sowieso eine eigene Implementierung eines Phase-Shifter-Generators für *Built-In-Self-Test*-Anwendungen mit CUDA vorgestellt. Zum Abschluss der Arbeit wird in *Kapitel 7* ein zusammenfassendes Fazit gegeben.

1 Einleitung

Nachdem in den letzten Jahren der Schritt von Single-Core-Prozessoren zu Multi-Core-Prozessoren mit üblicherweise zwei bis vier Kernen vollzogen worden ist, folgt nun der nächste logische Schritt hin zu massiv parallelen Many-Core-Architekturen, im Speziellen GPUs, mit hunderten oder sogar tausenden Prozessoren beziehungsweise Kernen.

Im Zuge der zunehmenden Parallelisierung besteht steigender Bedarf an geeigneten Algorithmen. Diese müssen hinsichtlich Datenparallelität, Taskparallelität, Multithreading, Thread-Granularität sowie Speichermanagement angepasst und optimiert werden. In manchen Fällen reicht eine Anpassung existierender Algorithmen aus, oft ist jedoch ein konzeptioneller Neuentwurf der Algorithmen erforderlich. Dabei muss auf die technischen Besonderheiten und Merkmale der GPUs geachtet werden. Moderne GPUs sind besonders bei immer wiederkehrenden, gleichen Operationen auf unterschiedlichen Daten (*Single Instruction Multiple Data*, SIMD) leistungsstark, jedoch gestaltet sich der Datenaustausch zwischen Hauptspeicher und Grafikkarte, beziehungsweise zwischen Grafikkarte und CPU (Central Processing Unit), performanzhemmend. Auch der Datenaustausch zwischen den Recheneinheiten einer GPU ist nicht vorgesehen und demnach nicht effizient realisierbar. Besondere Problembereiche sind unter anderem sehr kleine, prozessornahe Speicher, höchste Leistungsfähigkeit nur bei Gleitkommaoperationen einfacher Genauigkeit und langsame Kommunikation zwischen Hardware-Komponenten.

Das Parallelisierungspotential ist dabei sehr stark von der jeweiligen Aufgabe abhängig – manche Aufgaben lassen sich sehr effizient parallelisieren, andere sind von inhärent serieller Natur. Im Bereich der Bildverarbeitung liegen Stärken der GPUs beispielsweise bei Pixeloperationen, die die Ergebnisse benachbarter Bildbereiche nicht akkumulieren müssen, somit Datenunabhängigkeit implizieren. Besonders im Bereich der *Electronic Design Automation* (EDA), und der damit verbundenen Simulation im Rahmen der Designvalidierung, gibt es zahlreiche Algorithmen mit großem Parallelisierungspotential.

Diese Fachstudie diskutiert die Anforderungen und Ansätze, die auf einen Wechsel von serieller Problemlösung hin zu hochgradig parallel arbeitenden Algorithmen, zutreffen.

2 Rechner-Architekturen

Bevor auf verschiedene Rechnerarchitekturen, sowie die Vorstellung eines Beispielproblems, eingegangen wird, folgt eine Taxonomie von Rechnerarchitekturen nach der *Flynnschen Klassifikation* [Fly72].

2.1 Flynnsche Klassifikation

Diese Klassifikation separiert Rechnerarchitekturen in vier Kategorien, wie in *Tabelle 2.1* zu sehen.

	single instruction	**multiple instruction**
single data	SISD	MISD
multiple data	SIMD	MIMD

Tabelle 2.1: Flynnsche Klassifikation

Die hier genannten Klassen haben folgende Bedeutung:

SISD steht für *Single Instruction Single Data* und beschreibt das klassische Rechenvorgehen auf skalaren Prozessoren. Ein Prozessor bearbeitet ein Datum sequentiell. Vor der Massentauglichkeit von Multi-Core-Prozessoren war dies die Standardvorgehensweise.

MISD steht für *Multiple Instruction Single Data*. Hinter dieser Klasse steht die Idee mehrere Instruktionen auf das gleiche Datum anzuwenden. Es gibt wenige Anwendungsfälle, die wirkliche MISD-Verarbeitung voraussetzen. Ein Beispiel sind fehlertolerante Systeme, die neben der eigentlichen Verarbeitung des Datums Prüfungen vornehmen können. Somit sind Verarbeitung und Prüfung gesonderte „Instruktionen“.

SIMD steht für *Single Instruction Multiple Data*. Diese Klasse ist in Architekturen zu finden, die das Prinzip der Datenparallelität umsetzen. Eine Instruktion wird auf mehrere Daten angewendet. SIMD impliziert nicht zwingend, dass die unterschiedlichen Daten zur gleichen Zeit bearbeitet werden. Ob die Daten gleichzeitig oder hintereinander bearbeitet werden, ist von der jeweiligen Architekturausprägung abhängig. SIMD-Prinzipien werden häufig in Multimedia-Anwendungen, wie der Bild- oder Videoverarbeitung, eingesetzt.

MIMD steht für *Multiple Instruction Multiple Data*. In Analogie zu den vorherigen drei Klassen werden bei MIMD mehrere Instruktionen auf mehrere Daten angewendet. Dieses Vorgehen ist nur für sehr spezielle Problemstellungen umsetzbar. MIMD findet sich

Listing 2.1 Serielle Verarbeitung

```
for i in (1, k) do
  C[i] = A[i] + B[i]
od
```

ausschließlich in parallelen Architekturen. Ein Problem von MIMD ist der Nichtdeterminismus, der durch das parallele Ausführen unterschiedlich aufwändiger Instruktionen entsteht. Das *Message Passing Interface* (MPI) ist eine Schnittstelle, die darauf abzielt, Zugriff auf parallele Hardwarearchitekturen zu ermöglichen. In MPI-Programmen führt jeder Prozess eigene Instruktionen, jedoch das selbe Programm, auf eigenen Daten aus. Darum ist MPI ein anschauliches Beispiel für die Nutzung von MIMD-Architekturen.

2.2 Beispielproblem

Um die Unterschiede zwischen verschiedenen Prozessorarchitekturen aufzuzeigen, betrachten wir das Beispiel einer Vektoraddition. Es wird dabei auf skalare Prozessoren (mit und ohne Pipelining), Vektorprozessoren sowie Stream-Prozessoren eingegangen.

Im Beispiel der Vektoraddition sollen zwei Vektoren A und B der Länge k zu einem Vektor C der Länge k addiert werden. Die Elemente der Vektoren bestehen jeweils aus einem n-bit-Wort, wobei n so gewählt sei, dass diese Worte in einem Prozessortakt addiert werden können. Für die Verarbeitungsbeispiele wird ein Pseudocode verwendet.

2.3 Skalare Prozessoren

Um mit einem skalaren Prozessor zwei Vektoren zu addieren, implementiert der Programmierer üblicherweise eine Schleife über die Anzahl der Vektorelemente und addiert diese komponentenweise (siehe *Listing 2.1*).

Dabei entspricht jede Addition einer Instruktion. Die Vektoraddition umfasst also k gesonderte Additions-Instruktionen. Der Prozessor muss für jede Addition eine eigene Instruktion aus dem Instruktionsspeicher holen, diese dekodieren und behandeln (siehe *Listing 2.2*).

Dieses Vorgehen entspricht dem SISD-Prinzip. Traditionelle Rechner der Von-Neumann- und Harvard-Architektur basieren auf diesem Prinzip und führen jeweils eine Instruktion auf einem skalaren Datum aus.

Listing 2.2 Serielle Verarbeitung (Instruktionsebene)

```
for i in (1, k) do
  fetch instruction i
  decode instruction i
  read A[i]
  read B[i]
  add A[i], B[i], C[i]
  write back C[i]
od
```

2.3.1 Pipelining

Um die Verarbeitung zu beschleunigen können Pipelines für die Instruktionen eingesetzt werden (siehe *Abbildung 2.1*). Dabei werden die einzelnen Operationen in mehrere Teilaufgaben zerlegt, die von unterschiedlichen Baugruppen des Prozessors unabhängig bearbeitet werden können. Dadurch ist es dem Prozessor möglich an mehreren Operationen gleichzeitig zu arbeiten, sofern sich dadurch keine Belegungskonflikte für die Baugruppen ergeben.

Der Instruktionsdurchsatz wird durch Pipelining signifikant erhöht, sofern mehr als nur eine Instruktion zu bearbeiten ist. Die Verarbeitungsgeschwindigkeit einer einzelnen Instruktion wird jedoch nicht erhöht. Auch mit Pipelining muss für jedes Datum eine eigene Instruktion verarbeitet werden.

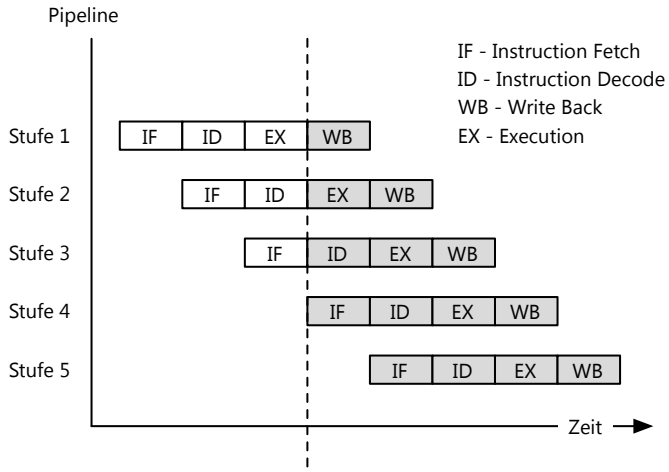

Abbildung 2.1: Beispielhaftes Pipelining eines skalaren Prozessors

Jedoch bringt auch Pipelining Probleme mit sich, wie beispielsweise *Hazards*. Diese lassen sich in drei Kategorien gliedern:

Struktur Aufgrund von Hardwarebeschränkungen ist es nicht immer möglich bestimmte Schritte in der Pipeline gleichzeitig auszuführen. Ist beispielsweise die Hardware so

konzipiert, dass IF und ID auf derselben Hardwarekomponente ausgeführt werden, muss die Pipeline möglicherweise einen Zyklus warten.

Daten Sind Befehle in der Pipeline voneinander abhängig, sprich benötigt ein späterer Befehl in der Pipeline das Ergebnis eines früheren Befehls, so muss die Pipeline so lange warten, bis der benötigte Befehl komplett abgearbeitet ist.

Kontrollfluss Beinhaltet ein Befehl eine bedingte Sprunganweisung (beispielsweise eine Verzweigung), so muss mit dem nächsten IF gewartet werden, bis der bedingte Sprung ausgewertet ist.

Die beschriebenen Hazards können zu *Stalls* (Wartezyklen) in der Pipeline führen. Das Umordnen von Befehlen kann diesen Stalling-Effekten entgegenwirken.

2.3.2 Superskalare Prozessoren

Eine spezielle Ausprägung von skalaren Prozessoren sind die sogenannten superskalaren Prozessoren. Diese besitzen mehrere Recheneinheiten und können somit mehrere Befehle (Anzahl prinzipiell nicht beschränkt) zur gleichen Zeit ausführen. Die auf der *Intel Microarchitecture (Nehalem)* basierenden Prozessoren wie beispielsweise der Intel Core i7-800 können laut [Cor09] bis zu vier Instruktionen in einem Taktzyklus ausführen. Superskalare Prozessoren implementieren dadurch eine Parallelität in der Befehlsdimension. Kombiniert mit dem bereits beschriebenen Pipelining ist dies das Optimum der Verarbeitung, welches mit skalaren Prozessoren erreicht werden kann.

2.3.3 Ausblick

Skalare und auch superskalare Prozessoren sind an ihre Grenzen gestoßen. Das immer weitere Erhöhen der Taktfrequenz zur schnelleren Verarbeitung ist durch physikalische und architektonische Beschränkungen begrenzt. Optimierungsideen wie Sprungvorhersagen, Wertspekulationen, übergroße Befehlsfenster und immer mehr Verarbeitungseinheiten für Befehlsparallelität führen zu immer komplexeren Architekturausprägungen. Der dadurch entstehende Aufwand wirkt hemmend auf die Steigerung der Taktrate, die damit eigentlich erreicht werden soll.

Dies hat Fred. J. Pollack 1999 in seiner als *Pollack's Rule* bekannten Aussage ([Pol99]) prägnant formuliert:

> *„Performance increase is roughly proportional to square root of increase in complexity."*

Mit anderen Worten erfordert es eine Vervierfachung der Komplexität um eine Performanzverdoppelung zu erreichen. Dieser Sachverhalt führt automatisch zu parallelen Architekturen, wobei parallel hier nicht auf eine bestimmte Art der Paralleliät (datenparallel, prozessparallel) beschränkt ist.

Listing 2.3 Vektor-Verarbeitung

```
C = vec_sum(A, B)
```

2.4 Vektorprozessoren

Im Gegensatz zu skalaren Prozessoren arbeiten Vektorprozessoren auf Datenvektoren von k Elementen, welche jeweils ein n-bit-Wort darstellen. Der Begriff Vektorprozessoren trifft keine Aussage darüber, wie die Vektoren intern verarbeitet werden. Der Begriff sagt lediglich aus, dass Prozessoren von diesem Typ einen um gesonderte Instruktionen für Vektoroperationen erweiterten Befehlssatz bereitstellen. Ob diese Instruktionen intern parallel oder seriell abgearbeitet werden ist dadurch noch nicht festgelegt. Unabhängig davon unterliegen beide Arbeitsstrategien dem SIMD-Prinzip. Auf die k Elemente der Eingabevektoren (multiple data) wird paarweise dieselbe Instruktion (single instruction) angewendet (siehe *Abbildung 2.2*).

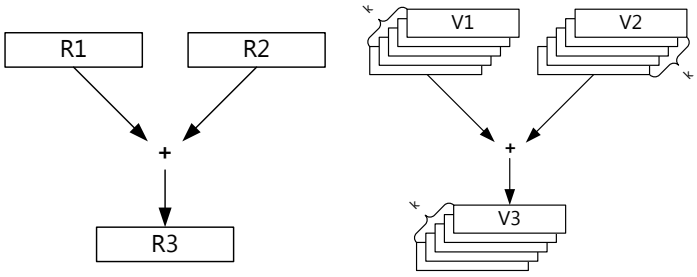

Abbildung 2.2: Unterschied skalarer Prozessor (links) und Vektorprozessor (rechts) bei der Vektoraddition

Zu beachten ist, dass der Ausgabevektor einer Vektoroperation immer von gleicher Länge und gleichen Typs wie die Eingabevektoren ist. Weiterhin sind Vektorprozessoren nur dann effizient einsetzbar, wenn die zu bearbeitende Aufgabe hauptsächlich vektorielle Daten beinhaltet. Da der Befehlssatz eines Vektorprozessors komplexer als der eines skalaren Prozessors ist, entsteht auch bei der Dekodierung der Instruktionen ein gewisser Overhead. Deshalb kann die Verarbeitung von skalaren Daten unnötigen Aufwand mit sich ziehen und somit insgesamt langsamer ablaufen als auf einem skalaren Prozessor.

Ein allgemeines Problem der Vektorprozessoren besteht darin, dass die zu lösenden Probleme zunächst vom Programmierer manuell auf die Struktur eines Vektorprozessors abzubilden sind, womit Programme komplexer werden können.

Listing 2.3 zeigt, wie ein Programmierer üblicherweise eine solche Vektoraddition auf einem Vektorprozessor implementieren würde. Dabei fällt auf, dass hierfür keine Schleife notwendig ist, weil bekannt ist, dass die Elemente der Vektoren in einem festen Abstand konsekutiv im Speicher abgelegt und damit leicht adressierbar sind.

Listing 2.4 beschreibt, wie der Programmcode aus *Listing 2.3* für einen Vektorprozessor umgesetzt wird. Größter Vorteil ist, dass nur noch eine Instruktion gelesen, dekodiert und

Listing 2.4 Vektor-Verarbeitung

```
fetch instruction
decode instruction
vec_read A
vec_read B
vec_add A, B, C
write back C
```

verarbeitet werden muss. Wie oben bereits angesprochen ist jedoch nicht klar, wie die Operation `vec_add` intern verarbeitet wird (seriell oder parallel).

