

Institut für Technische Informatik
Universität Stuttgart
Pfaffenwaldring 47
D-70569 Stuttgart

Studienarbeit Nr. 2226

Parallele Fehlersimulation auf GPGPUs

Marcel Schaal

Studiengang:	Dipl. Informatik
Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl. Ing. Christian Zöllin Dipl. Inf. Michael Kochte
begonnen am:	09. Juni 2009
beendet am:	09. Dezember 2009
CR-Klassifikation:	B.7.1, B.8.1, C.1.4, C.4, D.1.3, J.6

Inhaltsverzeichnis

1	Einleitung	5
2	Fehlersimulation für kombinatorische Schaltungen	7
2.1	Schaltungsgraph	7
2.2	Verzweigungsstamm	7
2.3	Fehlermodell	8
2.4	Tabellengesteuerte Simulation	10
2.5	Fehlersimulation	10
2.6	Serielle Fehlersimulation	10
2.7	Fehleräquivalenz	10
2.8	Gatteraufgabe	12
3	Parallel-Pattern-Single-Fault-Propagation-Algorithmus	13
3.1	Parallele Signalauswertung	15
3.2	Fehlersimulation in verzweigungsfreien Gebieten	15
3.3	Fehleraufgabe	17
3.4	Ereignisgesteuerte Simulation	17
3.5	Dominatoren	18
4	Atalanta	19
4.1	Funktionsweise	19
4.2	Profiling	21
5	NVIDIA Compute Unified Device Architecture	27
5.1	Funktionsweise	27
5.2	Speicherhierarchie	28
5.3	Effiziente Nutzung der Architektur	30
5.4	Eignung von GPGPUs für parallele Fehlersimulation	31
5.5	Werkzeuge	34
6	Parallele Implementierung mit CUDA	37
6.1	Gutsimulation und Fehleraufgabe	39
6.2	Verzweigungsstammsimulation	39

6.3	Fehlerauswertung	45
6.4	Scheduling	46
7	Auswertung	47
7.1	Bisherige Ansätze	47
7.2	Verzweigungsstammsimulation	48
7.3	Beobachtbarkeit und Fehlererkennung	49
7.4	Zusammenfassung	49
8	Zusammenfassung	55
A	Ereignisgesteuerte Methode	57
A.1	Parallelisierung	57
A.2	Kantensimulation	58
A.3	Verwaltung von Ereignissen	61
A.4	Beobachtbarkeit	62
A.5	Gutwerte wiederherstellen	62
B	Approximative Fehlersimulation	63
B.1	Funktionsweise	63
B.2	Auswertung	65
C	Literaturverzeichnis	69
D	Abbildungsverzeichnis	71
E	Tabellenverzeichnis	73

Einleitung

Bei der Produktion von Integrierten Schaltungen (engl. integrated circuits, IC) kann das Auftreten von strukturellen Störungen im Halbleiter, so genannte Defekte, nicht völlig ausgeschlossen werden. Je nach Defekt kann dieser Einfluss auf die Funktion der Schaltung haben oder nicht. Die Produktqualität kann gesteigert werden durch Aussortieren von fehlerhaften ICs vor der Auslieferung. Um defekte ICs zu finden, werden diese mit Hilfe von Testmustern überprüft. Da Testen für den Hersteller Kosten verursacht und keinen direkten Gewinn erwirtschaftet, muss es möglichst effizient durchgeführt werden. Daher sollte ein Test möglichst effizient sein, d.h. mit wenigen Testmustern möglichst viele Fehler entdecken. Um diese Testmuster zu generieren und ihre Fehlererfassung (engl. Fault coverage) zu ermitteln, wird Fehlersimulation benutzt.

Als weiteren Anwendungsfall lässt sich Fehlersimulation zur Überprüfung von Fehlertoleranztechniken in einem IC nutzen. Wird versucht einen IC durch Redundanz fehlertolerant zu machen, so muss sicher gestellt werden, dass die Fehlertoleranz tatsächlich effektiv ist.

Eine grobe Abschätzung der Komplexität für kombinatorische Fehlersimulation ist nach [WWWo6] $O(p * n^2)$, wobei p die Anzahl der Testmuster und n die Größe der Schaltung darstellt. Allerdings lässt sich die Fehlersimulation durch effizientere Algorithmen [BHK92] beschleunigen.

Die kombinatorische Fehlersimulation ist inhärent parallel. Jede Schaltung kann für jedes Testmuster und für jeden Fehler parallel ausgewertet werden. Auch die Auswertung des Schaltungsgraphen kann parallelisiert werden, allerdings ist auf die abhängige Auswertung von adjazenten Gatterknoten zu achten.

General-Purpose-Graphical-Processing-Units (GPGPUs) bestehen aus sehr vielen Rechenwerken mit wenig lokalem Speicher. Dies steht im Gegensatz zu herkömmlichen CPUs, welche aus relativ wenigen Rechenwerken aufgebaut, dafür aber mit sehr viel Speicher in Form von Cache-Hierarchien ausgestattet sind. Durch die massive Parallelität einer GPGPU und der inhärenten Parallelität der Fehlersimulation, bietet es sich an, die Fehlersimulation mit Hilfe von GPGPUs zu beschleunigen.

Das Ziel dieser Studienarbeit ist die Beschleunigung der Fehlersimulation für kombinatorische Schaltungen mit Haftfehlern mittels GPGPUs. Dafür wird ein bestehender Fehlersimulator analysiert und rechenintensive Teile beschleunigt. Diese sollen durch Ausnutzung der Datenstrukturen und Algorithmen parallel ausgeführt und somit eine Leistungssteigerung erreicht werden.

Die vorliegende Ausarbeitung beschreibt zunächst die grundlegenden Methoden der Fehlersimulation und der verwendeten Algorithmen und Datenstrukturen. Anschließend wird kurz der zu optimierende Fehlersimulator beschrieben. Danach folgt eine Beschreibung der Zielplattform NVIDIA CUDA sowie die Implementierung der parallelen Fehlersimulation und es folgt eine Auswertung der erzielten Ergebnisse. Den Schluss bildet ein Ausblick, wie diese Arbeit fortgeführt werden könnte.

Fehlersimulation für kombinatorische Schaltungen

Im folgenden Kapitel werden Begriffe im Zusammenhang mit der Fehlersimulation definiert. Unter Anderem findet sich darin die Erklärung des Fehlermodells, welches eine Abstraktion des eigentlichen physikalischen Defekts auf einen logischen Fehler bildet. Die Fehlersimulation überprüft einen Schaltungsgraphen mit Hilfe einer Menge von Testmustern und einer Fehlermenge. Der Anteil der dabei entdeckten Fehlern gibt die Fehlerabdeckung an. Bei der seriellen Fehlersimulation handelt es sich um eine unmittelbare Implementierung der Fehlersimulation ohne algorithmische Optimierungen. Durch die Einteilung der Fehler, die identische Auswirkung auf die Schaltung für die gegebene Testmuster Menge haben, in Fehleräquivalenzklassen lässt sich die Menge der zu testenden Fehler reduzieren.

Dieses Kapitel soll lediglich die Definitionen abstrakt wiedergeben. Für eine genaue Definition wird auf die entsprechende Fachliteratur verwiesen, wie z.B. [Sch88], [BA05].

2.1 Schaltungsgraph

Ein Schaltungsgraph $G := (V, E)$ ist ein gerichteter Graph, wobei V die Gatterknoten und $E \in V \times V$ die Signale darstellen.

2.2 Verzweigungsstamm

Ein Verzweigungsstamm ist ein Gatter $v \in V$, welches mehr als einen direkten Nachfolger besitzt: $|\{s \in V : (v, s) \in E\}| \geq 2$.

2.3 Fehlermodell

Ein Fehlermodell abstrahiert vom physikalischen Defekt zum abstrakten Fehler. Durch die Abstraktion des Fehlermodells werden manche Defekte nicht abgebildet. Als Fehlermodell wird hier das einfache Haftfehlermodell angenommen. Konkret heißt das, dass in einem Schaltungsgraphen genau eine Ein- oder Ausgangskante fälschlicherweise einen konstanten Boole'schen Wert propagiert. Durch diese Abstraktion werden unter anderem Mehrfachfehler, wie sie in der realen Welt auftreten können, nicht betrachtet.

Für einen Knoten mit zwei Eingangs- und einer Ausgangskante gibt es daher genau sechs mögliche Fehler. Die beiden Eingangskanten können jeweils einen Haftfehler nach Null (SA0) bzw. Eins (SA1) und die Ausgangskante kann ebenfalls einen SA0- oder SA1-Fehler aufweisen. Der SA0-Fehler wurde in 2.1 graphisch dargestellt. Für einen Knoten mit nur einer Eingangskante gibt es entsprechend vier Fehler. Die möglichen Fehler werden Abbildung 2.2 dargestellt.

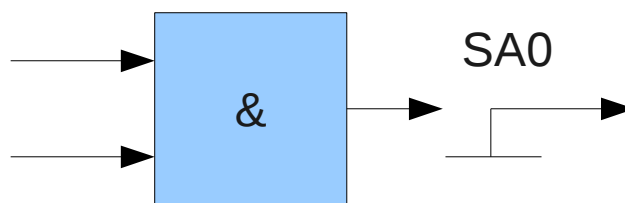


Abbildung 2.1: Stuck-At-0 Fehler am Ausgang eines Und-Gatters

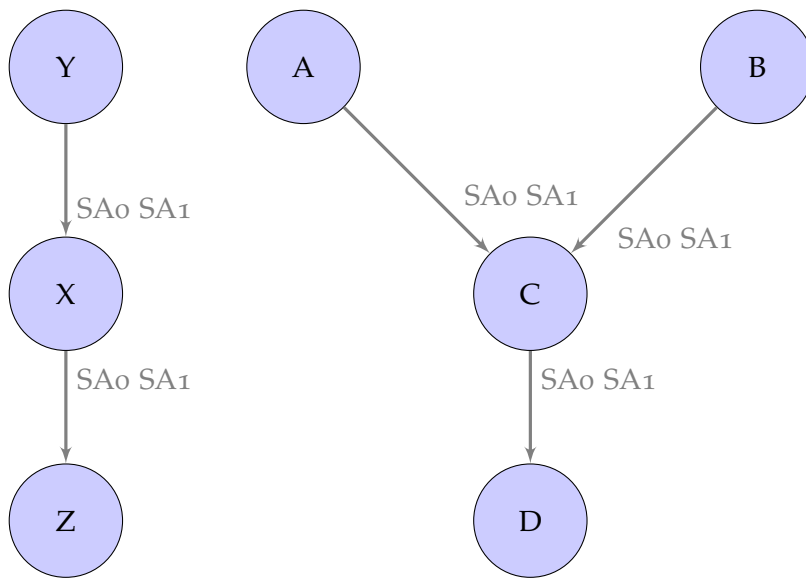


Abbildung 2.2: Beispiel zum Fehlermodell

2.4 Tabellengesteuerte Simulation

Bei der tabellengesteuerten Simulation werden die Funktionen eines Schaltungsgraphen in einer Datenstruktur kodiert. Die Auswertung eines Gatterknotens erfolgt durch Nachschlagen in der zugehörigen Datenstruktur und durch ausführen der entsprechenden Funktion. Ein anderer Ansatz ist die compiler-gesteuerte Simulation. Hierbei wird der Schaltungsgraph vor der Ausführung in eine Programmiersprache übersetzt, welche später als Maschinencode direkt vom Prozessor ausgeführt wird. Dadurch entfallen Zugriffe und Auswertung der entsprechenden Datenstruktur.

2.5 Fehlersimulation

Gegeben sei ein Schaltungsgraph G mit einer Menge von Fehlern F und einer Menge von Testmustern P . Die Fehlerabdeckung C gibt die Güte der Testmuster für den entsprechenden Schaltungsgraph unter der gegebenen Fehlermenge an. Sie ist definiert als das Verhältnis von erkannten Fehlern zu deren Gesamtzahl, d.h. $C := \frac{|F_{detected}|}{|F|}$.

Die Fehlersimulation ist das Überprüfen eines Schaltungsgraphen mittels gegebenen Fehler- und Testmuster Mengen. Aus dem Ergebnis der Fehlersimulation lässt sich unter Anderem die Fehlerabdeckung ableiten.

2.6 Serielle Fehlersimulation

Unter der seriellen Fehlersimulation wird eine naive Implementierung ohne algorithmische Optimierung der Fehlersimulation verstanden. Für jeden Fehler und jedes Muster wird eine Logiksimulation ohne Fehler, die so genannte Gutsimulation, und eine Logiksimulation mit injiziertem Fehler, eine explizite Fehlersimulation, ausgewertet. Anschließend werden die Ausgangsvektoren beider Simulationen verglichen. Existiert eine Differenz, so ist der Fehler durch das aktuelle Testmuster erkannt worden. Falls es keine Differenz gibt, so wird mit dem nächsten Testmuster fortgesetzt, bis entweder dieser Fehler erkannt wird oder alle Testmuster abgearbeitet wurden.

2.7 Fehleräquivalenz

In einem Schaltungsgraphen existieren viele Fehler, welche sich in äquivalente Fehlerklassen einteilen lassen. Fehler in einer Äquivalenzklasse verhalten sich für jedes Testmuster exakt gleich. Wird z.B. ein AND-Schaltungsknoten betrachtet, so hat der SAo-Fehler an einer der Eingangskanten dieselbe Auswirkung wie der SAo-Fehler an der Ausgangskante des AND-Schaltungsknoten. In Abbildung 2.3 gibt es für den zugehörige AND-Schaltungsknoten folgende Fehlerklassen: {SAo-A, SAo-B, SAo-D}, {SA1-A}, {SA1-B}, {SA1-D}

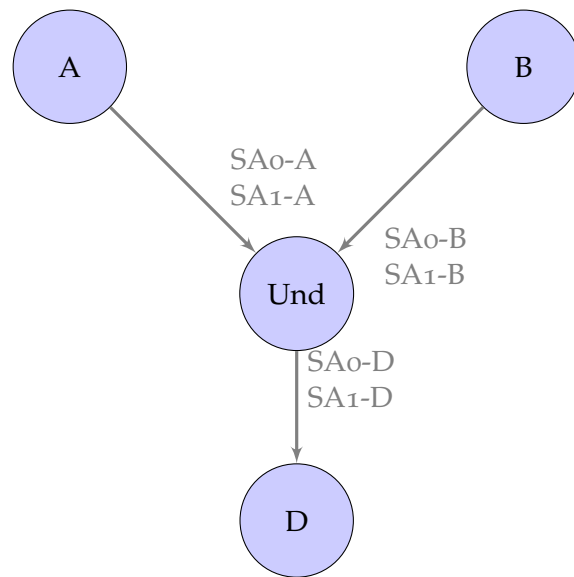


Abbildung 2.3: Beispiel zur Fehleräquivalenz

Das Gleiche gilt für einen NAND-Schaltungsknoten, allerdings mit einem SA₁-Fehler. Für Inverter- oder Puffergatterknoten entsprechen die Eingangsfehler den Ausgangsfehlern. Laut [BA05] ergibt sich dadurch eine Reduktion der zu simulierenden Einzelfehler um 50 bis 60%. Daher sollte die Bestimmung der Fehleräquivalenz in jedem Fall vor der Fehlersimulation Anwendung finden.

2.8 Gatteraufgabe

Die Gatteraufgabe (engl. fault dropping) reduziert die Zahl der auszuwertenden Gatterknoten. Sind alle Fehler eines Gatterknoten G im Schaltungsgraphen erkannt worden, so kann auf eine weitere Betrachtung des Knotens G durch die Fehlersimulation verzichtet werden.

Parallel-Pattern-Single-Fault-Propagation-Algorithmus

Im folgenden Kapitel wird der Parallel-Pattern-Single-Fault-Propagation-Algorithmus (PPSFP)[WEF⁺85] beschrieben. PPSFP basiert auf der parallelen Auswertung mehrerer Muster in einem Maschinenwort, wobei nur ein einzelner Fehler injiziert wird. Es werden nur kombinatorische Schaltungen betrachtet, wodurch sich weitere Optimierungen ergeben.

Eine Optimierung befasst sich mit der Reduktion der explizit zu simulierenden Fehler. Durch die Ausnutzung von verzweigungsfreien Gebieten wird die explizite Fehlersimulation auf Verzweigungsstämme beschränkt.

Weiter kann die Zahl der Fehlersimulationen durch Fehleraufgabe reduziert werden. Diese bezieht die lokale Aktivierung der Fehler mit ein. Sind in einem verzweigungsfreien Gebiet keine Fehler sensibilisierbar, so ist es nicht notwendig den dazugehörigen Verzweigungsstamm zu simulieren.

Die Zahl der nötigen Auswertungen kann durch die ereignisgesteuerte Simulation verringert werden. Bei dieser werden nur Gatterknoten ausgewertet, bei denen mindestens an einem der Vorgängerknoten eine Werteänderung durch die Fehlerinjektion statt gefunden hat.

In Abbildung 3.1 findet sich das Ablaufdiagramm des PPSFP-Algorithmus. Nach dem Einlesen und Initialisieren der Datenstrukturen, werden Testmuster erzeugt und eine Logiksimulation durchgeführt. Anschließend erfolgt die Fehlersimulation und die Auswertung der Beobachtbarkeit sowie der Fehlerentdeckung.

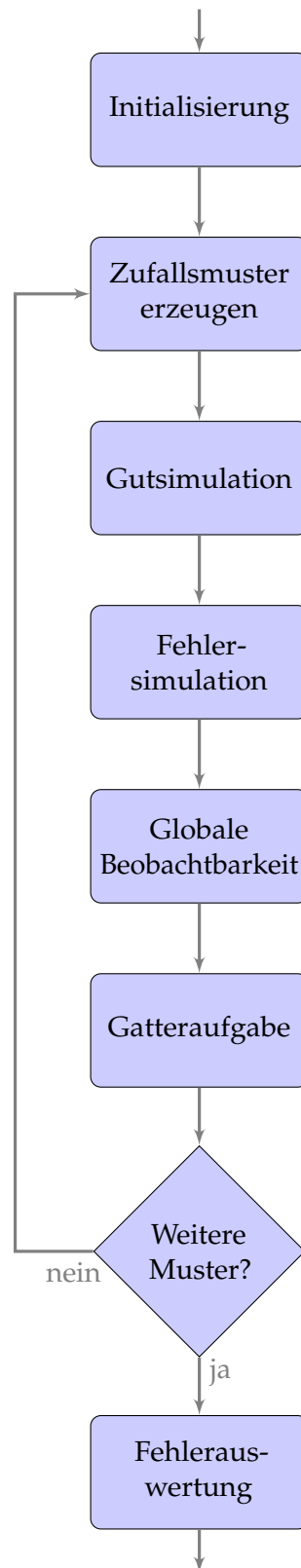


Abbildung 3.1: Ablaufdiagramm des PPFSP-Algorithmus

3.1 Parallele Signalauswertung

Bei der parallelen Signalauswertung (engl. parallel pattern) werden w Testmuster in ein einzelnes Maschinenwort der Breite w kodiert. Dadurch können w Testmuster, mit sehr geringem Mehraufwand, gleichzeitig simuliert werden. Dies ist ein Ansatz um die Parallelität in der Fehlersimulation bei zweiwertiger Logik auszunutzen. Durch Nutzung mehrerer Datenwörter oder spezieller Kodierungsschemen, lässt sich dies auch auf andere Logik ausweiten.