2.4.1 Arbeitsstrategien

Wie bereits erwähnt gibt es zwei Arbeitsstrategien um die Daten zu verarbeiten: einen seriellen und einen parallelen Ansatz.

Seriell

Der serielle Ansatz basiert darauf, dass die Daten in einer Pipeline verarbeitet werden. Die Instruktion wird von einer Instruktionseinheit bereitgestellt und die Elemente der Vektoren werden sequentiell an die Pipeline gesendet. Das bedeutet, dass mit der Verarbeitung neuer Operanden schon begonnen werden kann, während sich vorherige Operanden noch in der Pipeline (jedoch in einer anderen Stufe) befinden. Es befinden sich zu jedem Zeitpunkt maximal soviele Operanden in der Pipeline, wie diese Stufen hat. Somit wird eine Operation quasi parallel, jedoch deterministisch (FIFO), auf einem skalaren Prozessor ausgeführt.

Nach [Haa93] wird der Begriff Vektorrechner nur für diese Arbeitsstrategie verwendet.

Parallel

Eine andere Möglichkeit ist der parallele Ansatz, bei dem auch genau eine Instruktionseinheit existiert, jedoch mehrere Verarbeitungseinheiten für die Daten vorhanden sind. Die Operationen auf den einzelnen Operandenpaaren der Eingabevektoren werden also parallel auf den Verarbeitungseinheiten ausgeführt. Dies entspricht einer echten Parallelität. Man spricht hierbei laut [Haa93] von einem Feldprozessor. Der parallele Ansatz kann mit dem seriellen Ansatz kombiniert werden: Beispielsweise können in den parallelen Verarbeitungseinheiten Pipelines eingesetzt werden.

2.4.2 Ausblick

Vektorprozessoren finden nur in speziellen Problemstellungen Anwendung. Probleme müssen explizit auf Vektorstrukturen abgebildet werden, um Vorteile aus dieser Art der Verarbeitung zu ziehen. Viele Probleme bestehen oft aus einem großen, rein seriellen Problemanteil, der von Vektorprozessoren nicht performant behandelt werden kann. Auch haben Vektorprozessoren eine längere Anlaufzeit, bis überhaupt mit der Berechnung begonnen werden kann. Dort spielt die Größe der Vektoren eine Rolle: Kann eine Vektormaschine nur mit sehr kleinen Vektoren arbeiten, so wiegt der Overhead der Vektorverarbeitung und die Anlaufzeit schwerer.

Vektorrechner haben zum Zeitpunkt dieser Fachstudie keine wirkliche Bedeutung mehr.

2.5 Stream-Prozessoren

Auf den bisher genannten Prozessorarchitekturen ist es sehr aufwändig, die Instruktionen und Daten zu den ALUs (*Arithmetic Logic Unit*) zu übertragen. Stream-Prozessoren versuchen diesem Problem mit dem Grundsatz der Datenlokalität entgegenzuwirken. Dabei wird der Speicher so organisiert, dass die benötigten Daten immer in der Nähe der ALUs liegen. Somit entfällt viel Kommunikationsaufwand.

Diese Organisation umfasst drei Ebenen:

1. Die LRFs (*Local Register Files*) befinden sich in unmittelbarer Nähe der ALUs und speichern die von diesen benötigten Operanden.
2. Die SRFs (*Stream Register Files*) können größere Datenmengen (Streams) speichern und dienen dem effizienten Datentransfer zwischen den LRFs.
3. Der große globale Speicher wird nur dann verwendet, wenn es die Situation unbedingt erfordert. Da sich dieser Speicher nicht auf dem Prozessorchip befindet, ist die Zugriffszeit vergleichsweise hoch.

Die durch diese Speicherorganisation erzeugte Datenlokalität ermöglicht es, eine große Anzahl an ALUs nahe ihres maximalen Auslastungsgrades zu betreiben.

Im Gegensatz dazu werden die Speicher bei den zuvor genannten Rechnerarchitekturen von den Recheneinheiten räumlich getrennt. Dies erlaubt größere Speicher, bringt aber höhere Latenzzeiten mit sich.

Entwickler von Programmen für Stream-Prozessor-Architekturen müssen ihre Programme also an beschränktere Speicherkapazitäten anpassen. Insbesondere ist es bei Stream-Architekturen auch möglich, Caches und Speicher selbst zu organisieren. Somit hat ein Programmierer die Möglichkeit zu beeinflussen, wo Daten abgelegt werden und welche Daten gecached werden. Die Erfahrung zeigt, dass die Performanz von Programmen für Stream-Prozessor-Architekturen signifikant von deren Speicherzugriffsverhalten abhängt.

Um hochperformante Programme zu entwickeln, müssen die Programmierer also die Speicherorganisation der Zielhardware, sowie Latenzzeiten, Bandbreiten und Durchsatz kennen und ihre Algorithmen daran anpassen. Da in Stream-Prozessor-Architekturen zudem meist mehrere spezialisierte Speicher vorhanden sind, müssen Entwickler zudem wissen, welche dieser Speicher für welche Speicherzugriffsverhalten und Datenmengen ausgelegt sind. Unkenntnis über diese Speichercharakteristiken kann zu, hinsichtlich der Speicherlatenzzeiten, ungünstigen Programmen führen, welche weit hinter der optimalen Performanz zurückliegen.

Allerdings lässt sich dieser streambasierte Ansatz nicht auf alle Problemstellungen anwenden. Besonders gut lässt er sich dann nutzen, wenn sehr viele Operationen gleichzeitig ausgeführt werden sollen und nur minimale globale Kommunikation und Speicherung erforderlich ist. Ein Beispiel ist die Beleuchtungsberechnung in einem Pixelshader, welcher für jeden Pixel dessen Helligkeitswert ausrechnet.

2.5.1 Streams

Wie der Name des Stream-Prozessors schon andeutet, ist eines seiner Hauptkonzepte die Daten in sogenannte *Streams* zu organisieren. Ein Stream besteht aus einer Menge von Elementen des gleichen Typs. Diese können sehr simpel sein (beispielsweise ein Integer), oder von komplexerem Typ (beispielsweise Quaternionen). Verschiedene Streams, die im Prozessor verarbeitet werden, müssen nicht notwendigerweise gleich lang sein.

Die konsekutive Organisation der Daten in einem Stream stellt die Datenlokalität sicher. Für diese Datenorganisation muss dabei in vielen Fällen kein gesonderter Aufwand betrieben werden, da die Daten oft schon in Form eines Streams vorliegen (beispielsweise die Bilder einer Videokamera).

Vorteil ist auch, dass es mittels der SRFs möglich ist, einen Stream direkt von einer ALU in die nächste zu transportieren. Dies minimiert die I/O-Operationen für das Zwischenspeichern der Daten.

2.5.2 Kernels

Die eben angesprochenen Streams werden von sogenannten *Kernels* verarbeitet. Dabei kann ein Kernel mehrere Input-Streams, sowie mehrere Output-Streams besitzen (siehe *Abbildung 2.3*). Die Input- und Output-Streams müssen nicht vom gleichen Typ sein. Ein Kernel ist ein üblicherweise kleines Programm, welches mehrere simple Operationen kapselt, um eine spezielle Problemstellung zu erfüllen. Dabei kann ein Kernel nur sehr wenige Operationen, oder aber auch mehrere tausend Operationen umfassen.

Ein Kernel wird auf jedes Element des Streams einzeln angewendet. Kernels können dabei nur auf ihre Input- und Output-Streams, nicht aber auf andere externe Daten, zugreifen.

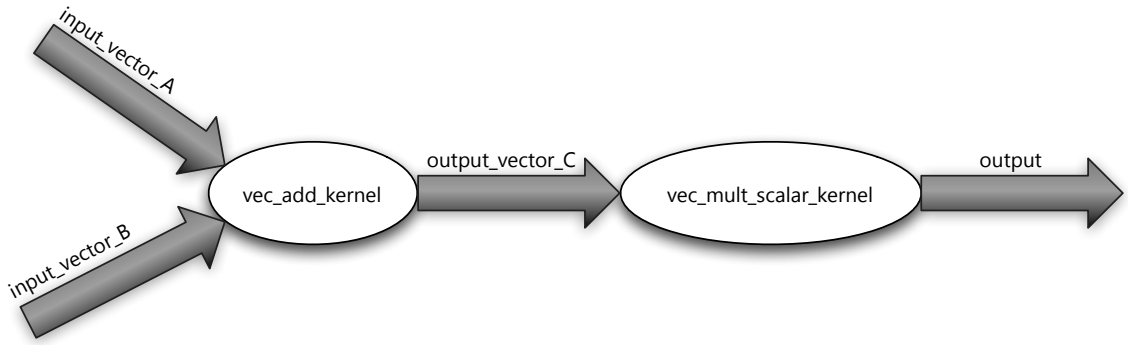

Abbildung 2.3: Verarbeitung von Streams mit Kernels

2.5.3 Parallelität

Das bisher vorgestellte Modell der Stream-Prozessoren beschreibt keine zwangsweise parallele Verarbeitung, jedoch eignet sich das Stream-Modell sehr gut dafür. Möchte man Parallelität in Stream-Prozessor-Architekturen realisieren, so kann dies prinzipiell in drei Dimensionen, die im Folgenden erläutert werden sollen, geschehen.

Instruktionsdimension Das Konzept der Kernels wurde bereits vorgestellt. Ein Kernel selbst gruppiert mehrere Operationen zu einem Programm. Diese Operationen innerhalb eines Kernels können parallel ausgeführt werden, sofern sie nicht voneinander abhängen. Dies entspricht einer Parallelität in der Instruktionsdimension.

Datendimension Da ein Stream aus mehreren Elementen des selben Typs besteht, auf die jeweils der gleiche Kernel angewendet wird, besteht auch hier Parallelisierungspotential. Mehrere Elemente eines Streams können zur gleichen Zeit, von verschiedenen Recheneinheiten, verarbeitet werden. Dies realisiert die Datenparallelität.

Taskdimension Diese Dimension der Parallelität basiert darauf, verschiedene, voneinander unabhängige, Aktionen innerhalb der Stream-Verarbeitung parallel auszuführen. Das Ausführen eines Kernels und das Weiterleiten eines Streams an einen anderen Kernel kann gleichzeitig geschehen, sofern diese beiden Aktionen nicht voneinander abhängig sind.

Diese drei Dimensionen der Parallelität schließen sich keinesfalls gegenseitig aus, sondern können auch miteinander kombiniert werden. Mehrdimensionale Parallelität erlaubt die flexible und hochperformante Parallelisierung des Lösens verschiedenartiger Probleme, während im Falle einer eindimensionalen Parallelität Problemlösungen für die gewählte Dimension geeignet sein und damit gegebenenfalls angepasst werden müssen. Die Dimensionen der Parallelität bilden die Grundlage für die Anwendung von Stream-Modellen in Grafikkarten.

Listing 2.5 Stream-Programm

```
stream <Vector> input_vector_A(k) # k is vector length
stream <Vector> input_vector_B(k)
stream <Vector> output_vector_C(k)

input_vector_A = read_vector()
input_vector_B = read_vector()

execute vec_add_kernel(in input_vector_A, in input_vector_B, out
    output_vector_C)
```

Listing 2.6 Kernel-Programm

```
vec_add_kernel(in input_vector_A, in input_vector_B, out output_vector_C)
{
  for(int i = 0; i < k; i++) do
    output_vector_C.push(input_vector_A.pop() + input_vector_B.pop())
  od
}
```

2.5.4 Programmbeispiel

Das beispielhafte Problem der Vektoraddition wird mittels eines Stream-Prozessors in *Listing 2.5* und *Listing 2.6* schematisch realisiert.

Das in *Listing 2.5* gezeigte Stream-Programm deklariert zuerst die verwendeten Input- und Output-Streams. Die Input-Streams werden mittels der Funktion `read_vector()` eingelesen. Auf Parameter zur Adressierung wird aus Simplizitätsgründen verzichtet. Zuletzt wird der Additionskernel aufgerufen.

Das in *Listing 2.6* gezeigte Kernel-Programm iteriert nun über die Anzahl der Vektorelemente. Es wird von jedem Input-Stream jeweils das nächste Element gelesen, diese werden addiert und das Ergebnis auf den Output-Stream geschrieben.

2.5.5 Klassifikation

Stream-Prozessor-Architekturen lassen sich nicht, wie skalare oder Vektorprozessoren, einfach in die Flynnsche Klassifikation einordnen. Dies ist hauptsächlich der Tatsache geschuldet, dass die Implementierungen des Stream-Prozessor-Konzepts sehr unterschiedlich sind. Das Konzept basiert nur auf der Existenz von Streams und Kernels, sowie der Datenlokalität, sagt aber nichts über eine konkrete Implementierung aus.

Insbesondere ist nicht festgelegt, über wieviele Instruktionseinheiten ein Stream-Prozessor verfügt. Ist nur eine Instruktionseinheit vorhanden, so führen alle Recheneinheiten des Stream-Prozessors stets dieselbe Instruktion aus, es handelt sich also, nach der Flynnschen

Klassifikation, um SIMD. Sind mehrere Instruktionseinheiten vorhanden, so können unterschiedliche Recheneinheiten zeitgleich an unterschiedlichen Instruktionen arbeiten, damit handelt es sich um MIMD. Jede Recheneinheit führt dabei den selben Kernel aus, jedoch können verschiedene Recheneinheiten zu einem bestimmten Zeitpunkt an unterschiedlichen Instruktionen dieses Kernels arbeiten. Eine solche *Divergenz* kann beispielsweise durch datenabhängige Sprünge entstehen. Ist dieser Sachverhalt gegeben, so spricht man von SPMD (*Single Program Multiple Data*). Bei den hier verwendeten Terminologien ist zu beachten, dass MIMD und SPMD nicht gleich einzuordnen sind. MIMD ist eine Klassifikation, die von der Hardwareseite motiviert ist, wohingegen SPMD softwareseitig einzuordnen ist. Trotzdem lassen sich diese beiden Begriff im Bezug auf Stream-Prozessoren und deren Arbeitsweise vereinen.

Aus den angeführten Gründen ist keine eindeutige Einordnung in die Flynnsche Klassifikation möglich. Dies ist auch einer der Kritikpunkte an der Flynnschen Klassifikation.

3 Hardware

Nachdem im vorherigen Kapitel auf verschiedene Architekturen eingegangen wurde, sollen nun konkrete Hardwareplattformen diskutiert werden. Es wird auf zum Zeitpunkt dieser Fachstudie aktuelle Grafikhardware eingegangen, welche die Ansätze aus *Kapitel 2* zum Teil unverändert, zum Teil angepasst, realisiert.

3.1 Allgemeine Rahmenbedingungen

Die zum Zeitpunkt dieser Fachstudie aktuellen GPUs können Gleitkommaoperationen doppelter Genauigkeit entsprechend dem „IEEE 754"-Standard (Binary Floating-Point Arithmetic for Microprocessor Systems [ANSI/IEEE Std 754-1985]) nur sehr ineffizient ausführen. Der Grund hierfür ist, dass aktuelle GPUs nur sehr wenige oder gar keine Recheneinheiten besitzen, die Gleitkommaoperationen doppelter Genauigkeit ausführen können. In letztgenanntem Fall können zwei Einheiten für Gleitkommaoperationen einfacher Genauigkeit gekoppelt werden, um so eine Operation doppelter Genauigkeit auszuführen. In beiden Fällen können jedoch pro Sekunde signifikant weniger Operationen doppelter Genauigkeit ausgeführt werden als Operationen einfacher Genauigkeit.