3.2 Fehlersimulation in verzweigungsfreien Gebieten

Statt jeden Fehler einzeln zu simulieren ist es ausreichend, lediglich die Verzweigungsstämme selbst zu simulieren und aus der Beobachtbarkeit auf die Fehler in den verzweigungsfreien Gebieten (engl. Fanout-Free-Region, FFR) zu schließen. Dafür werden die Fehler durch Invertierung des Gutwerts am Ausgang des Verzweigungsstammes injiziert. Danach wird eine explizite Fehlersimulation durchgeführt. Mit Hilfe der Beobachtbarkeit an den Verzweigungsstämmen wird nun der Graph rückwärts traversiert und in den verzweigungsfreien Gebieten der Verzweigungsstämme die noch fehlenden Beobachtbarkeiten festgestellt.

Die Beobachtbarkeiten $O \in \{0, 1\}$ in den verzweigungsfreien Gebieten lassen sich mit den Formeln aus Tabelle 3.2 rekursiv berechnen. Aus der Beobachtbarkeit des Nachfolgergatterknotens, welche hier als O_o bezeichnet wird kann bei Gatterknoten mit nur einem Eingang direkt auf die Beobachtbarkeit O_i geschlossen werden. Bei Gatterknoten mit mehreren Eingängen $i1, i2$ müssen für die Berechnung der Beobachtbarkeit eines Vorgängers O_{i1}, O_{i2} die Gutwerte $GV \in \{0, 1\}$ des anderen Vorgängerknotens GV_{i2}, GV_{i1} mit einbezogen werden.

Nun lässt sich anhand der Beobachtbarkeit an einem Gatterknoten ablesen, ob der Fehler erkennbar ist. Dem ist so, falls er durch ein Testmuster am Schaltungsausgang beobachtbar und durch das angelegte Testmuster aktivierbar ist. Dies wird anhand eines Beispiels in Abbildung 3.2 verdeutlicht. Dieses Schema wird Gebietsanalyse (engl. Critical-Path-Tracing) genannt und wurde in [AMM83] vorgestellt.

not	O_i	$= O_o$
and	O_{i1}	$= O_o \wedge GV_{i2}$
	O_{i2}	$= O_o \wedge GV_{i1}$
or	O_{i1}	$= O_o \wedge \neg GV_{i2}$
	O_{i2}	$= O_o \wedge \neg GV_{i1}$
xor	O_{i1}	$= O_o$
	O_{i2}	$= O_o$
PO	O_o	$= 1$

Tabelle 3.1: Rekursives Berechnungsschema für die Beobachtbarkeit

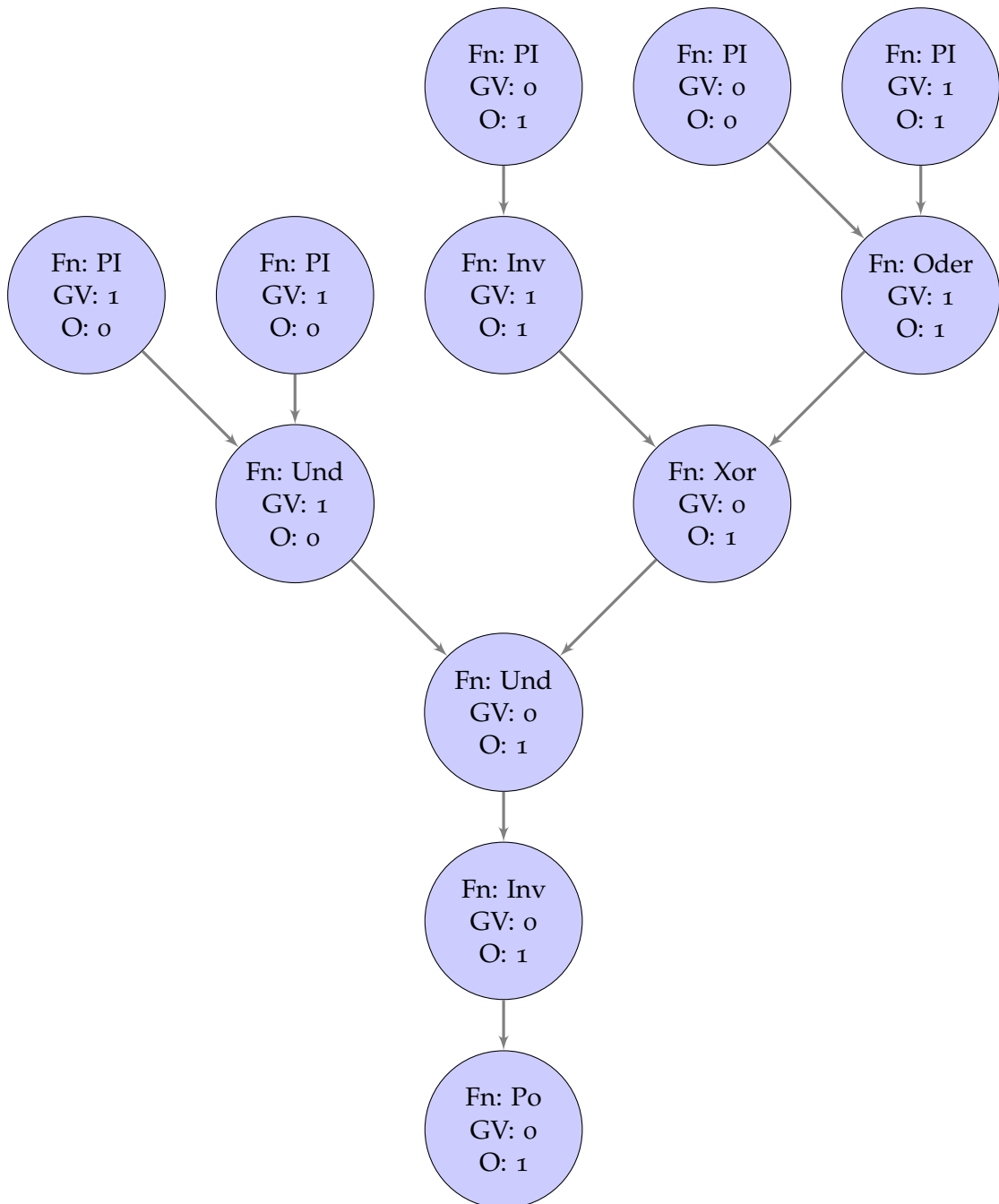


Abbildung 3.2: Beispiel der Beobachtbarkeit einer Schaltung mit wenigen Gattern.

3.3 Fehleraufgabe

Wenn ein Fehler erkannt wurde, so muss für die Fehlersimulation dieser Fehler nicht weiter betrachtet werden. Werden Zufallsmuster benutzt, so können bereits nach wenigen Mustern sehr viele Fehler erkannt worden sein, wodurch ein großer Teil der Simulationszeit eingespart werden kann.

Fehleraufgabe beschleunigt die serielle Fehlersimulation erheblich, denn jeder Fehler muss zuerst explizit simuliert werden. Da beim PPSFP aus den Beobachtbarkeiten der Verzweigungsstämme auf die Beobachtbarkeit der einzelnen Fehler geschlossen wird, können auch hier die Fehler aus der Fehlerliste entfernt werden. Jedoch erst, wenn in der verzweigungsfreien Region eines Verzweigungsstamms keine unentdeckten Fehler mehr vorhanden sind, kann der Verzweigungsstamm von der weiteren expliziten Fehlersimulation ausgeschlossen werden.

Eine weitere Beschleunigung wird durch die Einbeziehung der lokalen Aktivierung der Fehler selbst erreicht. Können die verbleibenden unentdeckten Fehler in der verzweigungsfreien Region nicht durch das momentane Testmuster lokal sensibilisiert werden, so ist es unnötig, die explizite Fehlersimulation für diesen Verzweigungsstamm durchzuführen, denn der Fehler wird nicht sensibilisiert.

3.4 Ereignisgesteuerte Simulation

Bei der Fehlersimulation kann die Auswertung der Gatterknoten auf diejenigen beschränkt werden, an denen tatsächlich eine Werteänderung an den Eingängen stattgefunden hat. Nur unter dieser Bedingung kann sich auch der Wert am Ausgang ändern. Ohne die ereignisgesteuerte Simulation müssen im besten Fall mindestens der Ausgangskegel des injizierten Fehlers ausgewertet werden.

Eine vollständige Simulation des Ausgangskegels kann zu Beginn der Fehlersimulation effizienter sein, wenn viele Fehler noch unentdeckt sind und dadurch sehr viele Kegel tatsächlich bis zu den Ausgängen simuliert werden müssen. In solchen Fällen trägt die Verwaltung der Datenstrukturen für die ereignisgesteuerte Simulation deutlich zur Laufzeit bei.

Darüber hinaus wirkt die musterparallele Methode für große w der Effizienz der ereignisgesteuerten Simulation entgegen. Mit zunehmender Zahl parallel ausgewerteter Muster, steigt die Wahrscheinlichkeit, dass ein Gatterknoten sensibilisiert wird und einen Wert propagiert. Speziell bei Zufallsmustern ist die Wahrscheinlichkeit hoch, dass viele Gatter sensibilisiert werden und fast alle Gatterknoten propagieren, so dass die Verwaltung der Datenstrukturen einen großen Mehraufwand zur Folge haben kann.

3.5 Dominatoren

Ein Knoten D dominiert einen Knoten G , wenn alle Ausgangspfade von G durch D führen. Dies lässt sich Ausnutzen, da die Beobachtbarkeit von G durch D bestimmt werden kann. Dadurch kann die explizite Simulation von Verzweigungstämmen auf die explizite Simulation des Dominators beschränkt werden. Weiteres kann in [HSU86], [MR90] nachgelesen werden.

Atalanta

Atalanta [LH91] ist ein Werkzeug zur automatischen Erzeugung von Testmustern (engl. Automatic Test Pattern Generation (ATPG)). Darüber hinaus enthält Atalanta einen eigenen Fehlersimulator, welcher zur Bewertung der erzeugten Testmuster genutzt werden kann. In diesem Kapitel wird dessen Implementierung erläutert sowie ein Profiling ausgewertet.

Implementiert wurde der Atalanta Fehlersimulator in der Programmiersprache C. Er steht zur wissenschaftlichen Nutzung frei zur Verfügung. Der Quelltext¹ kann heruntergeladen werden, so dass eigene Modifikationen vorgenommen werden können. Entwickelt wurde Atalanta an der „Virginia Polytechnic Institute and State University“ von H. K. Lee und Dr. Dong S. Ha.

4.1 Funktionsweise

Der Atalanta Fehlersimulator implementiert den PPSFP-Algorithmus, welcher im Kapitel 3 beschrieben wurde, in einer tabellengesteuerten Simulation. An dieser Stelle soll auf einige Implementierungsdetails eingegangen werden.

Einen groben Überblick aller Funktionen liefert der Funktionsaufrufgraph aus Abbildung 4.1. Wichtig sind hierbei die folgenden Methoden, welche dazu genauer erläutert werden:

read_circuit: Read_circuit liest die Netzliste aus einer entsprechenden Datei ein. Dabei werden sehr viele kleine Datenstrukturen allokiert, welche zuerst in einer verketteten Liste gespeichert werden. Anschließend wird aus dieser verketteten Liste das eigentliche topologisch sortierte Feld der Netzliste erzeugt. Durch die Nutzung der verketteten Listen wird ein doppeltes Einlesen der Schaltungsinformationen vermieden. Atalanta kann ISCAS85 und ISCAS89 Netzlisten verarbeiten. Das Einlesen der Schaltung kann bei großen Schaltungen und wenigen zu testenden Testmustern den größten Zeitanteil einnehmen.

¹<http://www.cesca.centers.vt.edu/tutorial/downloadTools.php>

pfault_free_simulation: Die Gutsimulation wird in jedem Zyklus zu Beginn aufgerufen. Hierbei wird das topologisch sortierte Netzlistenfeld vorwärts traversiert und jeder Gatterknoten ausgewertet.

Fault1_Simulation: Umfasst die eigentliche Verzweigungsstammsimulation sowie die globale Beobachtbarkeit und die Erkennung der Fehler. Dabei werden Dominatoreninformationen mit betrachtet. Fault1_Simulation ruft die Funktionen ftp_reverse und pfault_simulation auf.

ftp_reverse: Berechnet die lokale Sensibilisierung mit Hilfe der Gebietsanalyse. Dabei wird der Schaltungsgraph rückwärts traversiert wie in Abschnitt 3.2 erläutert.

pfault_simulation: Die eigentliche Fehlersimulation, welche bis zum Dominator oder zu den Primärausgängen simuliert und benutzt dabei eine ereignisgesteuerte Simulation.

restore_fault_free_value: Nach der Fehlersimulation müssen die eigentlichen Gutwerte wieder hergestellt werden. Die Gut- und Fehlerwerte werden bei Atalanta in der Gatterknotendatenstruktur gespeichert und werden speicherlokal kopiert.

update_all: Die Gatterknoten- und Fehleraufgabe sind in der update_all-Funktion implementiert. Hierbei wird geprüft, ob alle Fehler eines Gatterknotens bereits evaluiert wurden und falls ja, wird dieser Gatterknoten von der weiteren Simulation ausgeschlossen. Die update_all-Funktion kann auch ganze Partitionen der Schaltung vom Simulationszyklus ausnehmen, wenn darin alle Fehler evaluiert wurden. Dies entspricht im wesentlichen Abschnitt 3.3, geht durch Einbeziehen der Partionen aber noch einen Schritt weiter.

Die wichtigste Datenstruktur in Atalanta ist die Beschreibung eines Gatterknotens. In dieser finden sich zum Einen die notwendigen Informationen für die Simulation, d.h. direkte Vorgänger- und Nachfolgerknoten, als auch der Gut- und Falschwert und zum Anderen Informationen, die für den Aufbau der Schaltung notwendig sind. Die Gatterknoten werden einzeln auf der Halde gespeichert. Es existiert ein Feld, in dem Zeiger auf die einzelnen Gatterknoten gespeichert werden. Dieses lässt sich direkt mit einem während dem Schaltungsaufbau erzeugten Identifikator adressieren.

Eine weitere wichtige Datenstruktur zur Verwaltung aller Ereignisse ist der Stapelspeicher. Diese Datenstruktur besteht im Atalanta Fehlersimulator aus einem Zähler für die Zahl der gespeicherten Gatterknoten und einem Zeiger auf ein Feld von Gatterknotenzeigern. Die Beschreibung der Datenstruktur findet sich in Programmauflistung 4.1.

Listing 4.1 Datenstruktur eines Stapelspeichers in Atalanta

```
1 typedef struct STACK { /* LIFO stack */
2   int last; // Letzter Eintrag im Stapelspeicher
3   struct GATE **list; // Zeiger auf das Feld
4 };
```

Die ereignisgesteuerte Simulation benötigt zur Verwaltung der Ereignisse eine entsprechende Datenstruktur. Diese muss es ermöglichen, Elemente nach der Priorität ihres Rang zu entnehmen. Eine solchen Datenstruktur ist z.B. die Prioritätswarteschlange. Da die Prioritäten bei der kombinatorischen Fehlersimulation diskret sind, kann für jeden Rang ein eigener

Stapelspeicher genutzt werden. Auf diese Weise kann die Operationen „in die Prioritätswarteschlange legen“ mit einem konstanten Aufwand $O(1)$ ausgeführt werden. Jeder Stapel wird bei der Initialisierung groß genug allokiert, um hinreichend viele Gatterknoten zu verwalten.

Es existieren weitere Stapelspeicher zur Verwaltung der noch zu simulierenden Verzweigungsstämme sowie Gatterknoten für die Gutsimulation.

4.2 Profiling

Soll ein Programm beschleunigt werden, so sollte zuerst untersucht werden, welche Teile besonders rechenaufwändig sind. Dies lässt sich mit unterschiedlichen Methoden bewerkstelligen. Eine Möglichkeit ist die Auswertung der Performance-Counter, welche in heutigen CPUs vorhanden sind. Diese Performance-Counter werden genutzt um die Anzahl bestimmter Ereignisse, wie z.B. L1-Cache-Misses oder vergangene CPU-Takte, während der Ausführung eines Programms oder Systems zu bestimmen. Je nach Prozessor und Qualität des Profilingwerkzeugs kann die Aufschlüsselung dabei sehr fein und genau sein.

Ein gutes Profilingwerkzeug bietet die Möglichkeit sehr genau zuzuordnen, wo ein Ereignis im Quellcode des Programms auftrat. Dies lässt sich ausnutzen, um ein Programm auf eine spezifische Prozessorarchitektur zu optimieren oder auch um allgemeine Leistungsengpässe zu finden. Um einen groben Überblick zu erhalten, wurden für zehn Benchmark-Schaltungen ein Profiling erzeugt. Die Schaltungen sind aufsteigend sortiert. Die Zahl der Gatter wächst in den Abbildungen 4.2, 4.3 und 4.4 von Schaltung zu Schaltung, von links nach rechts ungefähr um den Faktor 1,5. Dabei werden die fünf Funktionen des Simulationszyklus betrachtet, in denen die meisten CPU-Takte ausgeführt wurden.

In Abbildung 4.2 werden die Funktionen in Abhängigkeit ihrer anteilig verbrauchten CPU-Zyklen dargestellt. Wie zu erwarten, benötigt die allgemeine Verzweigungsstammsimulation den größten Teil der Rechenzeit. Darauf folgt die Auswerteroutine für Fehler- und Gatterknotenaufgabe. In dieser konkreten Implementierung werden die Gatterknoten vollständig aus dem Scheduling herausgenommen, d.h. Teile der Schaltung werden auch nicht mehr in der Gutsimulation betrachtet. Diese beiden Funktionen benötigen zusammen bereits teilweise 90% des Rechenaufwandes.

Als nächstes wurden in Abbildung 4.3 die L1-Daten-Cache-Miss-Rates dargestellt. Erst für größere Schaltungen ab „p89k“ lässt sich ein Trend ableiten. Die ereignisgesteuerten Funktionen „pfault_simulation“, „ftp_reverse“ und „restore_fault_free_values“ haben eine niedrigere Miss-Rate als die sequentiell arbeitenden Funktion „pfault_free_simulation“ und „update_all“.

In Abbildung 4.4 sind die L2-Cache-Miss-Rates dargestellt. Es zeigt sich der gleiche Trend wie in 4.3. Die ereignisgesteuerten Funktionen haben eine geringere Miss-Rate als die sequentiell arbeitenden.