Ein generelles Problem bei GPGPU besteht darin, dass die Daten, auf denen gerechnet werden soll, vom Hauptspeicher der CPU in den Speicher der GPU transferiert werden müssen. Ebenso müssen nach dem Beenden der Berechnungen die Ergebnisse vom GPU-Speicher wieder in den Hauptspeicher der CPU zurückgeschrieben werden. Es eignen sich also prinzipiell nur solche Algorithmen für GPGPU, bei denen das Verhältnis von Datenmenge zu Anzahl darauf ausgeführter Instruktionen klein ist.

$$\frac{\text{Datenmenge}}{\text{Instruktionen}} \ll 1$$

Ist diese Voraussetzung nicht gegeben, muss die Grafikkarte mit der Bearbeitung solange warten, bis die CPU die zur Berechnung notwendigen Daten über den Bus geliefert hat. Auch wenn die momentan weit verbreiteten PCI-Express-Schnittstellen sehr hohe Datentransferraten (bis zu 32 GB/s mit PCIe 3.0 und 32 Lanes) unterstützen, kann der Datentransfer bei nach obigem Kriterium ungünstigen Algorithmen einen Flaschenhals darstellen.

3.2 NVIDIA

Zunächst soll der Aufbau von CUDA-fähiger Hardware skizziert werden, der für die späteren Ausführungen in dieser Fachstudie grundlegend ist.

3.2.1 Hardwareaufbau

Eine GPU der Firma NVIDIA besteht je nach Baureihe aus einer bestimmten Anzahl von sogenannten *Multiprozessoren* (siehe *Abbildung 3.1*). Jeder dieser Multiprozessoren beherbergt acht Stream-Prozessoren und besitzt einen 16KB großen *Shared Memory* sowie 8192 Register, die dynamisch den Stream-Prozessoren zugeteilt werden.

Auf den *Shared Memory* haben alle Stream-Prozessoren Lese- und Schreibzugriff. Er dient als Zwischenspeicher für temporäre, gemeinsame Daten der Stream-Prozessoren. Dieser Speicher kann also auch für die Kommunikation zwischen den Stream-Prozessoren eines Multiprozessors genutzt werden. Die Stream-Prozessoren anderer Multiprozessoren haben keinen Zugriff darauf.

Neben dem *Shared Memory* existieren mit *Global Memory*, *Texture Memory* und *Constant Memory* noch drei andere Speicher, wobei die zwei letztgenannten lediglich dedizierte Bereiche des *Global Memory* sind.

Auf den *Global Memory* haben alle Prozessoren Lese- und Schreibzugriff. Er ist üblicherweise sehr groß, weist aber mangels eines Caches eine sehr hohe Speicherlatenzzeit auf.

Auf den *Constant Memory* haben alle Stream-Prozessoren eines Multiprozessors lesenden Zugriff. Beschrieben kann der *Constant Memory* nur von der CPU (Hostsystem) werden. Der Zugriff auf den *Constant Memory* ist gecached (8KB Cache), die optimale Performanz wird jedoch nur dann erreicht, wenn alle Stream-Prozessoren zeitgleich auf dasselbe Datum im *Constant Memory* zugreifen.

Die Größe des *Texture Memory* ist prinzipiell variabel, jedoch durch die Größe des verfügbaren *Global Memory* begrenzt. Er ist darauf ausgelegt ein- und zweidimensionale Datenstrukturen effizient verarbeiten zu können und wird gecached. Der Cache des *Texture Memory* variiert zwischen 6KB und 8KB pro Multiprozessor, abhängig von der konkreten Grafikhardware. Die Multi- beziehungsweise Stream-Prozessoren haben auf den *Texture Memory* nur lesenden Zugriff. Beschrieben kann dieser ebenfalls nur vom Hostsystem werden.

3.2.2 Aktuell verfügbare Hardware

Im Folgenden sollen zwei aktuelle Hardwareplattformen der Firma NVIDIA vorgestellt werden.

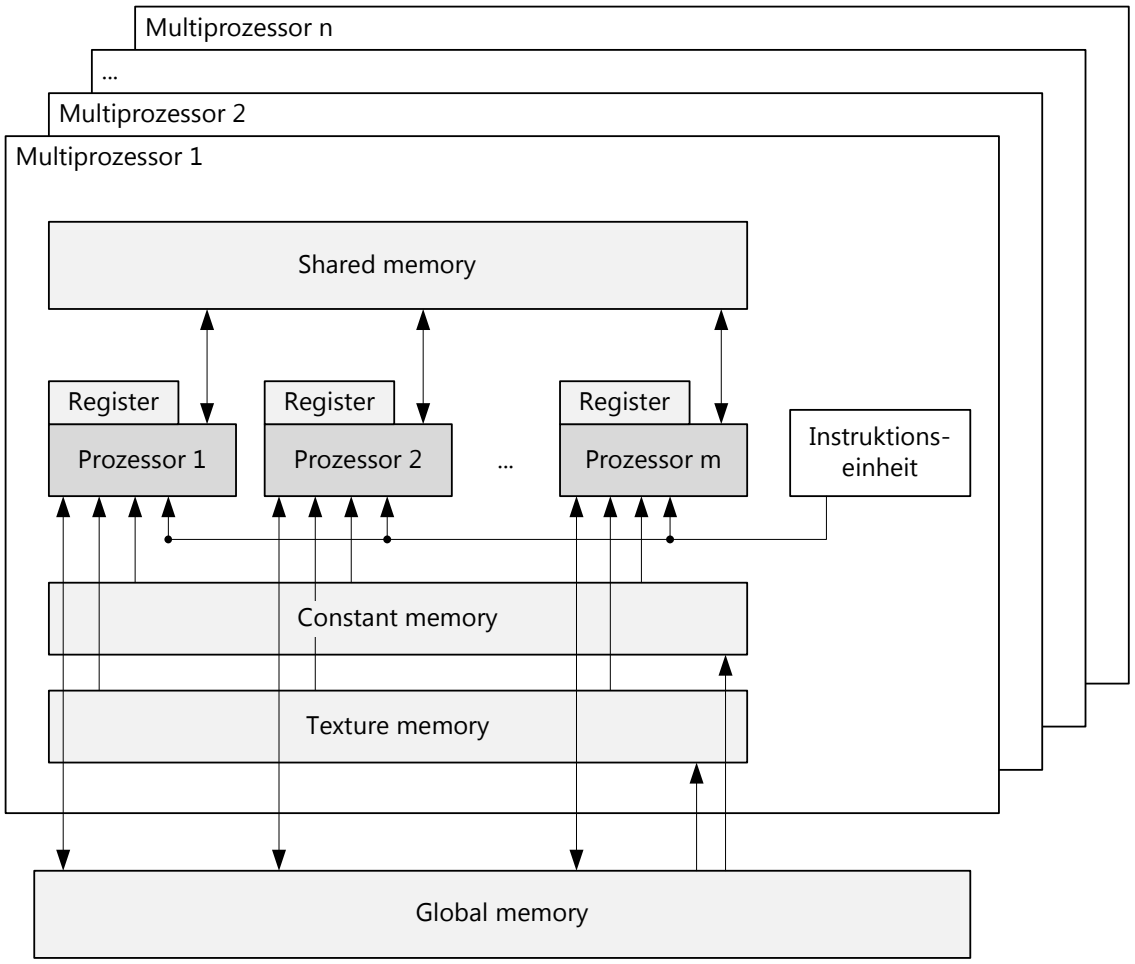

Abbildung 3.1: Aufbau eines CUDA-Geräts

GeForce GTX 295 ist die zur Zeit dieser Studie aktuellste Desktop-Grafikkarte von NVIDIA. Sie besitzt zwei GPUs mit jeweils 30 Multiprozessoren, also insgesamt 480 Stream-Prozessoren. Die 896 MB Speicher pro GPU sind über einen 448-bit breiten Bus mit einer maximalen Bandbreite von 223.8 GB/s angebunden. Es ergibt sich eine maximale Rechenleistung von 1,79 Teraflops.

Tesla S1070 ist ein High-Performance-Computing-System von NVIDIA. Im Gegensatz zur oben genannten GeForce GTX 295 kann das Tesla-System keine Grafik auf ein Display ausgeben, sondern ist lediglich ein Prozessorsystem für Simulationen und Bildgenerierungen in wissenschaftlichen Systemen. Das System besteht aus vier GPUs zu je 240 Multiprozessoren. Die 4GB Speicher pro GPU sind über einen 512-bit breiten Bus mit einer maximalen Bandbreite von 410 GB/s angebunden. Es ergibt sich eine maximale Rechenleistung von 4,32 Teraflops.

3.3 AMD/ATI

Im Folgenden sollen, basierend auf [Advo9], die Grundzüge des *ATI Stream Computing* sowie die Struktur der zugrunde liegenden Hardware kurz erläutert werden.

3.3.1 Hardwareaufbau

Der Hardwareaufbau eines Stream-Prozessors von AMD/ATI ist in *Abbildung 3.2* dargestellt. Eine GPU wird hierbei allgemein als Stream-Prozessor bezeichnet. Dieser besteht aus einer gewissen Anzahl sogenannter *SIMD-Engines*. Wieviele SIMD-Engines ein Stream-Prozessor umfasst, hängt vom jeweiligen Gerät ab. Dies bedeutet, dass Skalierbarkeit über die Anzahl der SIMD-Engines auf einem Stream-Prozessor gegeben ist.

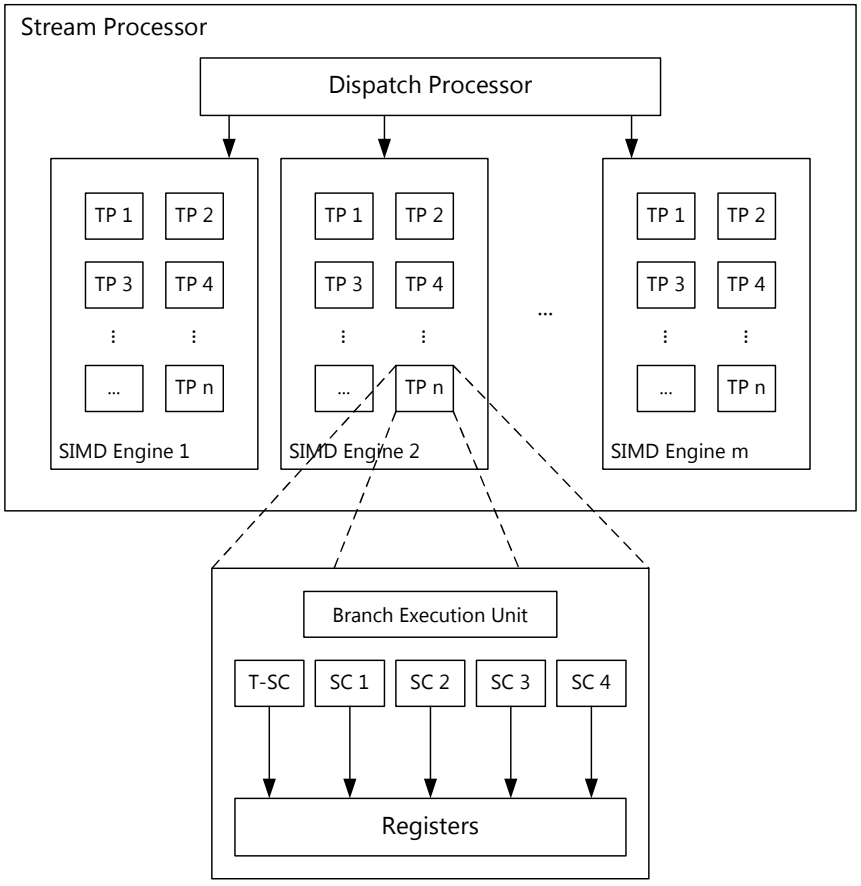

Abbildung 3.2: Aufbau eines AMD/ATI Stream-Prozessors

Eine SIMD-Engine besteht nun ihrerseits aus einer gewissen Anzahl von *Thread Processors*, wobei deren Anzahl wiederum vom konkreten Gerät abhängig ist. Ein Thread Processor kann

bis zu fünf skalare Operationen in VLIW-Instruktionen (*Very Long Instruction Word*) verarbeiten. Er enthält dafür vier *Streaming Cores*, die Gleitkommaoperationen einfacher Genauigkeit sowie Ganzzahloperationen ausführen können. Zudem existiert ein weiterer *Streaming Core*, welcher transzendente Operationen wie Sinus, Logarithmus und dergleichen ausführen kann. Gleitkommaoperationen doppelter Genauigkeit können ausgeführt werden, indem zwei *Streaming Cores* zusammengeschaltet werden. Schließlich existiert auf jedem *Thread Processor* eine *Branch Execution Unit*, die für die Verarbeitung von Verzweigungsanweisungen zuständig ist.

Anders als bei NVIDIA existieren auf AMD/ATI-Hardware lediglich der große *Global Memory* des Stream-Prozessors sowie die lokalen Register. Zugriffe auf den *Global Memory* werden hier über eine sogenannte *Data Fetch Unit* ausgeführt und Threads, die auf Ausführung einer Speicheroperation warten, werden während dieser Wartezeit deaktiviert, sodass andere ausführbereite Threads ausgeführt und damit die Speicherlatenz versteckt werden kann. Zudem existiert ein Mechanismus, der als *Shared Registers* bezeichnet wird und es Threads ermöglicht, auf die lokalen Daten von anderen Threads lesend zuzugreifen.

Zusätzlich zu den lokalen Registern und dem *Global Memory* des Stream-Prozessors steht der PCIe-Speicher zur Verfügung, welcher ein ausgezeichneter Bereich des CPU-Hauptspeichers ist. Auf diesen Speicher können sowohl das auf der CPU ausgeführte Host-Programm, wie auch das auf dem Stream-Prozessor ausgeführte Programm lesend und schreibend zugreifen. Deshalb ist dieser Speicher für Datenaustausch zwischen CPU und Stream-Prozessor beziehungsweise GPU geeignet. Auf den Hauptspeicher der CPU haben auf dem Stream-Prozessor ausgeführte Programme keinen Zugriff.

Weitere Details bezüglich der AMD/ATI-Hardware würden den Rahmen dieser Fachstudie sprengen. Diese konzentriert sich auf CUDA, auch weil dieses zum Zeitpunkt dieser Fachstudie eine bei Weitem größere Verbreitung genießt als der konkurrierende *ATI Stream Computing*-Ansatz. Da die Ansätze von NVIDIA aber in weiten Teilen denen von AMD/ATI ähneln, gilt der Großteil der in dieser Fachstudie vorgestellten Konzepte und Einschränkungen gleichermaßen für CUDA wie auch für *ATI Stream Computing*.

3.3.2 Aktuell verfügbare Hardware

Der AMD/ATI FireStream 9270 ist zum Zeitpunkt dieser Fachstudie die High-End-Lösung von AMD/ATI. Er umfasst eine GPU mit zehn *SIMD-Engines* zu je 16 *Thread Processors*. Da jeder Thread Processor aus 5 *Streaming Cores* aufgebaut ist, entspricht dies einer Anzahl von insgesamt 800 *Streaming Cores*.

Der AMD/ATI FireStream 9270 beherbergt 2GB GDDR5 SDRAM-Speicher, der über einen 256-bit breiten und mit 850MHz arbeitenden Bus angebunden ist. Die maximale Speicherbandbreite beträgt 108.8 GB/s. Er kann Gleitkommaoperationen einfacher und doppelter Genauigkeit hardwaremäßig ausführen, wobei die maximale Rechengeschwindigkeit 1.2 Teraflops für einfache und 240 Gigaflops für doppelte Genauigkeit beträgt.

4 Programmierumgebungen

In den letzten Jahren hat sich die Rechenleistung von Grafikprozessoren signifikant erhöht, bei zugleich sinkenden Kosten. Damit wurde das Konzept des GPGPU zunehmend für Industrie und Wissenschaft interessant. Allerdings gab es lange Zeit eine Reihe von Hürden, die einen weiten Einsatz von Grafikprozessoren für General-Purpose-Anwendungen hemmten. Zunächst war nur ein Teil der Grafikpipeline frei programmierbar, wofür der Programmierer ein tiefes Verständnis für die gesamte Grafikrenderingpipeline benötigte. Ebenso wurden Grafikprozessoren lange Zeit mittels spezieller Grafik-APIs programmiert, was eine Anpassung der Algorithmen an das verfügbare API erforderte. Dies machte es selbst erfahrenen Programmierern schwer, General-Purpose-Anwendungen auf Grafikprozessoren zu implementieren.

Um diese Hürden zu überwinden, sind einige Programmierumgebungen entstanden, die das Implementieren von Algorithmen auf Grafikprozessoren angenehmer und einfacher gestalten sollen. Die prominentesten Vertreter sind hierbei CUDA (*Compute Unified Device Architecture*) von NVIDIA, Brook+ für AMD/ATI-Grafikprozessoren, sowie OpenCL, das jedoch lediglich eine Schnittstellendefinition darstellt und beispielsweise in der Version 2.2 von CUDA implementiert ist.

Da heutige GPUs auf dem Konzept der Stream-Prozessoren basieren, bieten all diese Programmierumgebungen Konstrukte zur Organisation der Daten als Streams, Erstellung von Unterprogrammen, die als Kernel ausgeführt werden, sowie Mechanismen die festlegen, welche Programmteile auf der GPU und welche auf dem Hostsystem (CPU) ausgeführt werden.

In den folgenden Unterkapiteln sollen diese Programmierumgebungen nun vorgestellt werden. Das Hauptaugenmerk liegt dabei auf CUDA, weil dies zum Zeitpunkt dieser Fachstudie die weiteste Verbeitung genießt und die meisten wissenschaftlichen Projekte zu GPGPU darauf basieren.

4.1 CUDA

Das von NVIDIA entwickelte CUDA bietet eine Programmierumgebung für die auf den NVIDIA GeForce-Grafikkarten (ab Serie 8) und in den NVIDIA Tesla-Systemen verbauten Grafikprozessoren. Es besteht im Wesentlichen aus einer Erweiterung der Standard-C-Programmiersprache für die Entwicklung von parallelen Anwendungen auf GPUs, was erfahrenen Programmierern einen einfachen Einstieg ermöglicht. Weiterhin stellt NVIDIA hochoptimierte Bibliotheken für gängige numerische Problemstellungen bereit. Dazu gehören

beispielsweise eine Implementierung der beliebten BLAS-Bibliothek (*Basic Linear Algebra Subroutines*) sowie eines FFT-Verfahrens (*Fast Fourier Transform*). CUDA ist für die 32- und 64-bit Versionen aller gängigen Betriebssysteme (Microsoft Windows, Linux, Apple MacOS) verfügbar.

4.1.1 Ausführungsmodell

Das von NVIDIA gewählte Ausführungsmodell für CUDA-Applikationen legt den Grundstein für deren Skalierbarkeit. Die Parallelität wird durch die Verwendung von gleichzeitig ausgeführten Threads realisiert. Dazu muss die zu verarbeitende Datenmenge in entsprechend feingranulare Einheiten zerlegt werden, sodass jeder Thread auf einer solchen Dateneinheit unabhängig arbeiten kann. Skalierbarkeit ergibt sich daraus, dass auf leistungsfähigeren Grafikkarten mehr Stream-Prozessoren verbaut sind und diese somit mehr Threads gleichzeitig ausführen können.

In CUDA werden diese Threads nun in *Blocks* organisiert, wobei ein Block maximal 512 Threads umfassen kann. Ein solcher Block wird auf einem Multiprozessor ausgeführt. Sollen mehrere Blocks gleichzeitig auf einem Multiprozessor ausgeführt werden, so wird ein Zeitschlitzverfahren eingesetzt. Die Threads innerhalb eines Blocks werden zu *Warps* gruppiert. Dabei werden immer 32 Threads mit fortlaufender Thread-ID zu einem Warp zusammengefügt. Ein solcher Warp wird dann auf einem Multiprozessor nach dem SIMD-Prinzip ausgeführt.

Auf höherer Ebene werden die Blöcke zu *Grids* zusammengefasst (siehe *Abbildung 4.1*), wobei auf jedem Grid der gleiche Kernel ausgeführt wird. Mit der Bearbeitung eines Grids wird erst begonnen, wenn das vorherige Grid vollständig abgearbeitet ist. Zwischen der Ausführung zweier unterschiedlicher Kernels findet also eine globale Synchronisation statt.

In CUDA gibt es drei Möglichkeiten eine Funktion zu deklarieren:

- Eine als `__host__` markierte Funktion wird auf der CPU ausgeführt.
- Als `__global__` markierte Funktionen werden von der CPU aufgerufen, aber auf der GPU ausgeführt.
- Funktionen, die als `__device__` markiert sind, werden von der GPU aufgerufen und komplett auf dieser ausgeführt.

Zu beachten ist hierbei, dass mit `__global__`-Funktionen keine Rekursion möglich ist.

Kernels werden immer von der CPU aufgerufen. Beim Aufruf eines Kernels muss die Dimension des Grids und der Blocks spezifiziert werden.

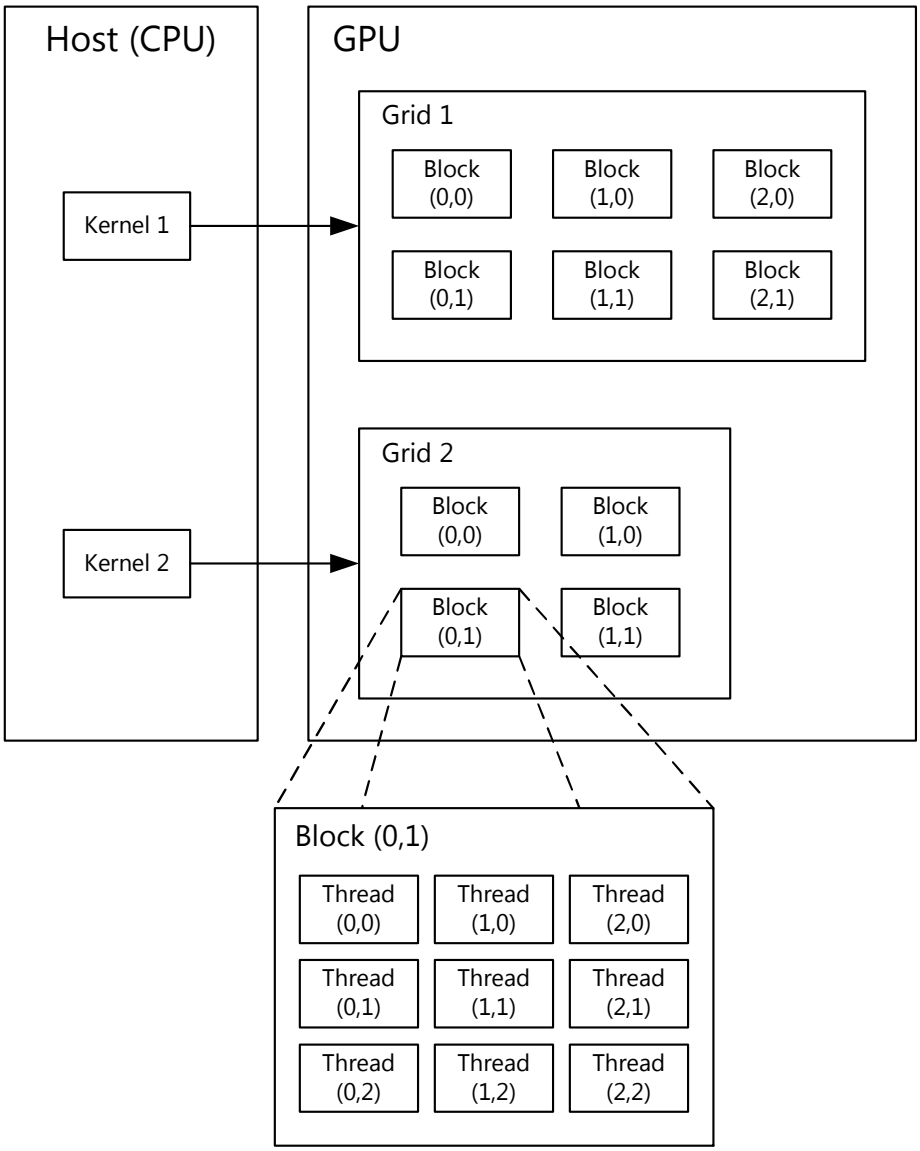

Abbildung 4.1: CUDA Ausführungsmodell

4.1.2 Speicherzugriff

In *Kapitel 3.2* wurden bereits die verschiedenen Speicher von NVIDIA-Hardware-Systemen erwähnt. Jetzt gilt es die Programmierumgebung, die CUDA bereitstellt, mit dieser Speicherorganisation in Verbindung zu bringen.

Logische Speicher in CUDA

Jeder Thread verfügt über eigene Register und einen *Local Memory*. Dieser *Local Memory* ist Teil des *Global Memory*, kann allerdings nur von **einem** Thread benutzt werden. Es ist somit ein für einen Thread reservierter Speicherbereich des *Global Memory*. Die Register werden den Threads vom Register-Pool des jeweiligen Multiprozessors bereitgestellt. Dabei ist die Verteilung der Register über die Threads dynamisch.

Alle Threads eines Blocks haben lesenden und schreibenden Zugriff auf den *Shared Memory*. Alle Threads eines Grids haben lesenden Zugriff auf den *Texture Memory* und den *Constant Memory*, sowie lesenden und schreibenden Zugriff auf den *Global Memory*.

Das Host-System kann nur auf *Global Memory*, *Texture Memory* und *Constant Memory* lesend und schreibend zugreifen. Auf die anderen Speicher hat das Host-System keinen Zugriff.

Synchronisation

Mittels `__syncthreads()` ist es in CUDA möglich, die Threads eines Blocks explizit zu synchronisieren. Es ist jedoch kein blockübergreifender Synchronisationsmechanismus vorgesehen. Mit dem Fertigstellen der Abarbeitung eines Grids werden alle Threads synchronisiert, bevor mit der Ausführung des nächsten Grids begonnen wird. Dies kann als eine Art globale Synchronisation betrachtet und entsprechend eingesetzt werden. Weiterhin ist es möglich, über den *Global Memory* Synchronisationsmechanismen manuell zu implementieren. Beide Lösungen sind jedoch mit eingeschränkter Performanz verbunden.

Latenzen

Die Zugriffszeiten auf die verschiedenen Speicher sind signifikant unterschiedlich. Dies soll in *Tabelle 4.1* aufgezeigt werden. Um diese Ergebnisse zu ermitteln, wurden aus jedem Speicher 20 aufeinanderfolgende Daten aus einem Array gelesen und einer Ausgangsvariable zugewiesen. Die Messung der Zyklen wird mittels der von CUDA bereitgestellen Funktion `clock()` durchgeführt. Die Zeit die dafür benötigt wird fließt dabei mit in das Ergebnis ein. Den größten Einfluss hat diese Zeit natürlich auf die Messung des *Shared Memory*, sodass dessen Zugriffszeit eigentlich deutlich unter der hier gemessenen liegen sollte. Nach [NVI08] sollte die Zugriffszeit gleich einem Registerzugriff sein. Desweiteren fallen die Zuweisungsoperationen negativ ins Gewicht. Diese sind aber nötig, da ansonsten ein ungenutzter Speicherzugriff von der Optimierung durch den Compiler entfernt wird.

Beim *Texture Memory* ist auch zu beachten, dass die genaue Organisation des Caches nicht bekannt ist. So ist davon auszugehen, dass bei dieser Beispielmessung der erste Zugriff ein Cache-Miss ist, somit sehr viel Zeit braucht, die folgenden Zugriffe Cache-Hits sind und somit wesentlich schneller ablaufen.

Speicher	Zugriffszeit (Zyklen)
Texture Memory	214
Global Memory	525
Shared Memory	7

Tabelle 4.1: Speicherlatenzen einer GeForce GTX280 im Vergleich

Funktionsargumente

Für die Übergabe von Funktionsargumenten wird ebenfalls der *Shared Memory* genutzt. Somit steht für die Daten, die ein Algorithmus benötigt, nicht der gesamte *Shared Memory* zur Verfügung.

4.1.3 Implementierungsbeispiel

Als Implementierungsbeispiel soll hier die parallele Vektoraddition in CUDA aufgeführt und erläutert werden. *Listing 4.1* zeigt den entsprechenden CUDA-Quelltext.

Auf den in jedem CUDA-Programm benötigten `include`-Befehl folgt die Definition des Kernels für die Vektoraddition. Dieser bekommt die beiden Eingabevektoren sowie den Ergebnisvektor als Zeiger übergeben. Dabei ist zu beachten, dass es sich bei den übergebenen Zeigern um Zeiger im Adressraum des CUDA-Geräts handelt. Wird der Kernel ausgeführt, so addiert jeder Thread die seinem Index entsprechenden Elemente der Eingabevektoren und speichert das Ergebnis im Ergebnisvektor unter demselben Index.

Im Hauptprogramm werden zunächst auf dem Host zwei Vektoren angelegt und initialisiert, sowie Speicher für den Ergebnisvektor reserviert. Der entsprechende Speicher wird mittels `cudaMalloc()` auch auf dem CUDA-Gerät reserviert. `cudaMalloc()` liefert dabei die Speicheradresse im Adressraum des CUDA-Geräts zurück, an welcher der reservierte Speicherbereich beginnt. Anschließend werden die beiden Eingabevektoren in den reservierten Speicher im *Global Memory* des CUDA-Geräts kopiert.

Nach Ausführen des Kernels wird der Ergebnisvektor aus dem *Global Memory* des CUDA-Geräts in den Hostspeicher kopiert und anschließend das Ergebnis auf der Konsole ausgegeben.

4.2 OpenCL

OpenCL ist ein Ansatz, Portabilität für GPGPU-Programme zu ermöglichen, ohne dabei die Effizienz zu stark einzuschränken. Durch Abstraktion von Grafik-APIs bietet es dem OpenCL-Programmierer die Möglichkeit, ohne Kenntnis einer Grafik-API wie OpenGL oder DirectX *General Purpose*-Programme auf Grafikhardware zu implementieren.

Listing 4.1 Parallele Vektoraddition in CUDA

```
#include <cutil_inline.h>

__global__ void
vectorAddKernel(float* vecA, float* vecB, float* vecResult) {
  vecResult[threadIdx.x] = vecA[threadIdx.x] + vecB[threadIdx.x];
}

int main(int argc, char** argv) {
  float vecA[16] = {0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f, 0.9f,
      1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f};
  float vecB[16] = {2.1f, 2.2f, 2.3f, 2.4f, 2.5f, 2.6f, 2.7f, 2.8f, 2.9f,
      3.0f, 3.1f, 3.2f, 3.3f, 3.4f, 3.5f, 3.6f};
  float vecResult[16];

  float* devVecA;
  cudaMalloc((void**)&devVecA, 16 * sizeof(float));
  cudaMemcpy(devVecA, vecA, 16 * sizeof(float), cudaMemcpyHostToDevice);

  float* devVecB;
  cudaMalloc((void**)&devVecB, 16 * sizeof(float));
  cudaMemcpy(devVecB, vecB, 16 * sizeof(float), cudaMemcpyHostToDevice);

  float* devVecResult;
  cudaMalloc((void**)&devVecResult, 16 * sizeof(float));

  vectorAddKernel<<< 1, 16 >>>(devVecA, devVecB, devVecResult);

  cudaMemcpy(vecResult, devVecResult, 16 * sizeof(float),
      cudaMemcpyDeviceToHost);

  for (int i = 0; i < 16; ++i) {
    printf("vecResult[%i] = %f\n", i, vecResult[i]);
  }

  cutilExit(argc, argv);
}
```

Die OpenCL zugrundeliegenden Prinzipien sind dabei denen von CUDA sehr nahe. So implementiert CUDA in der Version 2.2 die OpenCL-Schnittstelle, welche in [Khro8] spezifiziert ist. Die folgenden Ausführungen beziehen sich auf diese Spezifikation.