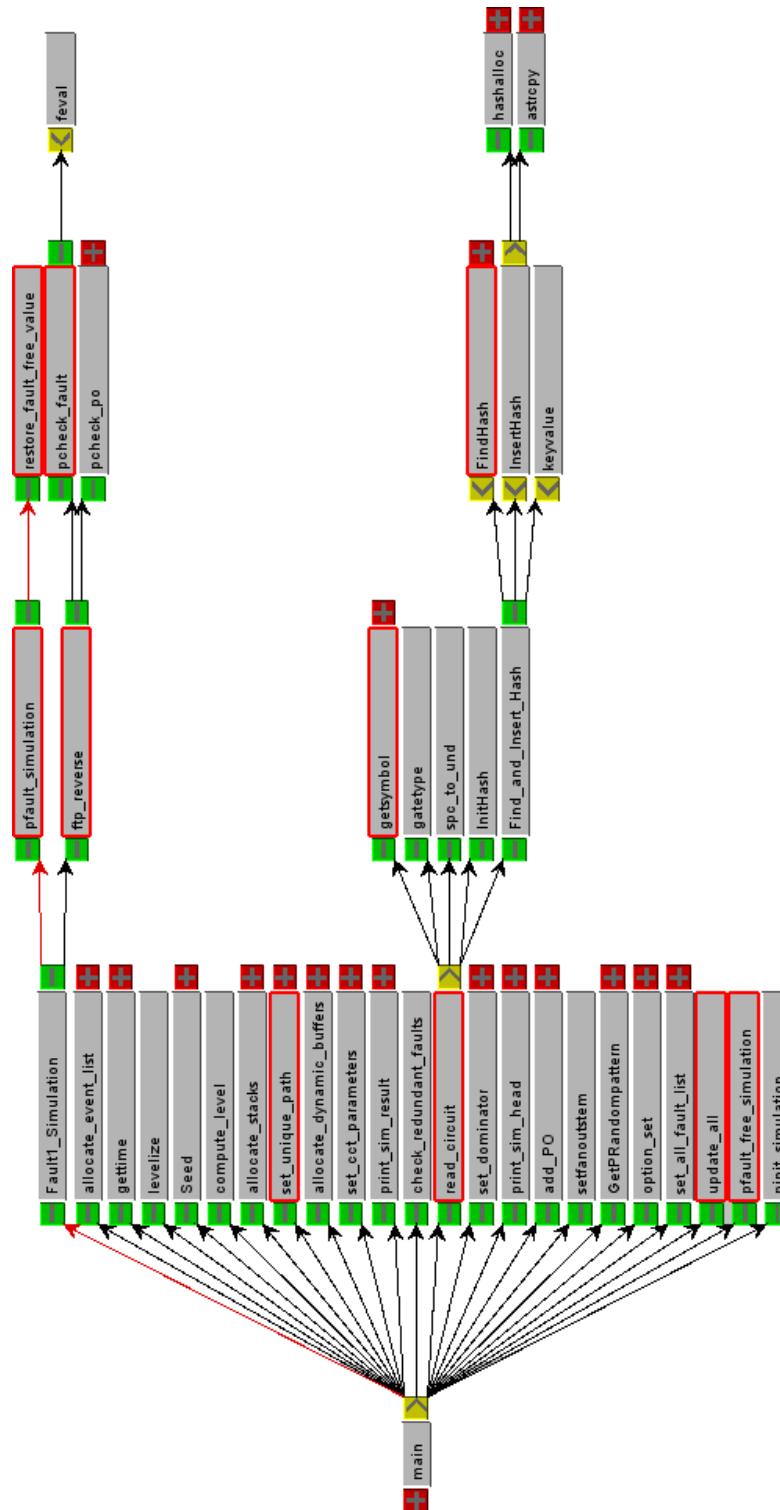


Abbildung 4.1: Aufrufgraph – Atalanta Fehlersimulator

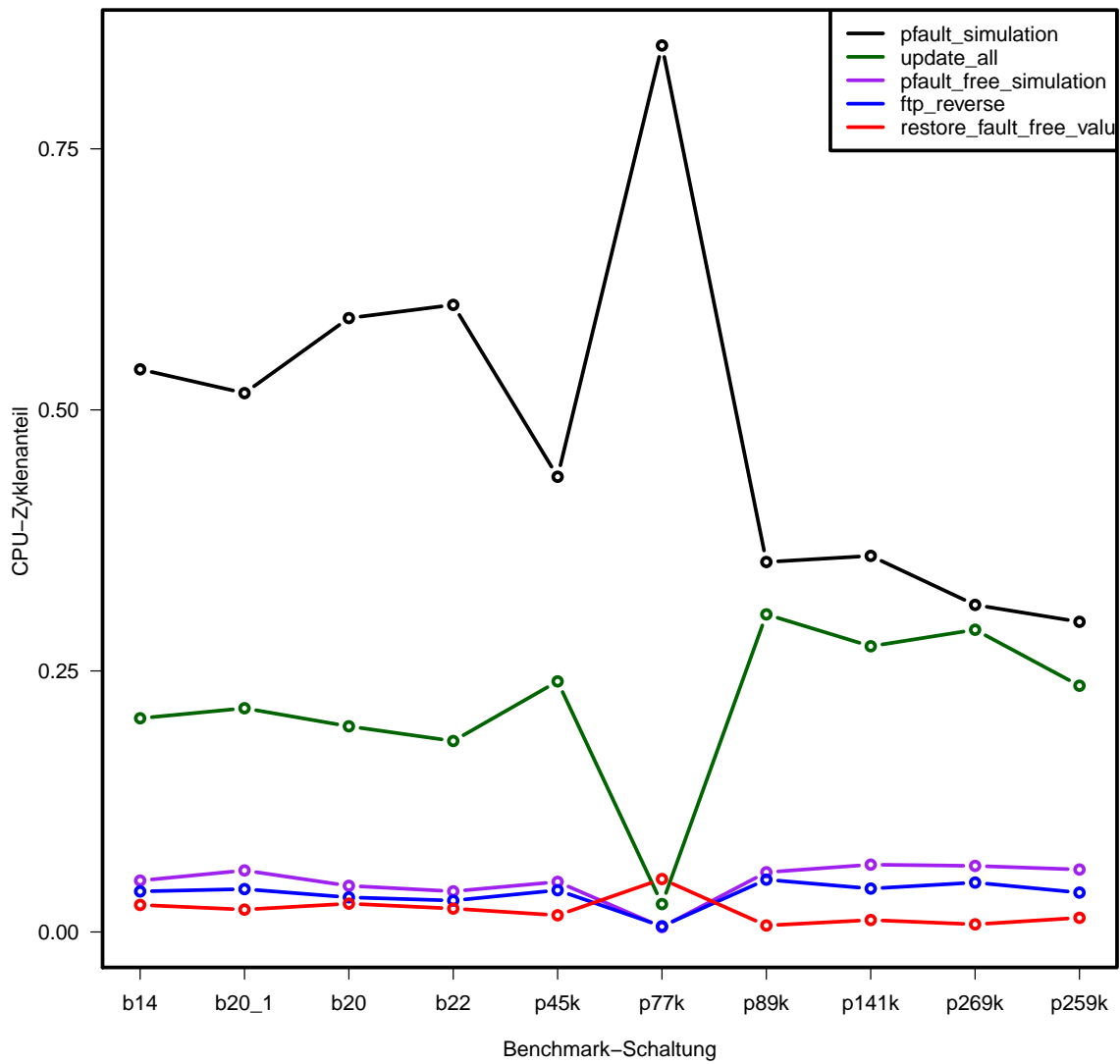


Abbildung 4.2: Profiling – Genutzter CPU-Zyklus-Anteil

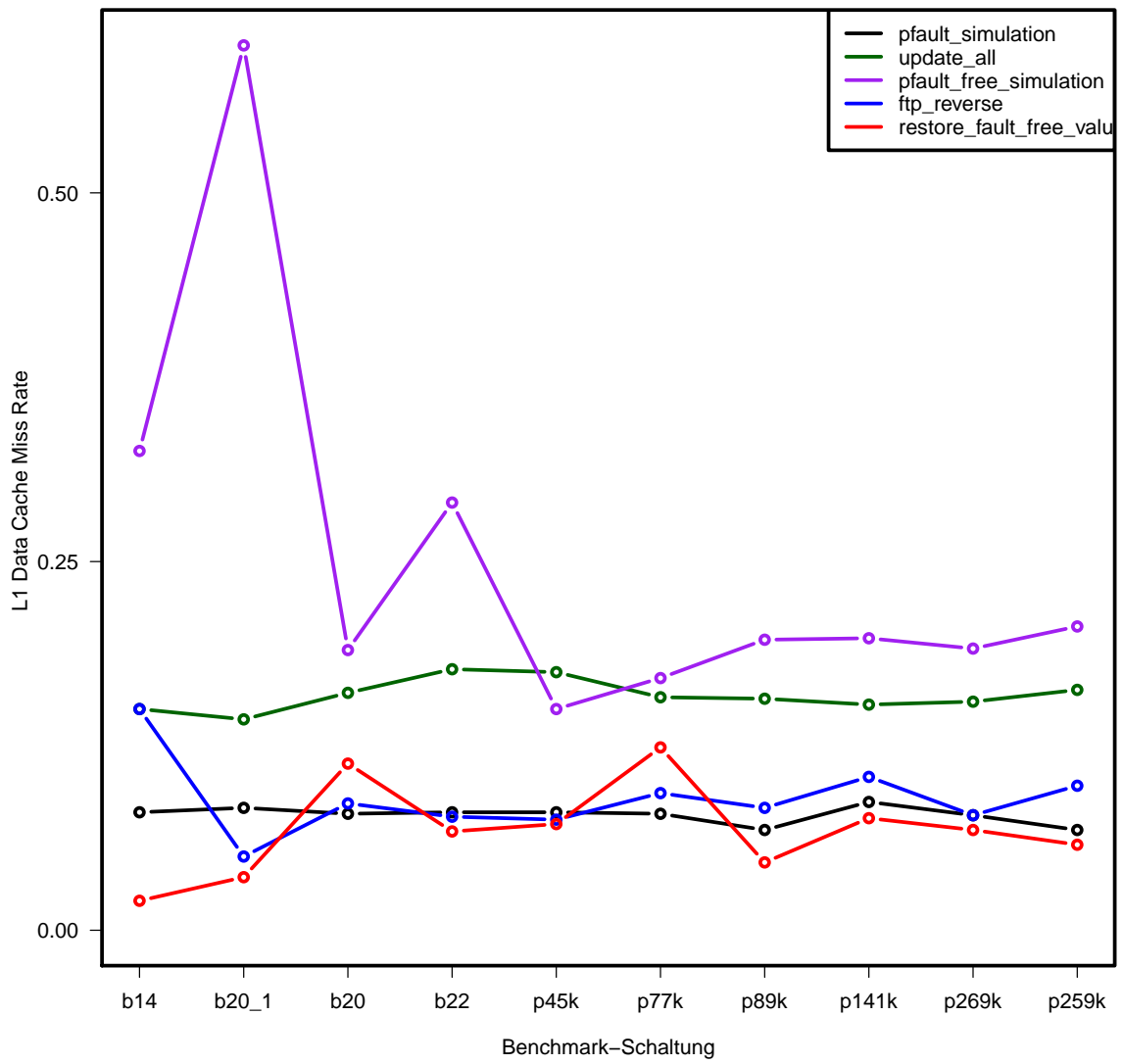


Abbildung 4.3: Profiling – L1-Data-Cache Miss Rate

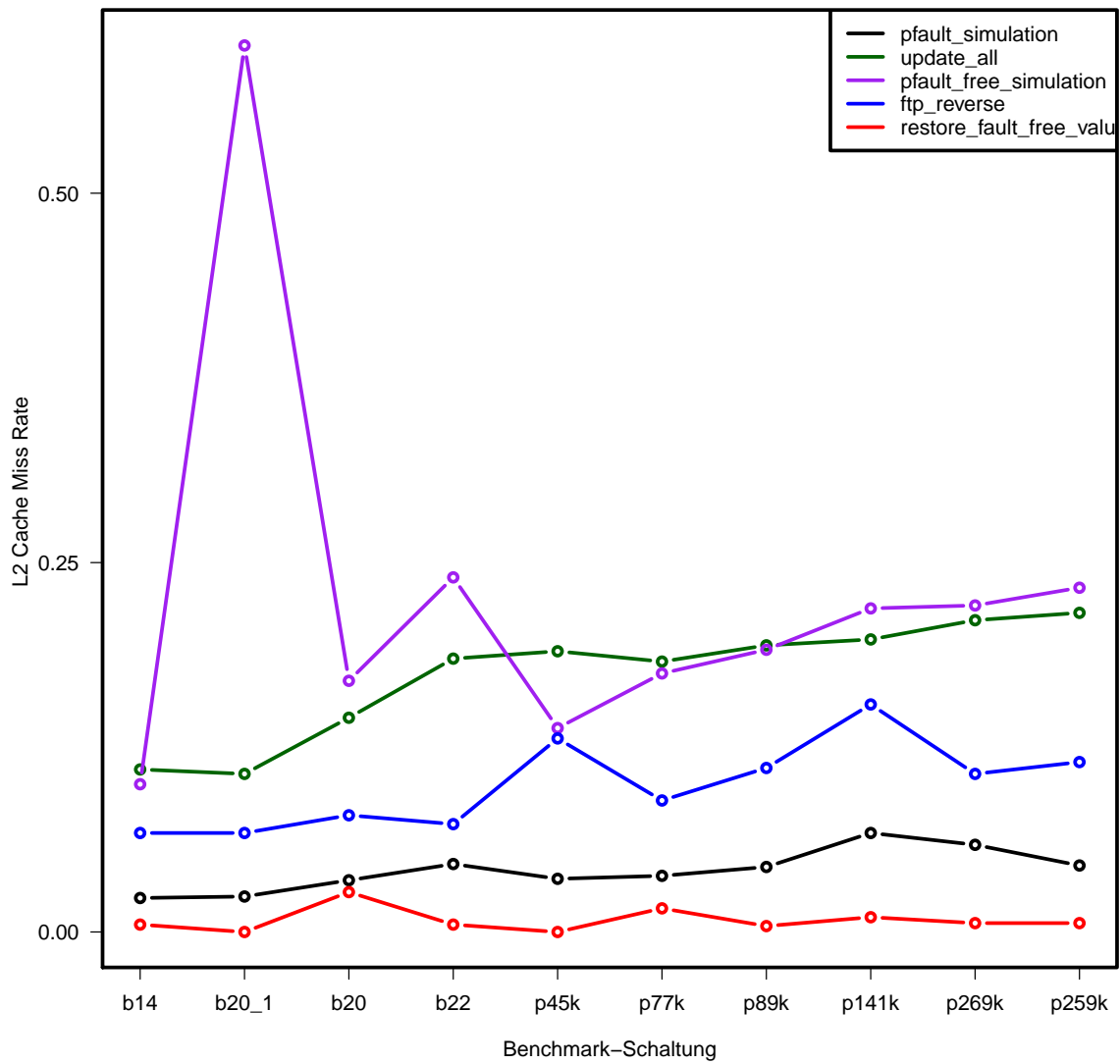


Abbildung 4.4: Profiling – L2-Cache Miss Rate

NVIDIA Compute Unified Device Architecture

CUDA ist NVIDIAS GPGPU-Implementierung und besteht aus entsprechender Hard- sowie Software. Es ermöglicht die einfache Programmierung der GPGPU in C++ mit nur wenigen Erweiterungen der Programmiersprache. Die Spracherweiterung CUDA ermöglicht es, viele tausende Threads parallel zu starten. Durch die große Zahl von Threads kann die GPGPU-Hardware Speicherlatenzen bei der Ausführung verstecken. Dies ist notwendig, weil die durchschnittliche Zugriffslatenz des Hauptspeichers bei der GPGPU nicht durch Caches vermindert wird.

Dieses Kapitel geht zuerst grob auf die Funktionsweise und den Aufbau der Architektur ein. Danach wird die Speicherhierarchie erläutert. Darauf folgt ein kurzer Abriss über die effiziente Nutzung der Plattform und wie sie sich für die Fehlersimulation nutzen lässt. Abschließend folgt die Erwähnung einiger nützlicher Werkzeuge für die Entwicklung mit CUDA.

5.1 Funktionsweise

Ein NVIDIA Grafikprozessor besteht im Wesentlichen aus sehr vielen einfachen Prozessorkernen. Der genaue Aufbau ist ein Geheimnis von NVIDIA, so dass an dieser Stelle teilweise spekuliert werden muss. Abbildung 5.1 gibt einen groben Überblick über die Architektur. Jeder Prozessorkern ist ein 8-Wege-SIMD-Kern (Single Instruction Multiple Data), bei NVIDIA „Multicore“ genannt. Der Multicore basiert auf einer In-Order-Architektur, d.h. die Befehle des Instruktionstroms werden in genau dieser Reihenfolge ausgeführt. Eine Umsortierung findet nicht statt. Jede Spur des SIMDs wird jedoch unabhängig ausgeführt, weshalb NVIDIA an dieser Stelle von SIMT (Single Instruction Multiple Threads) spricht. Sobald verschiedene Threads divergierende Sprünge nehmen, werden diese sequentiell ausgeführt.

Bei CUDA bilden 32 Threads einen Warp. Warps besitzen die Eigenschaft, dass diese prinzipiell im Gleichschritt-Verfahren (engl. lock-step) ausgeführt werden, was bedeutet, dass

innerhalb eines Warps alle Threads den gleichen Instruktionsstrom ausführen, es sei denn einzelne Threads divergieren. CUDA fügt während der Übersetzung bei möglichen Verzweigungen die notwendigen Instruktionen automatisch hinzu, weshalb sich der Programmierer nicht darum kümmern muss.