4.2.1 OpenCL-Plattform

Ein grundlegender Unterschied ist, dass OpenCL von den konkreten Geräten, die den OpenCL-Code ausführen, abstrahiert. Auf einer OpenCL-Plattform steuert ein Host die Ausführung von Kernels auf einem oder mehreren OpenCL-Geräten, welche durchaus heterogen sein können. Ähnlich den CUDA-fähigen GPUs umfasst ein OpenCL-Gerät eine gewisse Anzahl *Compute Units*, die weiter in *Processing Elements* unterteilt sind. Dies ist analog zu den Streaming Multiprozessoren und Stream-Prozessoren in CUDA. Die *Processing Elements* (PE) einer *Compute Unit* führen denselben Code entweder nach SIMD oder nach SPMD aus (also entweder mit Lockstep oder mit einem eigenen Programmzähler pro *Processing Element*).

Abbildung 4.2 zeigt die OpenCL-Plattform.

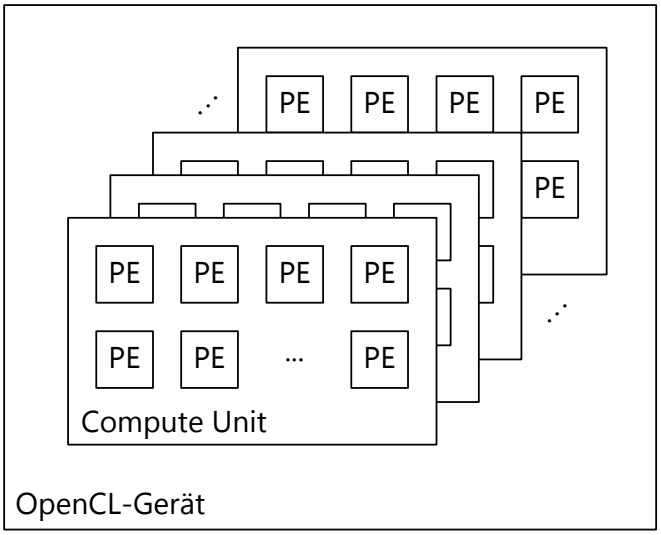

Abbildung 4.2: OpenCL-Plattform

4.2.2 Ausführung eines OpenCL-Programms

OpenCL-Programme werden von einem Host ausgeführt, der zudem eine *Command Queue* anlegt, mit deren Hilfe die Ausführung der Kernels auf den OpenCL-Geräten koordiniert wird. Bei der Abarbeitung des OpenCL-Programms legt der Host Befehle in die *Command Queue*, die dann von den OpenCL-Geräten ausgeführt werden. Es existieren drei Arten solcher Befehle:

- Befehl zur Kernelausführung.

- Speicherbefehle wie Datentransfer zwischen Host- und Gerätespeicher.
- Synchronisationsbefehle.

OpenCL unterstützt zwei Operationsarten: Bei der *In Order Execution* werden die Befehle seriell in der Reihenfolge abgearbeitet, in der sie in der *Command Queue* eintreffen, während bei der *Out of Order Execution* nicht gewartet wird, bis ein Befehl vollständig abgearbeitet ist, sondern bereits zuvor mit der Bearbeitung der folgenden Befehle begonnen wird. Hierbei muss der Programmierer mittels expliziten Synchronisationsbefehlen die korrekte Ausführung seines Programms sicherstellen.

4.2.3 Logische Speicherhierarchie

OpenCL-Kernels können auf vier verschiedene Speicher zugreifen:

- **Global Memory**: Innerhalb jedes Kernels existiert Lese- und Schreibzugriff auf jedes Element des *Global Memory*. OpenCL trifft keine Aussage, ob die Zugriffe gecached sind. Dies hängt vom jeweiligen konkreten Gerät ab.
- **Constant Memory**: Ein Teil des *Global Memory*, der während der Kernelausführung unverändert bleibt. Er wird vom Host initialisiert und beschrieben.
- **Local Memory**: Gemeinsamer Speicher für eine *Work Group*, also für alle *Processing Elements* einer *Compute Unit*. Alle *Processing Elements* der *Compute Unit* haben lesenden und schreibenden Zugriff auf alle Elemente des *Local Memory*. OpenCL trifft keine Aussage darüber, ob es sich um einen physikalisch getrennten Speicher oder nur um einen gesonderten Bereich im *Global Memory* handelt. Auch dies hängt vom jeweiligen konkreten Gerät ab.
- **Private Memory** (PM): Eine Speicherregion, auf die nur ein einzelnes *Work Item*, also die Instanz eines Kernels, Zugriff hat.

Damit ist die logische OpenCL-Speicherhierarchie der logischen CUDA-Speicherhierarchie sehr ähnlich. Da OpenCL jedoch von konkreten Geräten abstrahiert, trifft OpenCL weniger konkrete Aussagen über die Eigenschaften der Speicher. So kann die physikalische Speicherhierarchie von der logischen grundlegend verschieden sein.

Abbildung 4.3 stellt die logische OpenCL-Speicherhierarchie grafisch dar. Dabei sei nocheinmal darauf hingewiesen, dass die genauen Ausführungen der Speicher vom konkreten OpenCL-Gerät abhängen. So können beispielsweise *Local* und *Constant Memory* als Bereiche des *Global Memory* ausgeführt sein und ein Cache kann vollständig entfallen.

4.3 Brook+

Einen weiteren Ansatz für GPGPU stellt die Brook+-Entwicklungsumgebung dar. Genau wie CUDA auch, basiert Brook+ auf den Arbeiten zu BrookGPU, einem Projekt der Stanford Universität. Brook+ wurde von AMD/ATI entwickelt.

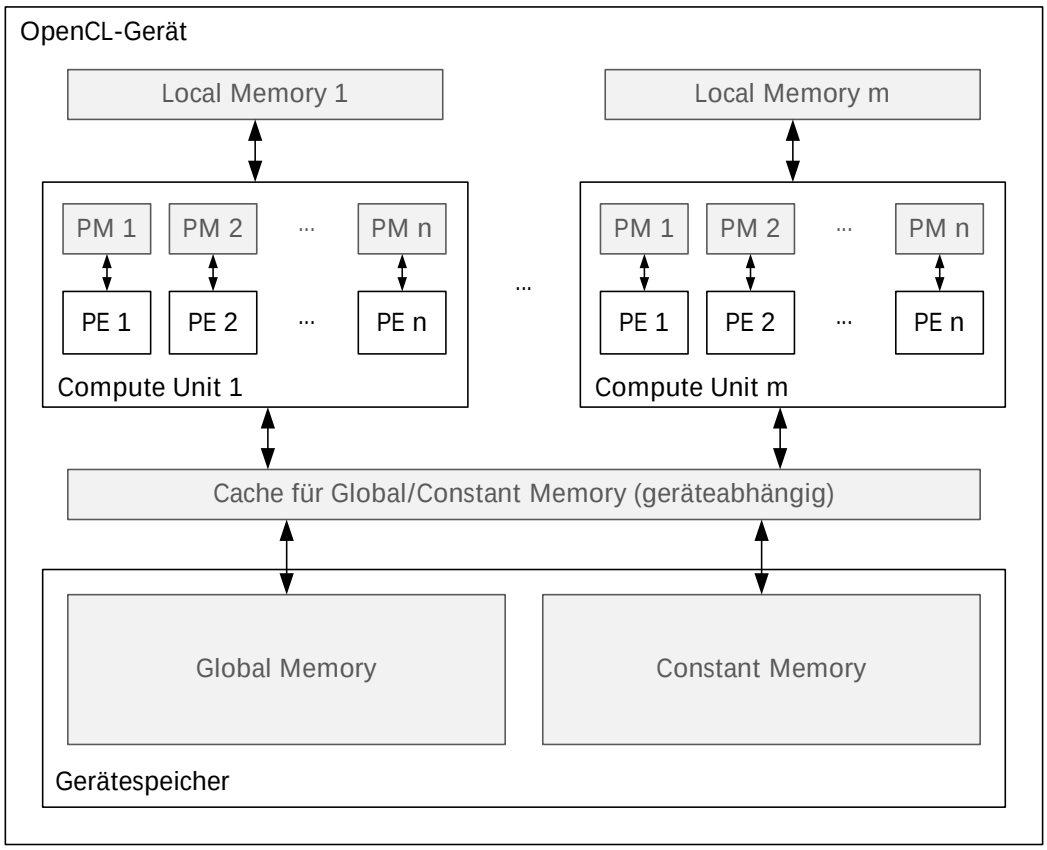

Abbildung 4.3: Logische OpenCL-Speicherhierarchie

Der Brook+-Compiler verwendet die Hochsprache C/C++. Er übersetzt den Programmcode in zwei verschiedene Kompilate. Der Teil, der auf der GPU ausgeführt werden soll, wird in eine Zwischensprache, bei AMD/ATI IL (Intermediate Language) genannt, übersetzt. Diese abstrakte Sprache wird dann schließlich in einen Assembler-Code für die jeweils real verwendete Grafikkarte übersetzt. Die Laufzeitumgebung für die IL (den GPU-Teil) eines Brook+-Programms stellt CAL (Compute Abstraction Layer) dar. Diese ermöglicht die Unterstützung diverser Chipsatzreihen von AMD/ATI. Der Teil, der auf der CPU ausgeführt wird, wird mit einem C/C++-Compiler in Maschinencode übersetzt.

Brook+ hat jedoch keine große Verbreitung gefunden, und unter dem Gesichtspunkt, dass mit OpenCL der Khronos Group eine hardwareunabhängige GPGPU-Programmierumgebung geschaffen wurde, ist die weitere Verbreitung von Brook+ fraglich. Dies liegt vor allem auch an der spärlichen Dokumentation und den wenigen Implementierungsbeispielen, ganz im Gegensatz zu CUDA.

5 Anforderungen an den Algorithmenentwurf

Das Erreichen der maximalen Integrationsdichte bei skalaren Prozessoren führte zum Übergang von serieller zu paralleler Programmierung. Man kann hierbei von einem Paradigmenwechsel sprechen, denn das Augenmerk des Programmierers bei Entwurf und Implementierung effizienter Algorithmen wechselt bei diesem Übergang zu teilweise essentiell anderen Aspekten.

So erfordert die parallele Programmierung einen eigenen Algorithmenentwurf, der das zu lösende Problem in möglichst unabhängige Teilprobleme zerlegt und so die unabhängige parallele Bearbeitung dieser Teilprobleme ermöglicht. Je nach zu lösendem Problem kann dies eine sehr schwierige, bei inhärent seriellen Problemen sogar eine unlösbare, Aufgabe sein, die in jedem Falle vom Programmierer eigenhändig gelöst werden muss.

Während der Anspruch der Speichersparsamkeit bei der seriellen Programmierung im Wesentlichen in der daraus resultierenden Cache-Lokalität und den damit verbundenen signifikanten Effizienzsteigerungen begründet liegt, ist dieser bei der parallelen Programmierung aufgrund der üblicherweise sehr kleinen aber gleichzeitig schnellen, prozessornahen Speicher durch das Hardwaredesign von Multi- beziehungsweise Many-Core-Architekturen unmittelbar gegeben. Dies kann sogar soweit führen, dass es effizienter ist, mehrmals benötigte Daten bei der parallelen Programmierung nicht zwischenzuspeichern, sondern jedes Mal bei Bedarf neu zu berechnen (*we don't cache, we calculate*).

Im Folgenden werden die durch den Übergang von serieller zu paralleler Programmierung implizierten neuen Anforderungen an den Algorithmenentwurf genauer erläutert.

5.1 Ansätze zur Parallelisierung von Algorithmen

Durch Zerlegung der zu bearbeitenden Daten, oder durch Aufteilung des Algorithmus, lassen sich eigentlich serielle Algorithmen in parallele überführen. Diese beiden Ansätze, die sich gegenseitig keinesfalls ausschließen, sollen im Folgenden genauer beschrieben werden.

5.1.1 Datenparallelität

Eine Parallelisierung kann in der Datendimension erfolgen, wenn dasselbe Problem für mehrere Daten gelöst werden soll. Somit ist dieser Ansatz natürlich nur geeignet, wenn ausreichend viele Daten bearbeitet werden sollen. Das Vorgehen besteht im Wesentlichen

darin, die zu bearbeitende Datenmenge möglichst gleichmäßig auf die verschiedenen Recheneinheiten zu verteilen, sodass jede Recheneinheit die ihr zugewiesenen Daten lokal bearbeiten kann. Als einfaches Beispiel soll das Umwandeln eines Videos in ein anderes Format dienen. Das umzuwandelnde Video kann ohne großen Aufwand auf die verfügbaren Recheneinheiten aufgeteilt, und die entstehenden Teile können lokal umgewandelt werden. Dabei ist während des eigentlichen Umwandelvorganges keinerlei Kommunikation zwischen den Recheneinheiten notwendig. Weiterhin führt jede Recheneinheit exakt dieselben Operationen auf unterschiedlichen Daten aus, weshalb es sich hier um eine optimale Realisierung entsprechend des SIMD-Paradigmas handelt. Ein weiteres Beispiel stellt die Fehlersimulation nach [GK08] dar.

Es existiert aber auch eine Vielzahl von Problemen, die sich zwar datenparallel lösen lassen, jedoch nicht ohne Kommunikation zwischen den beteiligten Recheneinheiten. Als Beispiel kann hier eine FEM-Berechnung dienen, die auf mehreren Recheneinheiten parallelisiert wird. Dabei wird das zugrundeliegende Gitter auf die verfügbaren Recheneinheiten verteilt und jede Recheneinheit löst anschließend das FEM-Problem auf ihrem lokalen Gitter. Allerdings müssen die Daten am Rand der lokalen Gitter nach jedem Rechenschritt an andere Recheneinheiten übertragen werden, um das FEM-Problem korrekt zu lösen. Dadurch ist mit einer solchen Parallelisierung ein gewisser Aufwand verbunden.

5.1.2 Aufteilen der Aufgabe

Bei einer Parallelisierung in der Taskdimension versucht man den Algorithmus in mehrere voneinander unabhängige Verarbeitungsschritte aufzuteilen und diese dann auf verschiedenen Recheneinheiten parallel zu bearbeiten. Hierbei ist nur sehr wenig Kommunikation zwischen den Recheneinheiten notwendig. Meist muss nur die Aufgabe verteilt und das Ergebnis akkumuliert werden. Falls es nicht trivial ersichtlich ist, dass der Algorithmus aus unabhängigen Teilschritten besteht, kann eventuell ein *Divide and Conquer*-Ansatz verfolgt werden, um die Teilprobleme zu identifizieren, die parallel gelöst werden können. Der Ansatz der Aufgabenteilung eignet sich natürlich weniger für den Einsatz auf der GPU, da dieser auf dem MIMD-Paradigma aufbaut.

5.2 Speicher

Im Bereich der Speicherorganisation, sowie der Organisation der Speicherzugriffe, bringt der Übergang zu Many-Core-Architekturen besonders wichtige neue Anforderungen an den Algorithmenentwurf mit sich. So müssen Algorithmen explizit auf die Speicherorganisation, beziehungsweise Speicherhierarchie der Zielarchitektur (beispielsweise CUDA), angepasst werden. Zentral ist hierbei nicht nur das Halten von oft benötigten Daten in prozessornahen, schnellen Speichern, sondern insbesondere auch das Zugriffsmuster auf den Speicher. Many-Core-Hardware, wie beispielsweise die NVIDIA GeForce-Grafikkarten, bieten ihre optimale Speicherbandbreite meist nur unter der Voraussetzung, dass der Speicherzugriff entsprechend gewissen Zugriffsmustern erfolgt.