Probleme werden in CUDA in Gitter (engl. grids) und Blöcke (engl. blocks) aufgeteilt. Dabei können Blöcke in bis zu drei Dimensionen definiert werden, was eine natürlichen Entsprechung vieler Probleme darstellen soll. Die maximale Größe der Dimensionen beträgt 512 für die x- und y-Koordinate sowie 64 für die z-Koordinate. Bis zu acht Blöcke werden von einem Multiprozessor gleichzeitig ausgeführt. Innerhalb eines Blocks können sich die Threads synchronisieren, was vom Programmierer aber erzwungen werden muss.

Ein Grid bildet eine weitere Abstraktion für die Blöcke, so dass mehrere Teilprobleme in einem Durchlauf gelöst werden können. Für Grids stehen nur zwei Dimensionen zur Verfügung, welche in beide Dimensionen jeweils 65,535 groß sein können.

CUDA führt weitere Schlüsselwörter für die Sprache C++ ein, um z.B. Funktionen, die auf der GPGPU ausgeführt werden können, oder spezielle Speicherbereiche zu markieren. Es gibt dabei nur wenige Einschränkungen, die das Arbeiten mit CUDA erschweren.

Ein Kernel ist eine Funktion, die auf der GPGPU ausgeführt wird. Dafür muss eine C++ Funktion mit dem `__global__`-Pragma versehen werden. Der Kernel kann dann mit Angabe der entsprechenden Dimensionen gestartet werden. Eine Funktion, die als Kernel deklariert wurde, darf als Rückgabewert nur den Datentyp „void“ haben. Darüber hinaus kann ein Kernel keinen eigenen Speicher allokkieren. Die Allokation wird von der CPU, bei NVIDIA Host genannt, angefordert und von der CUDA-Laufzeitumgebung verwaltet.

Um die Verwaltung der vielen Threads möglich zu machen, wurde auf einen Stack verzichtet, weshalb die zu verwaltenden Kontextinformation eines Threads sehr gering ausfallen. Dadurch ist es allerdings nicht möglich, dass Funktionen sich rekursiv aufrufen. Diese werden bei CUDA stets in den Kernel eingefügt und sind daher ähnlicher zu Makros als zu Funktionen selbst.

Der typische Programmablauf mit CUDA sieht folgendermaßen aus. Zuerst bereitet die CPU Daten für die GPGPU auf. Danach werden die Daten auf die GPGPU kopiert und anschließend ein oder mehrere Kernel aufgerufen, welche die Daten verarbeiten. Zum Schluss werden die Ergebnisse von der GPGPU auf den Host zurück kopiert und dort ausgewertet.

In CUDA gibt es noch weitere Konzepte wie z.B. Streams. Diese wurden in dieser Studienarbeit nicht genutzt, weshalb nicht näher darauf eingegangen wird.