Listing 5.1 Kernel mit vertauschten Speicherzugriffen

```
__global__ void
unOptimizedKernel(float* data) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;

  if (threadIdx.x % 16 == 5) {
    idx = blockDim.x * blockIdx.x + (16 * (threadIdx.x / 16)) + 4;
  }

  if (threadIdx.x % 16 == 4) {
    idx = blockDim.x * blockIdx.x + (16 * (threadIdx.x / 16)) + 5;
  }

  __syncthreads();

  data[idx]++;

  __syncthreads();
}
```

5.2.1 Zugriffsmuster auf den Global Memory

Die Bedeutung der verwendeten Speicherzugriffsmuster soll anhand eines CUDA-Beispiels aufgezeigt werden, bei dem auf den *Global Memory* zugegriffen wird. *Listing 5.1* zeigt einen Kernel, der entsprechend seines Thread-Index ein Datum im *Global Memory* inkrementiert. Dabei greift jeder Thread auf das Datum am Index seines Thread-Index zu, mit der Ausnahme, dass die Speicherzugriffe der Threads 4 und 5 in jedem Threadblock von 16 Threads vertauscht sind. *Listing 5.2* zeigt einen Kernel, der dieselbe Funktionalität bietet, jedoch ohne die Speicherzugriffe zu vertauschen. Um die Laufzeitmessung nicht aufgrund einer unterschiedlichen Berechnungskomplexität zu verfälschen, wurden die Berechnungen der Indizes analog beibehalten, es werden jedoch keine Indizes vertauscht. *Abbildung 5.1* illustriert die entsprechenden Speicherzugriffsmuster.

Führt man diese Kernel hinreichend oft aus, um interpretierbare Messergebnisse zu erhalten, so ergeben sich die in *Tabelle 5.1* gezeigten Ergebnisse. Dabei fällt auf, dass auf CUDA-Geräten mit *Compute Capability* 1.1 die Laufzeit des Kernels mit vertauschten Speicherzugriffen um einen Faktor von etwa 6 höher ist als bei konsekutiven Speicherzugriffen. Grund dafür ist, dass bei konsekutivem Zugriff eine Speicherzugriffe in eine Transaktion verschmolzen werden können, was bei nicht konsekutivem Zugriff nicht möglich ist. Bemerkenswert ist nun, dass diese Laufzeitdiskrepanz bei CUDA-Geräten der *Compute Capability* 1.3 nicht auftritt, weil diese einen solchen wahlfreien Zugriff auf den *Global Memory* effizient ausführen können.

Listing 5.2 Kernel ohne vertauschte Speicherzugriffe

```
__global__ void
optimizedKernel(float* data) {
  int idx = blockDim.x * blockIdx.x + threadIdx.x;

  if (threadIdx.x % 16 == 5) {
    idx = blockDim.x * blockIdx.x + (16 * (threadIdx.x / 16)) + 5;
  }

  if (threadIdx.x % 16 == 4) {
    idx = blockDim.x * blockIdx.x + (16 * (threadIdx.x / 16)) + 4;
  }

  __syncthreads();

  data[idx]++;

  __syncthreads();
}
```

Beim Entwurf von Algorithmen müssen also stets die Eigenschaften der konkreten Zielplattform berücksichtigt werden. Auch beim Einsatz von CUDA können zwischen verschiedenen CUDA-Geräten erhebliche Unterschiede bestehen, die beim Entwickeln von hochperformanten Algorithmen beachtet werden müssen.

CUDA-Gerät	Compute Capability	konsekutiver Zugriff	Laufzeit (ms)
GeForce 9600GT	1.1	ja	4674
GeForce 9600GT	1.1	nein	28453
GeForce GTX 280	1.3	ja	1938
GeForce GTX 280	1.3	nein	1939

Tabelle 5.1: Messergebnisse beim Zugriff auf den Global Memory

5.2.2 Bankkonflikte beim Zugriff auf den Shared Memory

Die Nähe des *Shared Memory* zu den Recheneinheiten sorgt dafür, dass seine Latenzzeit sehr gering ist. Dennoch können ungünstige Zugriffsmuster enorme Performanzeinbußen zur Folge haben, da der *Shared Memory* in *Bänke* unterteilt ist. Diese sind so organisiert, dass benachbarte 32-bit Worte in benachbarten Bänken liegen. Bei CUDA-Geräten der *Compute Capability 1.x* ist der *Shared Memory* laut [NVI08] in 16 Bänke unterteilt. Greifen zwei oder mehr Threads innerhalb eines Warps auf dieselbe Speicherbank zu, so werden die Speicherzugriffe hintereinander abgearbeitet, was eine signifikante Verzögerung mit sich bringt.

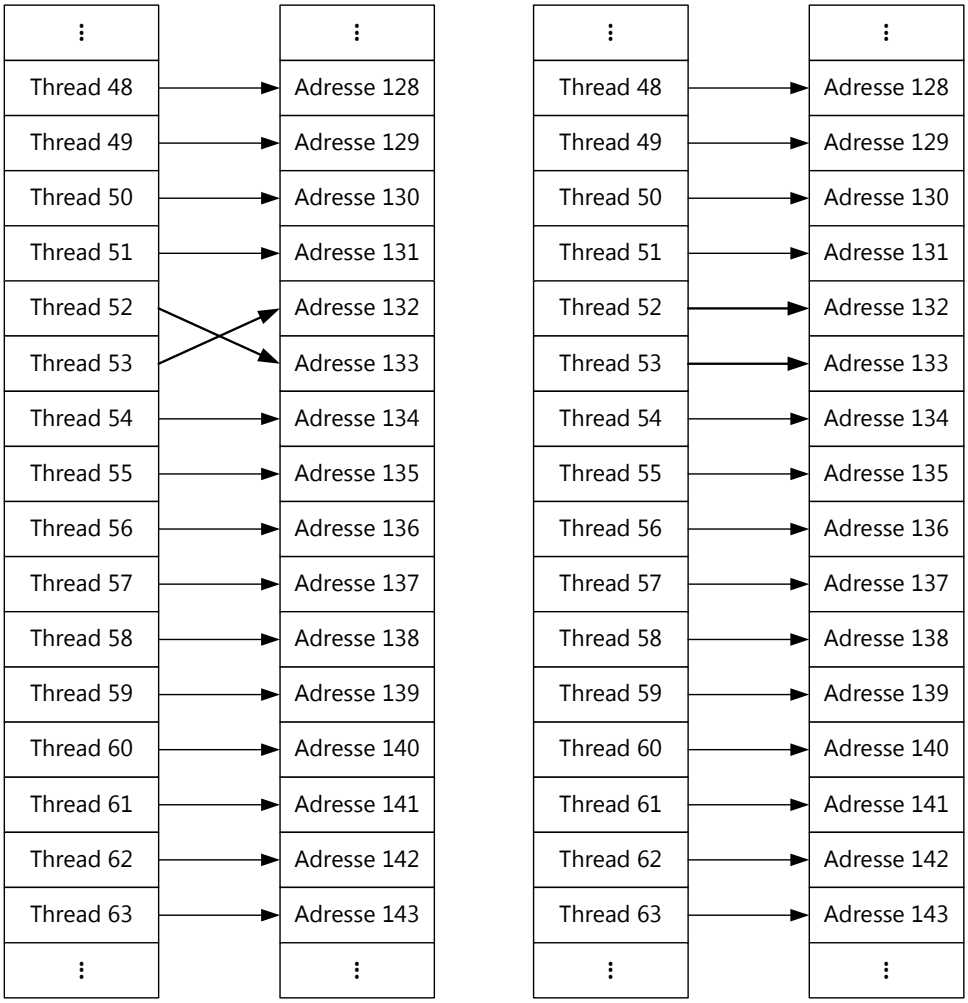

Abbildung 5.1: Speicherzugriffsmuster

Auch diese Eigenschaft soll mit einer CUDA-Beispielmessung belegt werden. *Listing 5.3* zeigt einen Kernel, bei dessen Ausführung abhängig von der gewählten Schrittweite (Parameter `stride`) Bankkonflikte unterschiedlicher Ordnung auftreten. Man spricht dabei von einem *k*-fachen Bankkonflikt, wenn *k* Threads gleichzeitig auf dieselbe Speicherbank zugreifen. Der Kernel wurde auf einer GeForce 9600GT für alle Schrittweiten zwischen 1 und 32 jeweils 61440 Mal ausgeführt, was zu den in *Tabelle 5.2* dargestellten Ergebnissen führt.

Liegen 16-fache Bankkonflikte vor, so ist die Laufzeit des Kernels bei dieser Messung also um einen Faktor von etwa 3.64 höher als bei einer Ausführung ohne Bankkonflikte (Schrittweite 1). Bei einer Schrittweite von 32 (und dementsprechend 32-fachen Bankkonflikten) stellt sich keine weitere Erhöhung der Laufzeit ein, was darauf zurückzuführen ist, dass in CUDA jeweils nur ein sogenannter *Halfwarp*, also 16 Threads, überlappend zur Ausführung gebracht werden.

Listing 5.3 Beispielkernel für Bankkonflikte

```
__global__ void
kernel(int stride)
{
  __shared__ int data[BLOCKSIZE];

  for (int i = 0; i < 1000; ++i)
  {
    data[(stride*threadIdx.x)%32] = 1235;
  }
}
```

Die durch Bankkonflikte hervorgerufene, deutliche Laufzeiterhöhung zeigt dennoch die große Bedeutung der Speicherzugriffsmuster beim Einsatz des *Shared Memory*. Es kann also beim Entwurf von Algorithmen für Many-Core-Architekturen nicht davon ausgegangen werden, dass die nahe an den Recheneinheiten gelegenen Speicher für alle Arten des Zugriffs ihre maximale Performanz aufweisen. Vielmehr müssen auch die Charakteristika dieser Speicher in die Konzeption von Algorithmen eingehen, um optimale Performanz zu erreichen.

5.2.3 Cache

Um den Nutzen des Cachings hervorzuheben, wurden verschiedene Speicherarten miteinander verglichen. Diese sind der *Texture Memory* (1D-Datenstruktur), der *Global Memory* und der *Shared Memory*. Dabei wird in jedem der Kernels aus *Listing 5.4* ein Array der Größe `ARRAY_SIZE` mit Integer-Elementen mehrfach durchlaufen, auf jedes Element dreimal zugegriffen und aufsummiert. Ausgeführt wird der Kernel jeweils mit einem Block und einem Thread.

Auf einer GeForce GTX 280 ergeben sich die in *Tabelle 5.3* aufgeführten Laufzeiten.

Das Ergebnis zeigt deutlich den Laufzeitvorteil des gecachten *Texture Memory* gegenüber dem *Global Memory*. Wie erwartet ist der *Shared Memory* am schnellsten, da hier kein Zugriff auf den *Global Memory* notwendig ist.

5.3 Kontrollfluss

Many-Core-Architekturen erreichen ihre maximale Performanz meist nur dann, wenn alle Threads in jedem Takt entsprechend dem SIMD-Paradigma exakt dieselben Anweisungen ausführen. Datenabhängige Verzweigungsanweisungen können aber dazu führen, dass auf verschiedenen Threads verschiedene Anweisungen ausgeführt werden müssen, worauf beispielsweise CUDA-Geräte nicht ausgelegt sind. Springen in CUDA zwei Threads eines

Listing 5.4 Kernel mit verschiedenen Speicherzugriffen

```
#define ITERATIONS 5000
#define ARRAY_SIZE 1024

__global__
void readTexture(int* out) {
  for(int j = 0; j < ITERATIONS; ++j) {
    *out = 0.0f;
    for(int i = 0; i < ARRAY_SIZE; ++i) {
      *out += tex1Dfetch(texRef, float(i));
      *out += tex1Dfetch(texRef, float(i));
      *out += tex1Dfetch(texRef, float(i));
    }
  }
}

__global__
void readGlobal(int* out, int* arr) {
  for(int j = 0; j < ITERATIONS; ++j) {
    *out = 0.0f;
    for(int i = 0; i < ARRAY_SIZE; ++i) {
      *out += arr[i];
      *out += arr[i];
      *out += arr[i];
    }
  }
}

__global__
void readShared(int* out) {
  for(int j = 0; j < ITERATIONS; ++j) {
    *out = 0.0f;
    for(int i = 0; i < ARRAY_SIZE; ++i) {
      *out += d_shared_array[i];
      *out += d_shared_array[i];
      *out += d_shared_array[i];
    }
  }
}
```

Schrittweite	Ordnung	Laufzeit (ms)
1	**-**	**3498**
2	**2-fach**	**4084**
3	-	3509
4	**4-fach**	**5321**
5	-	3488
6	2-fach	4105
7	-	3458
8	**8-fach**	**7831**
9	-	3459
10	2-fach	4130
11	-	3465
12	4-fach	5332
13	-	3457
14	2-fach	4085
15	-	3461
16	**16-fach**	**12752**
17	-	3456
18	2-fach	4085
19	-	3461
20	4-fach	5324
21	-	3461
22	2-fach	4080
23	-	3462
24	8-fach	7796
25	-	3461
26	2-fach	4080
27	-	3467
28	4-fach	5315
29	-	3466
30	2-fach	4078
31	-	3457
32	**32-fach**	**12749**

Tabelle 5.2: Bankkonflikte auf einer GeForce 9600GT

Warps in unterschiedliche Zweige (man spricht auch von *divergierenden Threads*), so berechnen dennoch alle Threads dieses Warps jeweils alle Zweige seriell. Dies kann insbesondere bei verschachtelten, datenabhängigen Verzweigungen zu enormen Performanzeinbußen führen, da hier dann jeder Thread den gesamten Anweisungsbaum ausführen muss, obwohl für ihn selbst letztendlich nur ein Zweig relevant ist.

Speicher	Laufzeit (ms)
Texture Memory	2389
Global Memory	5988
Shared Memory	1091

Tabelle 5.3: Verschiedene Speicher im Vergleich

Diese Situation divergierender Threads beeinträchtigt die Korrektheit des Programms in keinster Weise, jedoch stellt sie auf der CUDA-Plattform einen außerordentlich performanzhemmenden Faktor dar. Dies soll das folgende CUDA-Beispiel demonstrieren, das in *Listing 5.5* einen Kernel mit vielen datenabhängigen Verzweigungen und in *Listing 5.6* einen Kernel mit nur wenigen datenabhängigen Verzweigungen umfasst.

Auf einer GeForce 9600GT ergibt sich für den ersten Kernel eine Laufzeit von 1988 Millisekunden, für den zweiten 939 Millisekunden. Dies entspricht einem Faktor von ungefähr 2.1. Jeder Kernel wurde für diese Messung 450 Mal ausgeführt.

Es zeigt sich also, dass bei der Abbildung von Algorithmen auf Many-Core-Architekturen kontrollflussbezogene Einschränkungen unbedingt beachtet werden müssen, um eine hohe Performanz des abgebildeten Algorithmus sicherzustellen. Die beiden bisher existierenden GPGPU-Realisierungen von NVIDIA und AMD/ATI weisen beide die beschriebene Einschränkung auf, das heißt optimale Performanz wird nur bei strikter Einhaltung des SIMD-Paradigmas innerhalb jedes Warps (bei AMD/ATI *Wavefront*) erreicht. Demnach ist es für hochperformante GPGPU-Implementierungen momentan unverzichtbar, die Zahl der datenabhängigen Verzweigungen möglichst gering zu halten.

5.3.1 MIMD

GPGPU ist hauptsächlich für Algorithmen geeignet, die dem SIMD-Paradigma folgen. Das bedeutet, dass alle gestarteten Threads zu jedem Zeitpunkt dieselbe Instruktion ausführen. Eine Ausführung nach dem MIMD-Prinzip wird von CUDA nicht unterstützt, weil zu jedem Zeitpunkt nur ein Kernel auf der GPU aktiv sein kann. Man könnte jedoch mittels einer Fallunterscheidung in unterschiedlichen Threads gänzlich verschiedenen Programmcode auf verschiedenen Daten ausführen und damit eine MIMD-ähnliche Ausführung realisieren, wobei die Threads lediglich einen anderen Zweig desselben Kernels ausführen. Bei einer solchen Realisierung ist jedoch darauf zu achten, dass innerhalb eines Warps alle Threads weiterhin möglichst identischen Code ausführen, um die oben beschriebene Situation divergierender Threads und die damit verbundenen Performanzeinbußen zu vermeiden. Zudem ist die Ausführung erst dann beendet, wenn der Thread mit der längsten Ausführungszeit terminiert.