5.2 Speicherhierarchie

Jeder Multiprozessor der GPGPU hat auf der ersten Ebene der Speicherhierarchie 8.192 32-Bit-Register bis CUDA-Version 1.1 und doppelt so viele ab Version 1.2. Diese Zahl wirkt zuerst einmal erstaunlich groß, verglichen zu den vier frei verwendbaren Registern eines

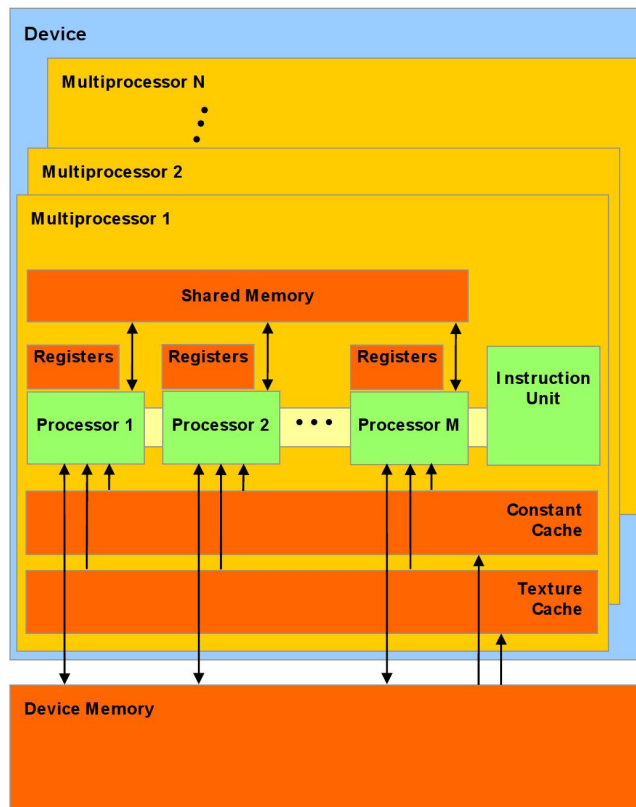


Abbildung 5.1: Aufbau der Architektur einer CUDA GPGPU

IA-32-Prozessors. Allerdings werden die Register auf alle Threads aus diesem Registerpool verteilt. So hat jeder Thread seine persönlichen Register, auf die nur er zugreifen kann. Wird die maximale Anzahl von Threads für einen Multiprozessor gescheduled, so stehen maximal 16 Register pro Thread für CUDA-Version 1.2 zur Verfügung. Jedes Register kann, wie bei Register-Register-Maschinen üblich, pro Takt gelesen bzw. geschrieben werden.

Zur Kommunikation zwischen verschiedenen Threads innerhalb eines Blocks besitzt CUDA auf der zweiten Speicherhierarchie den Shared Memory. Dieser besteht aus 16 kB große SRAM pro Multiprozessor, eingeteilt in 16 Bänke, welches sich auf dem Chip selbst befindet. Durch die Einteilung in Bänke kann eine hohe Bandbreite erzielt werden, solange keine Bankkonflikte auftreten. Diese treten auf, wenn zwei oder mehr Threads auf die gleiche Bank, aber auf verschiedene Indizes zugreifen. In diesem Fall werden die Zugriffe sequentiell ausgeführt. Der Shared Memory wird auf die einzelnen Blöcke aufgeteilt, die von einem Multiprozessor ausgeführt werden, so dass der ohnehin schon kleine Speicher weiter verringert wird. Der Shared Memory dient neben der Kommunikation zwischen Threads quasi als L1-Daten-Cache-Ersatz. Allerdings ist es dem Programmierer überlassen, diesen zu verwalten. Es stehen keine funktionalen Einheiten in Hardware bereit, die die Verwaltung beschleunigen könnten.

Als letzte Ebene steht der allgemeine DRAM-Speicher der GPGPU zur Verfügung, der bei CUDA als Global Memory bezeichnet wird. Dieser ist groß und mit hoher Bandbreite extern angebunden, aber die Lese- und Schreibzugriffe benötigen mehrere hundert Takte bis sie ausgeführt sind. Aus diesem Grund sollten Zugriffe auf den Global Memory möglichst vermieden werden oder geordnet und zusammenhängend (engl. coalesced) sein, um Streaming zu ermöglichen.

CUDA besitzt darüber hinaus weitere virtuelle Speicher. Der sogenannte Local Memory liegt je nach Größe und Auslastung entweder im Shared oder Global Memory und dient der Auslagerung von lokalen Feldern. Das Besondere am Local Memory ist, dass er für jeden Thread lokal ist, d.h. er bildet einen eigenen Speicherbereich pro Thread. Allerdings ist seine Maximalgröße auf 2 kB pro Multiprozessor beschränkt, so dass nur wenige Daten darin Platz finden. Wenn der Local Memory in den Global Memory abgebildet wird, ist die Latenz entsprechend groß, anders als es der Name suggerieren würde.

Der Texture Memory bietet spezielle Adressierungsarten für 3D-Szenen um Objekte im 3D-Raum effizienter verwalten zu können. Er lässt sich aber auch für 1D- oder 2D-Adressierungen nutzen. Der Texture Memory wird ebenfalls in den Global Memory abgebildet, bildet aber einen Nur-Lese-Speicher. Dafür besitzt jeder Multiprozessor einen Texture-Cache von ca. 8 kB, der Zugriffe beschleunigen kann. Der PTX-Befehlssatz bietet spezielle Instruktionen für den Zugriff auf den Texture Memory.

Zu aller Letzt gibt es noch den Constant Memory. In diesen Speicherbereich kann nur vom Host geschrieben werden. Die GPGPU kann die Werte lediglich lesen. Er besitzt eine globale Größe von 64 kB. Wie beim Texture-Memory gibt es auch für den Constant Memory einen Cache.

5.3 Effiziente Nutzung der Architektur

Um die Architektur effizient ausnutzen zu können, werden im CUDA Best Practices Guide [NVI] verschiedene Entwurfsmuster beschrieben. Diese werden zum besseren Verständnis an dieser Stelle kurz wiedergegeben, wobei nur die Punkte aufgegriffen wurden, die für diese Arbeit relevant sind.

Parallize sequential Code CUDA benötigt sehr viele Programmteile die unabhängig voneinander ausgewertet werden können. Dadurch ist es erforderlich, dass das Problem bzw. die Algorithmen entsprechend aufgebaut sind. Sequentielle Algorithmen sollten nach Möglichkeiten durch parallele Versionen ersetzt werden.

Coalesced Access to Global Memory Der Global Memory besteht aus DRAM, welches sich nicht auf dem Chip selbst befindet. Dadurch entstehen auf der einen Seite hohe Latenzen und auf der anderen Seite erfordert es bestimmte Zugriffsmuster um den Global Memory effizient zu nutzen, da DRAM in Bänken organisiert ist. Wenn eine Bank geöffnet wird und nur ein kleiner Teil daraus gelesen wird, z.B. ein Wort, dann ist dies ineffizient, weil die übrigen Daten in der Bank dennoch ausgelesen und anschließend verworfen werden.

Der Leitfaden rät deshalb auf diese Zugriffsmuster zu achten und alle Speicherzugriffe möglichst zusammenhängend zu organisieren. Zusammenhängend bedeutet, dass die 16 Threads eines Halb-Warps die Daten aus einer einzelnen Bank auslesen und die Daten dabei möglichst nebeneinander angeordnet im Speicher befinden. Es darf dabei auch einzelne Lücken im Zugriffsmuster geben.

Um die maximale Bandbreite des Speichers zu erreichen, müssen die Daten zusammenhängend gelesen werden.

Shared Memory Werden die Daten häufig wieder genutzt oder ist das Zugriffsmuster zufällig, aber nur auf einen kleinen Speicherbereich beschränkt, so bietet sich der Shared Memory zur Zwischenspeicherung an.

Allerdings gilt es beim Shared Memory auf Bankkonflikte zu achten. Diese zwingen den Multiprozessor zu einer sequentiellen Ausführung der Zugriffe. Anders als bei einem Zugriff auf den Global Memory, können andere Threads den Multiprozessor so lange nicht benutzen.

Branching and Divergence Es ist in CUDA möglich, dass Threads innerhalb eines Warps verschiedene Programmpfade ausführen. Jedoch kann ein Multiprozessor nur eine einzige Instruktion gleichzeitig ausführen, so dass die Programmpfade sequentiell ausgeführt werden müssen. Wie bei anderen SIMD-Architekturen gibt es z.B. die bedingte Ausführung sowie die Select-Instruktion. Diese ermöglichen es, bei gleichem Instruktionsstrom unterschiedliche Daten auszuwerten.

5.4 Eignung von GPGPUs für parallele Fehlersimulation

Die Vorteile von CUDA gegenüber einer CPU sind, kurz gefasst, die höhere Speicherbandbreite und viel mehr Recheneinheiten, wie es aus Tabelle 5.1 ersichtlich ist. Dies eignet sich im Prinzip sehr gut für die Fehlersimulation, da die Parallelität der Auswertung von Gatterknoten, Mustern und Fehlern genutzt werden kann. Die kritische innere Schleife der Verzweigungsstammsimulation besteht im wesentlichen aus folgenden Schritten:

Listing 5.1 Kritische Innere Schleife

```
1 while (Ereignis vorhanden) do
2   Lade naechstes Ereignis
3   Lade Eingangswerte der Vorgaengerknoten
4   Werte Gatter aus
5   Lade Gutwert
6   Vergleiche Gutwert mit berechnetem Wert
7   Propagiere Ereignis
8   Speichere Ausgangswert
9 done
```

Im Pseudocode aus Programmauflistung 5.1 ist erkennbar, dass für eine einzelne Logiksimulation eines Gatterknotens sehr viele Speicheroperationen ausgeführt werden müssen. Das Auswerten und Vergleichen eines einzelnen Gatterknotens kostet dahingegen kaum Rechenleistung.