Listing 5.5 Kernel mit vielen Pfaden

```
__global__ void
kernel(int* devData, float* sum) {
  *sum = 0.0f;
#ifdef __DEVICE_EMULATION__
  printf("thread %i: i have got data %i\n", threadIdx.x,
      devData[threadIdx.x]);
#endif
  float temp = 0;
  int mod = devData[threadIdx.x] % 8;
  for (int i = 0; i < KERNELITERATIONS; ++i) {
    switch (mod) {
      case 0:
        temp += 5;
        break;
      case 1:
        temp -= 5;
        break;
      case 2:
        temp += 5;
        break;
      case 3:
        temp -= 5;
        break;
      case 4:
        temp += 5;
        break;
      case 5:
        temp -= 5;
        break;
      case 6:
        temp += 5;
        break;
    case 7:
      temp -= 5;
      break;
    }
  }

  sum[0] = temp;
  __syncthreads();
}
```

Listing 5.6 Kernel mit wenigen Pfaden

```
__global__ void
kernel_op(int* devData, float* res) {
  float sum = 0;
  int mod = devData[threadIdx.x]%2;

#ifdef __DEVICE_EMULATION__
  printf("thread %i: i have got data %i\n", threadIdx.x,
      devData[threadIdx.x]);
#endif
  for (int i = 0; i < KERNELITERATIONS; ++i) {
    switch (mod) {
      case 0:
        sum += 5;
        break;
      case 1:
        sum += 5;
        break;
    }
  }

  res[0] = sum;
  __syncthreads();
}
```

5.4 Auslastung

Bei der Programmierung für Many-Core-Architekturen ist es enorm wichtig, jederzeit alle Recheneinheiten beschäftigt zu halten, da ungenutzte Recheneinheiten eine Verminderung der Performanz und damit brachliegendes Potential implizieren.

Da Threads oft durch Zugriffe auf den ungecacheten Speicher, durch *Cache-Misses* beim Zugriff auf gecachete Speicher oder auch durch Bankkonflikte bei Speicherzugriffen zu beträchtlichen Wartezeiten gezwungen werden, ist das Starten eines Threads für jede Recheneinheit unzureichend, um alle Recheneinheiten jederzeit beschäftigt zu halten.

Diesem Problem kann prinzipiell auf zwei Arten begegnet werden. Einerseits kann weitestmöglich auf Speicherzugriffe verzichtet werden. Mit anderen Worten soll das Verhältnis von Speicherzugriffen zu, auf den zugehörigen Daten ausgeführten, Operationen minimiert werden. In diesem Zusammenhang kann es sich auch lohnen, ein Datum mehrmals zu berechnen anstatt es im Speicher abzulegen (*we don't cache, we calculate*). Ein solcher Einsatz verringert offensichtlich das angesprochene Verhältnis von Speicherzugriffen zu ausgeführten Operationen.

Andererseits ist es möglich, dem Problem mit aggresivem Parallelismus entgegenzutreten. Dabei werden so viele Threads gestartet, dass der Scheduler jederzeit über eine ausreichende Anzahl einsatzbereiter Threads verfügt, die er zur Ausführung bringen kann, um so die Recheneinheiten beschäftigt zu halten, während andere Threads auf Speicherzugriffe warten. Es sollten mindestens sechs aktive Warps pro Multiprozessor vorhanden sein, sodass der CUDA-Scheduler durch verzahnte Ausführung die Instruktionslatenzzeit geeignet verstecken kann.

Eine Kombination aus beiden Ansätzen erscheint dabei sinnvoll. Führt ein Thread auf einem Datum ausreichend viele Operationen aus, und wurden ausreichend viele Threads gestartet, so kann der Scheduler die Recheneinheiten ausgelastet halten, während andere Threads auf Daten aus dem Speicher warten. Damit lässt sich die Speicherlatenzzeit weitgehend verbergen (engl. *Memory Latency Hiding*).

6 Anwendungen im EDA-Bereich

Um die Bedeutung von GPGPU im Bereich der *Electronic Design Automation* aufzuzeigen, soll im Folgenden die Abbildung zweier prominenter Anwendungen im EDA/CAD-Bereich auf GP-GPUs (*General-Purpose Graphics Processing Units*) beschrieben werden. Konkret handelt es sich dabei um Fehlersimulation von Schaltnetzen und die Generierung von Phase-Shiftern für *Built-in-Self-Test*-Anwendungen.

6.1 Fehlersimulation

Beim Entwurf von VLSI-Schaltkreisen (*Very Large Scale Integration*) spielt Fehlersimulation eine sehr wichtige Rolle. Das übergeordnete Ziel ist, für die Eingangsgatter eines Schaltkreises solche Testmuster zu finden, dass diese im Falle einer Störung des Schaltkreises eine Veränderung der Ausgangsgatter hervorrufen. Solche Testmuster, auch *Vektoren* genannt, sind geeignet um die Funktion des Schaltkreises zu testen. Um geeignete Testmuster zu finden, müssen zunächst Testmuster generiert werden, bevor der *gestörte* Schaltkreis mit dem generierten Testmuster als Eingabe simuliert wird. Das automatische Erzeugen von Testmustern wird als *Automatic Test Pattern Generation* bezeichnet und ist nicht Teil dieser Fachstudie. Liegen die Testmuster vor, so werden die aus der Simulation des *gestörten* Schaltkreises erhaltenen Werte an den Ausgangsgattern mit denen aus der Simulation des *ungestörten* Schaltkreises verglichen. Treten dabei Unterschiede auf, so ist das betrachtete Testmuster geeignet, Fehler im Schaltkreis aufzudecken.

Der Prozess, Schaltkreise für gegebene Testmuster zu simulieren, ist jedoch sehr aufwändig, da heutige VLSI-Schaltkreise häufig Millionen von Logikgattern umfassen. Aus diesem Grund ist es erforderlich, sehr schnelle, skalierbare und dennoch kostengünstige Verfahren zur Fehlersimulation zu finden.

Da im Rahmen der Fehlersimulation sehr viele Testmuster auf unterschiedlich gestörten Schaltkreisen simuliert werden müssen, und diese Simulationen vollkommen unabhängig voneinander sind, erscheint die Fehlersimulation ideal geeignet für parallele Ansätze. Als mit GPGPU-fähigen Grafikkarten sehr günstige Geräte für hochgradig paralleles Rechnen erschienen, wurden diese interessant für den Einsatz im Bereich der Fehlersimulation.

6.1.1 Ansatz von Gulati und Khatri

In [GK08] stellen Kanupriya Gulati und Sunil Khatri einen Ansatz zur Fehlersimulation auf Grafikprozessoren unter Verwendung von CUDA vor. Dabei erfolgt eine Parallelisierung sowohl über Fehler, wie auch über Testmuster. Das bedeutet, dass verschiedene Testmuster auf verschieden gestörten Varianten des betrachteten Schaltkreises parallel simuliert werden.

Der zu simulierende Schaltkreis wird in *Levels* eingeteilt, wobei die Eingabegatter auf *Level* 0 liegen und für alle anderen Gatter G gilt: $level(G) = \max_{i \in fanin(G)} level(i) + 1$. Der Schaltkreis wird dann Level für Level von den Eingangs- zu den Ausgangsgattern simuliert.

Dazu benötigt jeder simulierende Thread zunächst Zugriff auf die Gatterbibliothek. Diese wird in Form einer Look-Up-Tabelle (LUT) im *Texture Memory* der Grafikkarte hinterlegt, wobei diese so angelegt wird, dass sich mit einem Look-Up für ein Gatter die Ergebnisse für zwei verschiedene Eingaben lesen lassen.

Im in [GK08] vorgestellten Ansatz berechnet jeder Thread die Ergebnisse eines Gatters für zwei verschiedene Eingaben, wobei das simulierte Gatter auch gestört sein kann. Dazu benötigt der Thread die Information, welches Gatter simuliert werden soll, welche Eingaben vorliegen sowie ob und gegebenenfalls in welcher Weise das Gatter gestört ist. Diese Informationen werden in einer Datenstruktur für jeden Thread im *Global Memory* der Grafikkarte abgelegt.

Der Kernel zur Fehlersimulation führt damit nur folgende Schritte durch:

1. Lese Informationen aus dem *Global Memory*.
2. Lese Ergebnis aus LUT aus.
3. Führe logische Verknüpfungen auf dem Ergebnis aus, um gegebenenfalls Störungen zu injizieren.
4. Schreibe Ergebnis in den *Global Memory* zurück.

Für das letzte Level, also für die Ausgangsgatter des Schaltkreises, wird ein geringfügig anderer Kernel ausgeführt:

1. Lese Informationen aus dem *Global Memory*.
2. Lese Ergebnis aus LUT aus.
3. Falls dieser Thread den *ungestörten* Schaltkreis simuliert, lege das Ergebnis an eine dafür ausgezeichnete Stelle im *Global Memory*.
4. Sonst vergleiche das Ergebnis mit dem bei der Simulation des *ungestörten* Schaltkreises erhaltenen Ergebnis und lege das Resultat im *Global Memory* ab.

Damit ist die Information, welche Testmuster bei welchen Störungen eine Änderung der Werte an den Ausgangsgattern hervorrufen, im *Global Memory* verfügbar und kann vom Hostprogramm ausgelesen werden.

6.1.2 Bewertung des Ansatzes

Die Beachtung folgender wesentlicher Punkte gewährleistet die Eignung des Ansatzes von Gulati und Khatri für den Einsatz auf CUDA-Geräten:

- Es existieren keinerlei Datenabhängigkeiten zwischen den Threads.
- Alle Threads führen exakt dieselben Instruktionen auf verschiedenen Daten aus (SIMD).
- Die Simulation der Logikgatter erfolgt über eine Look-Up-Tabelle im *Texture Memory* des CUDA-Geräts.

Da die Threads keinerlei Daten untereinander austauschen müssen, ist kein Synchronisationsmechanismus notwendig und es besteht keine Gefahr von Stillstandzeiten, die beispielsweise durch Bankkonflikte beim Zugriff auf den *Shared Memory* entstehen können.

Ebenso wichtig ist die Tatsache, dass der in [GK08] gezeigte Kernel zur Fehlersimulation keine Verzweigungen enthält. Damit führt jeder Thread exakt dieselben Instruktionen aus, was dem SIMD-Paradigma entspricht und bei der Ausführung die Entstehung von divergierenden Threads und den damit verbundenen Performanzeinbußen verhindert.

Schließlich ist der *Texture Memory* zur Ablage der für die Simulation der Logikgatter benötigten Look-Up-Tabellen aus folgenden Gründen geeignet:

- Der *Texture Memory* ist im Gegensatz zum *Global Memory* gecached, das heißt nur im Falle eines Cache-Misses müssen Daten aus dem *Global Memory* nachgeladen werden. Zudem wurde die Größe der LUT so gewählt, dass sie komplett in den Cache des *Texture Memory* passt.
- Anders als der *Global* und der *Constant Memory* arbeitet der *Texture Memory* effizient bei willkürlichen Speicherzugriffsmustern.
- Im Vergleich zum *Shared Memory* können beim Zugriff auf den *Texture Memory* keine Bankkonflikte auftreten, beispielsweise wenn zwei Threads auf das selbe Datum im Speicher zugreifen.
- Für den *Texture Memory* bietet CUDA vorgefertigte, extrem effiziente Zugriffsmethoden.
- Laut [NVI08] wird bei Einsatz des *Texture Memory* die mit Adressberechnungen verbundene Latenzzeit besser versteckt, was bei Anwendungen mit willkürlichen Speicherzugriffsmustern, wie der Fehlersimulation, zu Performanzverbesserungen führen kann.

Weiterhin ist anzumerken, dass der in [GK08] beschriebene Ansatz für jede Gatterauswertung einen eigenen Thread startet, was zur Folge hat, dass jeweils nur die für diese Gatterauswertung benötigten Daten, und nie der gesamte Schaltkreis, im Speicher des CUDA-Geräts gehalten werden müssen. Damit stellt der beschränkte Speicher des CUDA-Geräts keine Schranke für die Größe des Schaltkreises dar.

Als kritischer Punkt dieses Ansatzes ist anzuführen, dass die Kernel zur Fehlersimulation hauptsächlich Speicherzugriffe ausführen und nur sehr wenige Berechnungen auf den gelesenen Daten ausführen. Wie in *Kapitel 3.1* dargestellt, eignen sich für GPGPU vor allem

Listing 6.1 Verzweigungskonstrukt

```
if (x) {
  z = a & b;
} else {
  z = 0;
}
```

Listing 6.2 Verzweigungskonstrukt als logische Operation

```
z = (a & b) & x;
```

solche Algorithmen, die auf einer bestimmten Datenmenge möglichst viele Instruktionen ausführen. Dies ist bei dem in [GK08] gezeigten Ansatz nicht der Fall. Der Kernel zur Fehlersimulation enthält im Wesentlichen zwei kostspielige Zugriffe auf den *Global Memory*, einen Look-Up im *Texture Memory* und einige Bitoperationen auf Integer-Zahlen. Das Verhältnis von Speicherzugriffen zu auf den entsprechenden Daten ausgeführten Operationen ist somit sehr groß und damit ungünstig für GPGPU-Anwendungen.

Dennoch erzielt der Ansatz laut [GK08] gegenüber einem aktuellen, kommerziellen Fehlersimulationswerkzeug einen durchschnittlichen *Speed-Up*-Faktor von etwa 38. Damit ist der in [GK08] beschriebene Ansatz zur Fehlersimulation als gutes Beispiel für den Einsatz von Grafikprozessoren, oder Many-Core-Architekturen im Allgemeinen, im EDA-CAD-Bereich zu sehen. Bei diesem Ansatz wurden die von der CUDA-Architektur implizierten Bedingungen an den Algorithmenentwurf mit der oben genannten Ausnahme berücksichtigt, und so ein effizienter Algorithmus zur Fehlersimulation entworfen.

6.1.3 Mögliche Modifikation

In diesem Abschnitt soll eine mögliche Modifikation des in [GK08] beschriebenen Ansatzes vorgeschlagen werden. In [GK08] wird in jedem Kernel ein Gatter für zwei verschiedene Eingaben mittels eines Look-Ups ausgewertet. Den Autoren von [GK08] zufolge würde eine Look-Up-Tabelle zur Auswertung eines Gatters für drei oder mehr verschiedene Eingaben nicht mehr in den Cache des *Texture Memory* passen und damit Performanzeinbußen mit sich bringen.

Würde man statt des Auswertens der Gatter über die Look-Up-Tabelle deren logische Operationen direkt im Kernel kodieren, so könnte man jedoch ein Gatter für 32 verschiedene Eingaben in einem einzigen Thread auswerten und könnte zusätzlich auf den *Texture Memory*-Zugriff verzichten. Dabei kann man ausnutzen, dass das in *Listing 6.1* gezeigte Verzweigungskonstrukt durch die in *Listing 6.2* gezeigten logischen Operationen ausgedrückt werden kann, wobei für $x = -1$ (alle Bits 1) der if-Zweig und für $x = 0$ der else-Zweig ausgeführt wird.

Auf diese Weise lässt sich die Gatterbibliothek im Kernelcode selbst kodieren, ohne dabei Sprunganweisungen einsetzen zu müssen. Solche Sprunganweisungen würden zu divergie-

renden Threads innerhalb eines Warps führen und somit drastische Performanzeinbußen mit sich bringen (siehe *Kapitel 5.3*). Dazu werden für größere Gatterbibliotheken eine große Anzahl an logischen Bitoperationen benötigt, die jedoch sehr effizient berechenbar sind.

Insgesamt müssten weniger Daten aus dem Speicher gelesen werden (die teuren Zugriffe auf den *Global Memory* blieben aber erhalten) und auf diesen Daten würden mehr Operationen ausgeführt. Gleichzeitig könnte ein Gatter in einem Thread nicht nur für zwei sondern für 32 verschiedene Eingaben ausgewertet werden.

Dieser Ansatz könnte somit eine weitere Steigerung des erzielten Speed-Ups ermöglichen.

6.1.4 Andere Ansätze

Neben dem in [GK08] beschriebenen Ansatz existieren noch weitere Ansätze zur Parallelisierung von Fehlersimulationen. An dieser Stelle soll der in [CDB09] beschriebene *Gate-level Concurrent Simulator* (GCS) von Debapriya Chatterjee, Andrew DeOrio und Valeria Bertacco kurz skizziert werden.

Bei diesem Ansatz werden die Daten zunächst auf dem Hostsystem für die Ausführung auf dem CUDA-Gerät optimiert. Dazu wird der zu simulierende Schaltkreis in das GCS-interne Datenformat überführt und der kombinatorische Schaltplan extrahiert. Nun wird der Ansatz verfolgt, dass auf dem CUDA-Gerät alle Level der Schaltung simuliert werden sollen, ohne dabei mit dem Hostsystem kommunizieren zu müssen. Da in CUDA zwischen Threadblöcken keine Kommunikation möglich ist, muss für jeden Thread der gesamte, für die von diesem Thread durchgeführte Simulation benötigte, Teil des Schaltplans auf dem CUDA-Gerät vorliegen.

Zusätzlich muss jeder Thread zu allen für ihn relevanten Gattern die aktuellen Zustände speichern. Dies erfolgt in GCS in Form einer Wertematrix, die im *Shared Memory* abgelegt wird, während Eingabe- und Ausgabewerte im *Global Memory*, sowie die Schaltplantopologie im *Texture Memory* abgelegt werden. Aufgrund des relativ kleinen *Shared Memory* ergibt sich damit eine Beschränkung für die Größe des von einem Thread simulierten Teils des Schaltplans. Diese wird vom GCS-Compiler in einem sogenannten *Clustering*-Schritt berücksichtigt, bei dem der Schaltkreis so zerlegt wird, dass alle für die Berechnung des Werts eines Ausgangsgatters benötigten Gatter einen *Cone* (nach [CDB09]) bilden. Mehrere solcher Cones werden zu einem Cluster zusammengefasst, und zwar so, dass die einem Cluster zugewiesenen Cones eine möglichst große Überlappung aufweisen. Damit sind die Cluster meist nicht disjunkt, können aber völlig unabhängig voneinander simuliert werden. Diese Cluster werden dann auf die verfügbaren Multiprozessoren verteilt.

Da jeder Cluster aus einer Vielzahl unabhängiger Cones besteht, führt GCS im *Balancing*-Schritt eine Optimierung der Auswertungsreihenfolge aus, mit dem Ziel, dass mehr Gatter parallel ausgewertet werden können. Für weitere Details zu diesem *Balancing*-Schritt sei auf [CDB09] verwiesen.

Nach dem *Balancing* werden die Daten auf das CUDA-Gerät übertragen und die Simulation gestartet.

Der wohl größte Unterschied zum Ansatz von Gulati und Khatri ist der, dass das Hostsystem bei GCS, vom Kompiliervorgang abgesehen, wesentlich weniger in die eigentliche Simulation einbezogen ist. Es ist im Zuge der Simulation keinerlei Kommunikation oder Datenaustausch zwischen Hostsystem und CUDA-Gerät erforderlich. Da die aktuellen Gatterwerte in GCS im *Shared Memory* und die Schaltplantopologie im *Texture Memory* abgelegt sind, ist der Zugriff auf häufig benötigte Daten effizient möglich. Zudem ist das Verhältnis von teuren Zugriffen auf den *Global Memory* zu den ausgeführten Operationen bei GCS deutlich günstiger als bei dem in [GK08] beschriebenen Ansatz.

Negativ anzumerken ist, dass GCS die Werte für bestimmte Gatter mehrmals berechnet, weil die Cluster nicht notwendigerweise disjunkt sind. Laut [CDB09] hat dies jedoch keine großen Auswirkungen, da die Überlappung der Cluster in der Anwendung meist klein ist und der zusätzliche Aufwand deshalb gegenüber dem Aufwand für die restliche Simulation nicht ins Gewicht fällt.

GCS erzielt gegenüber einem aktuellen, ereignisgesteuerten Fehlersimulationswerkzeug laut [CDB09] lediglich einen *Speed-Up*-Faktor von durchschnittlich 14.4, wobei jedoch für spezielle Anwendungen ein *Speed-Up*-Faktor von bis zu 62 erreicht wird. Allerdings ist unklar, mit welchen Simulationswerkzeugen die Ergebnisse aus [GK08] und [CDB09] verglichen wurden. Somit können die Speed-Up-Faktoren nicht sinnvoll gegenübergestellt werden.

Als potentielles Problem von GCS ist zu sehen, dass bei diesem Ansatz anders als in [GK08] die gesamte Schaltplantopologie in den Speicher des CUDA-Geräts passen und so zerlegbar sein muss, dass die Wertematrix jedes Clusters in den 16KB großen *Shared Memory* eines Multiprozessors passt.

6.2 Phase-Shifter-Entwurf für BIST-Applikationen

Heutzutage besitzen viele elektronische Bausteine eine integrierte Schaltung, welche die korrekte Funktion des Bausteins überprüfen kann. Man spricht von *Built In Self Test* (BIST). Dabei erzeugt eine spezielle Schaltung Testsignale, die den Baustein auf sogenannten *Scan-Chains* durchlaufen. Am Ende der Scan-Chain wird das Ergebnis abgegriffen und mit einem gespeicherten Referenzergebnis verglichen. Unterscheiden sich die Ergebnisse, so wurde ein Fehler detektiert. Um einen möglichst zuverlässigen Test zu gewährleisten, sollten die Sequenzen der Testsignale für die Scan-Chains paarweise möglichst unkorreliert sein.

Für die Erzeugung der Testsignale werden häufig *Linear Feedback Shift Registers* (LFSR) eingesetzt. Dabei handelt es sich um ein zyklisches Schieberegister, wobei der Eingang durch eine Linearkombination der übrigen Flipflops gegeben ist. Da die Zustände der übrigen Flipflops aber jeweils dem Zustand des vorgeschalteten Flipflops im vorhergehenden Takt entsprechen, und damit in hohem Maße korreliert sind, können die Scan-Chains nicht direkt mit den Zustandssequenzen der Flipflops betrieben werden. Stattdessen kombiniert für jede Scan-Chain eine Schaltung aus XOR-Gattern die Zustände der Flipflops so, dass die an den Ausgängen dieser XOR-Schaltungen entstehenden Sequenzen möglichst weit

gegeneinander verschoben (man spricht von Phasenverschiebung – *phase shift*) und damit möglichst unkorreliert sind.

Nach [RTTo0] lassen sich solche XOR-Schaltungen berechnen, in dem man den *dualen LFSR* logisch simuliert. Dabei werden in der XOR-Schaltung die Zustände derjenigen Flipflops kombiniert, an deren Index im Zustand des dualen LFSR eine 1 steht. Mit jedem Simulationsschritt wird die von der so definierten XOR-Schaltung erzeugte Sequenz gegenüber der Referenzsequenz (diese entspricht der durch den Zustand des ersten Flipflops gegebenen Sequenz) um eine Position weiter verschoben. Ist eine minimale Phasenverschiebung k gegeben, so erhält man eine XOR-Schaltung, die diese Phasenverschiebung realisiert, indem man den dualen LFSR k Schritte simuliert.

Damit ist prinzipiell ein Vorgehen zur Erstellung einer geeigneten Phase-Shifter-Schaltung gegeben. Da diese Schaltung auf dem Chip jedoch möglichst wenig Platz verbrauchen soll, wird meist eine obere Schranke für die Anzahl der XOR-Gatter pro Scan-Chain festgelegt. Dies bedeutet, dass bei der Simulation des dualen LFSR nur solche Zustände als Grundlage für XOR-Schaltungen erlaubt werden, deren Anzahl von Einsen unterhalb dieser Schranke liegt. Nun wird dieses Vorgehen sehr ineffizient, da Zustände mit wenigen Einsen in der Zustandsfolge des dualen LFSR relativ selten sind.

In [RTTo0] wird deshalb ein anderes Vorgehen vorgeschlagen. Dabei werden nicht alle Zustände des dualen LFSR durchlaufen, sondern für jede Scan-Chain zunächst eine XOR-Schaltung (und damit ein Zustand des dualen LFSRs) anhand einer längenlexikographischen Ordnung erzeugt, der duale LFSR mit diesem Zustand initialisiert und sodann um jeweils k Schritte nach vorne und hinten simuliert, wobei k der geforderten Phasenverschiebung entspricht. Wird bei dieser Simulation ein Zustand erreicht, der bereits für eine XOR-Schaltung einer anderen Scan-Chain verwendet wurde, so ist die Phasenverschiebung der Sequenz zu dieser Scan-Chain zu gering und die XOR-Schaltung kann nicht verwendet werden. In diesem Fall wird mit dem nächsten Element der längenlexikographischen Ordnung fortgefahren, andernfalls wird die Schaltung akzeptiert und mit der Generierung einer XOR-Schaltung für die nächste Scan-Chain begonnen.

Der Hauptaufwand bei diesem Verfahren ist die Simulation des dualen LFSRs, der zudem linear mit der geforderten minimalen Phasenverschiebung zunimmt. Diese Simulation kann aber prinzipiell für verschiedene XOR-Schaltungen parallel durchgeführt werden.

6.2.1 Unser Ansatz

Im Rahmen dieser Fachstudie wurde der in [RTTo0] beschriebene Ansatz zur Generierung von Phase-Shifter-Netzwerken implementiert und die Überprüfung der anhand der längenlexikographischen Ordnung generierten XOR-Schaltungen mittels CUDA auf einer GPU für viele Schaltungen parallel durchgeführt. Das Erzeugen der XOR-Schaltungen entsprechend der längenlexikographischen Ordnung erfolgt weiterhin auf der CPU. Diese Aufteilung liegt darin begründet, dass die Generierung der längenlexikographischen Ordnung seriell erfolgt und sich damit nicht für eine parallele Ausführung auf der GPU eignet. Zudem besteht der

Hauptaufwand ohnehin in der Simulation des dualen LFSR, sodass die serielle Generierung der längenlexikographischen Ordnung auf der CPU nicht ins Gewicht fällt.

Es wird jedem Thread eine XOR-Schaltung zugewiesen, ein Kernel führt die Simulation des entsprechenden dualen LFSR durch und schreibt schließlich, für den Fall dass die XOR-Schaltung akzeptiert wurde, die Anzahl der in die Linearkombination einzubeziehenden Flipflops in ein Ausgabefeld. Wurde die Schaltung abgelehnt, so wird der größtmögliche Wert in das Ausgabefeld geschrieben. Auf der CPU wird nun dieses Ausgabefeld gelesen und festgestellt, welche Threads die ihnen zugewiesene XOR-Schaltung akzeptiert haben. Hat keiner der Threads seine zugewiesene XOR-Schaltung akzeptiert, so werden neue Schaltungen generiert und der Vorgang wiederholt. Andernfalls wird der Thread bestimmt, dessen XOR-Schaltung die geringste Anzahl an einzubeziehenden Flipflops aufweist. Diese Schaltung wird dann für die aktuelle Scan-Chain verwendet und alle abgelehnten Schaltungen durch neu generierte ersetzt, bevor der Vorgang für die nächste Scan-Chain erneut durchgeführt wird.

Um festzustellen, ob eine gegebene XOR-Schaltung eine geforderte minimale Phasenverschiebung gegenüber allen bereits festgelegten Scan-Chains gewährleistet, muss der überprüfende Thread über eine Liste der bereits festgelegten XOR-Schaltungen verfügen (im Folgenden Dictionary genannt). Diese wird im *Texture Memory* der GPU als Array abgelegt, wobei dessen Größe der Anzahl der Scan-Chains entspricht. Nachdem eine Schaltung für eine Scan-Chain akzeptiert wurde, wird diese Schaltung in das Array eingefügt.

Schließlich muss der Thread die Struktur des zugrundeliegenden LFSRs kennen. Diese ist jedoch für alle Threads identisch und wird ebenfalls im *Global Memory* der GPU hinterlegt.

Hier wurde also ein Ansatz gewählt, der nacheinander für jede Scan-Chain eine geeignete XOR-Schaltung generiert. Der Versuch, für mehrere Scan-Chains parallel XOR-Schaltungen zu generieren, würde Kommunikation und Synchronisation zwischen allen gestarteten Threads erfordern, was in der CUDA-Architektur nicht vorgesehen ist. Akzeptiert ein Thread eine XOR-Schaltung für eine Scan-Chain, so müsste er diese in das Dictionary eintragen und alle anderen Threads müssten ihre Simulation mit dem aktualisierten Dictionary neu starten, um falsches Akzeptieren von XOR-Schaltungen mit zu geringer Phasenverschiebung zu der zuletzt akzeptierten Schaltung zu verhindern.

7 Fazit

In dieser Fachstudie wurden die grundlegenden Konzepte von GP-GPUs und den entsprechenden Programmierumgebungen vorgestellt und deren Bedeutung für den Bereich der *Electronic Design Automation* beziehungsweise des *Computer Aided Design* aufgezeigt. Die Beispiele zeigen, dass auch der EDA/CAD-Bereich von der enormen Leistung von GP-GPUs profitieren kann, gerade weil viele Algorithmen im EDA/CAD-Bereich großes Parallelisierungspotential besitzen. Es wurde aber auch deutlich, dass nicht jede Parallelisierung eines Algorithmus für GPGPU geeignet ist. So sollten beispielsweise die ausgeführten Instruktionen möglichst genau dem SIMD-Paradigma entsprechen und Speicherzugriffe auf die Fähigkeiten der gewählten Zielhardware angepasst sein, um das volle Potential von GPGPU auszuschöpfen. Werden diese Aspekte nicht beachtet, ist es unwahrscheinlich, dass die Performanz des so abgebildeten Algorithmus die eines optimierten CPU-Codes übersteigt.

Literaturverzeichnis

[Adv09] Advanced Micro Devices, Inc.: *ATI Stream Computing – Technical Overview*, 2009. http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf (Zitiert auf Seite 26)

[CDB09] Chatterjee, Debapriya ; DeOrio, Andrew ; Bertacco, Valeria: GCS: High-performance gate-level simulation with GPGPUs. In: *DATE*, IEEE, 2009, S. 1332–1337 (Zitiert auf den Seiten 55 und 56)

[Cor09] Corporation, Intel: Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series Based on Intel Microarchitecture (Nehalem). (2009). download.intel.com/products/processor/corei7/319724.pdf (Zitiert auf Seite 14)

[Fly72] Flynn, Michael J.: Some Computer Organizations and Their Effectiveness. In: *Computers, IEEE Transactions on* C-21 (1972), Sept., Nr. 9, S. 948–960. http://dx.doi.org/10.1109/TC.1972.5009071. – DOI 10.1109/TC.1972.5009071. – ISSN 0018–9340 (Zitiert auf Seite 11)

[GK08] Gulati, Kanupriya ; Khatri, Sunil P.: Towards acceleration of fault simulation using graphics processing units. In: Fix, Limor (Hrsg.): *DAC*, ACM, 2008. – ISBN 978–1–60558–115–6, S. 822–827 (Zitiert auf den Seiten 40, 52, 53, 54, 55 und 56)

[Haa93] Haan, Oswald: *Vektorrechner: Architektur – Programmierung – Anwendung*. 1. Ausgabe. Sauer, 1993 (Zitiert auf Seite 16)

[Khr08] Khronos OpenCL Working Group: *The OpenCL Specification, version 1.0.29*, 8 December 2008. http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf (Zitiert auf Seite 35)

[NVI08] NVIDIA: *NVIDIA CUDA Programming Guide 2.0*. 2008 (Zitiert auf den Seiten 32, 42 und 53)

[Pol99] Pollack, Fred J.: New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In: *MICRO*, 1999, S. 2– (Zitiert auf Seite 14)

[RTT00] Rajski, Janusz ; Tamarapalli, Nagesh ; Tyszer, Jerzy: Automated synthesis of phase shifters for built-in self-testapplications. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 19 (2000), Nr. 10, S. 1175–1188 (Zitiert auf Seite 57)

Alle URLs wurden zuletzt am 30.11.2009 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

_____

(Matthias Müller, Jochen Puff, Mark Silberberger)