		CPU	GPGPU
Kerne		2	240
Transistoren	[<i>mio.</i>]	154	1400
Die-Größe	[<i>mm</i> ²]	147	470
Fertigungstechnologie	[<i>nm</i>]	90	55
L1-Daten-Cache	[<i>kB</i>]	64	16
L2-Cache	[<i>kB</i>]	512	–
Chip-Taktrate	[<i>MHz</i>]	2412	648
Speicherbandbreite	[<i>GB/sec</i>]	5,4	159,0
Energieverbrauch	[<i>W</i>]	110	204

Tabelle 5.1: Vergleich von AMD Athlon64 X2 4600+ und NVIDIA GTX-285

Wie bereits ausgeführt sind Zugriffe auf den Hauptspeicher sehr teuer, da diese bei CUDA nicht in einem schnellen Zwischenspeicher (engl. Cache) gepuffert werden. Darüber hinaus liegen die Eingangswerte eines Gatterknotens relativ zufällig im Speicher. Dies ist für DRAM nicht optimal, denn es müssen sehr viele Bänke geöffnet und geschlossen werden, so dass es zu weiteren Verzögerungen kommt bis alle Threads aus einem Warp ihre Daten erhalten haben. In CUDA ist das Problem durch Multi-Threading gelöst. Es werden mehrere hundert Threads pro Multiprozessor gestartet, dann können bereits Threads ihre Verarbeitung fortführen, während andere Threads warten, bis ihre Speicherzugriffe abgeschlossen sind.

CUDA ist für datenparallele Probleme, wie die Berechnung von 3D-Szenen in der Visualisierung, entworfen worden. Alle Threads innerhalb eines Warps werden im Gleichschritt ausgeführt. Sollte es durch Sprünge zu einer oder mehreren Verzweigungen im Programm kommen, so werden die einzelnen Teile seriell ausgeführt. Daraus ergeben sich Synchronisationsschwierigkeiten, wenn eine einzelne Datenstruktur, wie z.B. Prioritätswarteschlange, von mehreren Threads genutzt werden soll. Aus diesem Grund ist die Verwaltung der Stapelspeicher, wie es im Atalanta Fehlersimulator implementiert wurde, nicht ohne weiteres effizient auf der GPGPU möglich.

Gutsimulation

In der Gutsimulation und bei der Fehleraufgabe wird der Schaltungsgraph topologisch sortiert ausgewertet und alle Gatterknoten eines Rangs können parallel bearbeitet werden. Dafür wird die Maximalzahl an Threads in einem Block ausgenutzt. Diese beträgt bei CUDA 1.3 512 Threads.

Eine effiziente Synchronisation über mehrere Multiprozessoren oder Blöcke ist in CUDA nicht vorgesehen. Eine Voraussetzung um Probleme mit GPGPUs zu beschleunigen ist die datenparallele Unabhängigkeit. Ein Problem sollte sich unabhängig partitionieren und berechnen lassen, ohne auf andere Berechnungen synchronisieren zu müssen. Eine Möglichkeit ist die Synchronisation über den Host. Wenn ein Kernel auf der GPGPU gestartet wird, kann der Host angewiesen werden bis zu dessen Beendigung zu warten. So lassen sich mehrere Multiprozessoren synchronisieren, aber die Kommunikation über den Host verlangsamt die Simulation zu sehr, als dass sich der Geschwindigkeitsvorteil mehrerer Multiprozessoren bemerkbar machen würde. Die Synchronisation über den Host ist daher nicht sinnvoll.

Verzweigungsstammsimulation

Es lassen sich sehr viele Threads für die Verzweigungsstammsimulation nutzen, da jeder Verzweigungsstamm unabhängig in einem Warp berechnet werden kann. Hierfür benötigt jeder Verzweigungsstamm seinen eigenen Speicherbereich für die Zwischenergebnisse.

Beobachtbarkeit und Fehlererkennung

Die Auswertung der Beobachtbarkeit und der Fehlererkennung wird, wie in der Gutsimulation auch, topologisch sortiert ausgeführt. Dabei wird die maximale Anzahl an Threads pro Block gestartet um möglichst viel Nutzen aus dem Multi-Threading zu erhalten. Ein Thread kann einen Gatterknoten auswerten. So lassen sich die lesenden Zugriffe auf Gatterknoteninformation und Beobachtbarkeit effizient mit zusammenhängenden Zugriffsmustern implementieren. Das Schreiben der Beobachtbarkeit und der Fehlerentdeckung kann ebenfalls zusammenhängend erfolgen.

5.5 Werkzeuge

CUDA bietet neben mehreren Anleitungen auch nützliche Werkzeuge um den Entwickler bei der Arbeit zu unterstützen. Diese werden folgend kurz vorgestellt.

CUDA Profiler

Als erstes sei hier der CUDA Profiler erwähnt. Dieser kann Performance Counter für Speicherzugriffe, divergente Sprünge und Instruktionen auslesen und grafisch darstellen. Im Gegensatz zur VTune-Suite von Intel, kann der CUDA Profiler keine Verbindung zum ursprünglichen Quellcode herstellen. Leider sind die ermittelten Daten nicht sehr aussagekräftig. Es fehlt eine genaue Erklärung bzw. Architekturbeschreibung, wie sich die einzelnen Werte auf die Leistung der GPGPU auswirken.

NVCC

Der CUDA Compiler NVCC bietet eine Option um PTX-Assembler-Code zu erzeugen. Dies ist nützlich um zu sehen wie der C++-Code auf der Grafikkarte umgesetzt wird. Durch den PTX-Code können unter Anderem Zugriffe auf den Global Memory gefunden werden. So wurde z.B. festgestellt, dass mehrere lesende Zugriffe auf den gleichen Index eines Felds im Global Memory nicht in Registern zwischengespeichert werden. Statt dessen wird jeder Zugriff explizit auf den Global Memory abgebildet. Dies ist korrekt, da jeder Multiprozessor den Inhalt nebenläufig ändern kann. Die Zahl der Register kann in diesem Zwischenschritt der Kompilierung nicht erkannt werden. Der Assembler-Code wird, vor der eigentlichen Ausführung auf der GPGPU, noch durch die CUDA-Laufzeitumgebung in den binären Instruktionsstrom übersetzt. Dies ist notwendig, da ältere CUDA-Versionen nach der Kompilierung des C++-Codes ebenfalls noch funktionieren müssen.

NVCC kann auch den Binärcode für eine spezifische Architektur ausgeben. Das Projekt Decuda¹ entwickelt ein Programm, welches den CUDA Binärcode dekompilet. So

¹<http://wiki.github.com/laanwj/decuda>

kann der tatsächliche Instruktionsstrom in PTX für die GPGPU erhalten werden. Dieser ist allerdings stark optimiert und somit ist es schwierig, die Verknüpfung zwischen Originalcode zu dekompiertem Assembler herzustellen. In dieser Form können jedoch die kritischen inneren Schleifen genauer untersucht und weiter optimiert werden, um z.B. die Zahl der Register zu reduzieren.

Auslastungsberechnung

NVIDIA bietet ein Tabellenkalkulations-Dokument mit Makros, um die Auslastung der GPGPU auszurechnen und grafisch darzustellen. Diese ist im CUDA SDK bereits enthalten. Durch Eingabe der Zahl der pro Thread genutzten Register, sowie die Zahl der Threads und die Größe des Shared Memories pro Block, wird die maximale Auslastung errechnet. Darüber hinaus wird in drei Grafiken dargestellt, wie sich eine Änderung an einer der drei Variablen auf die Auslastung auswirkt. So lassen sich einzelne Kernel weiter optimieren.

Parallele Implementierung mit CUDA

In dieser Studienarbeit wurden zwei unterschiedliche Ansätze für die Verzweigungsstammssimulation verfolgt. Einen Ansatz, der die topologisch sortierte Netzliste vollständig, aber sehr schnell abarbeitet und eine ereignisgesteuerte Variante, welche versucht, nur notwendige Gatterknotenauswertungen durchzuführen, dafür aber mehr Aufwand pro Auswertung benötigt. Die ereignisgesteuerte Variante wird als Vorschlag getrennt in Anhang A betrachtet.

Das Ablaufdiagramm in Abbildung 6.1 gibt einen Überblick über die Implementierung des mittels GPGPU beschleunigten PPSFP-Algorithmus. Nach der Initialisierung der Datenstrukturen und Einlesen der Schaltung werden beim Test mit Zufallsmuster die entsprechenden Testmuster erzeugt. Die nächsten Schritte werden für all diese ausgeführt, bis sie alle simuliert wurden. Zuerst werden die Testmuster auf die GPGPU kopiert. Dort wird eine Gutsimulation durchgeführt, um die Wertebelegung der Schaltung im fehlerfreien Fall zu ermitteln. Anschließend werden die Werte wieder zurück auf den Host kopiert. Nun werden die folgenden Berechnungen parallel auf dem Host und der GPGPU ausgeführt. Die Berechnung der Beobachtbarkeit an den Verzweigungsstämmen wird zuerst durchgeführt. Dafür wird für jeden Verzweigungsstamm der Gutwert invertiert und somit die Fehler injiziert. Für jede Injektion wird eine Fehlerfortpflanzung, d.h. eine Logiksimulation, ausgeführt. Danach wird die Beobachtbarkeit aus der Kontravalenz des Gut- und Falschwerts gebildet. Abschließend werden alle veränderten Gutwerte zurückgesetzt. Nachdem alle Fehler für die Verzweigungsstämme injiziert und simuliert wurden, kann nun die Beobachtbarkeit in den verzweigungsfreien Regionen berechnet werden. Mit Hilfe der Beobachtbarkeiten kann letztlich die Fehlererkennung bestimmt werden. Nachdem alle Testmuster simuliert wurden, wird die Matrix der erkannten Fehler von der GPGPU auf den Host zurückkopiert und dort ausgewertet. Abschließend kann die Fehlerabdeckung abgeleitet werden oder die Menge der erkannten Fehler in eine Datei gespeichert werden.

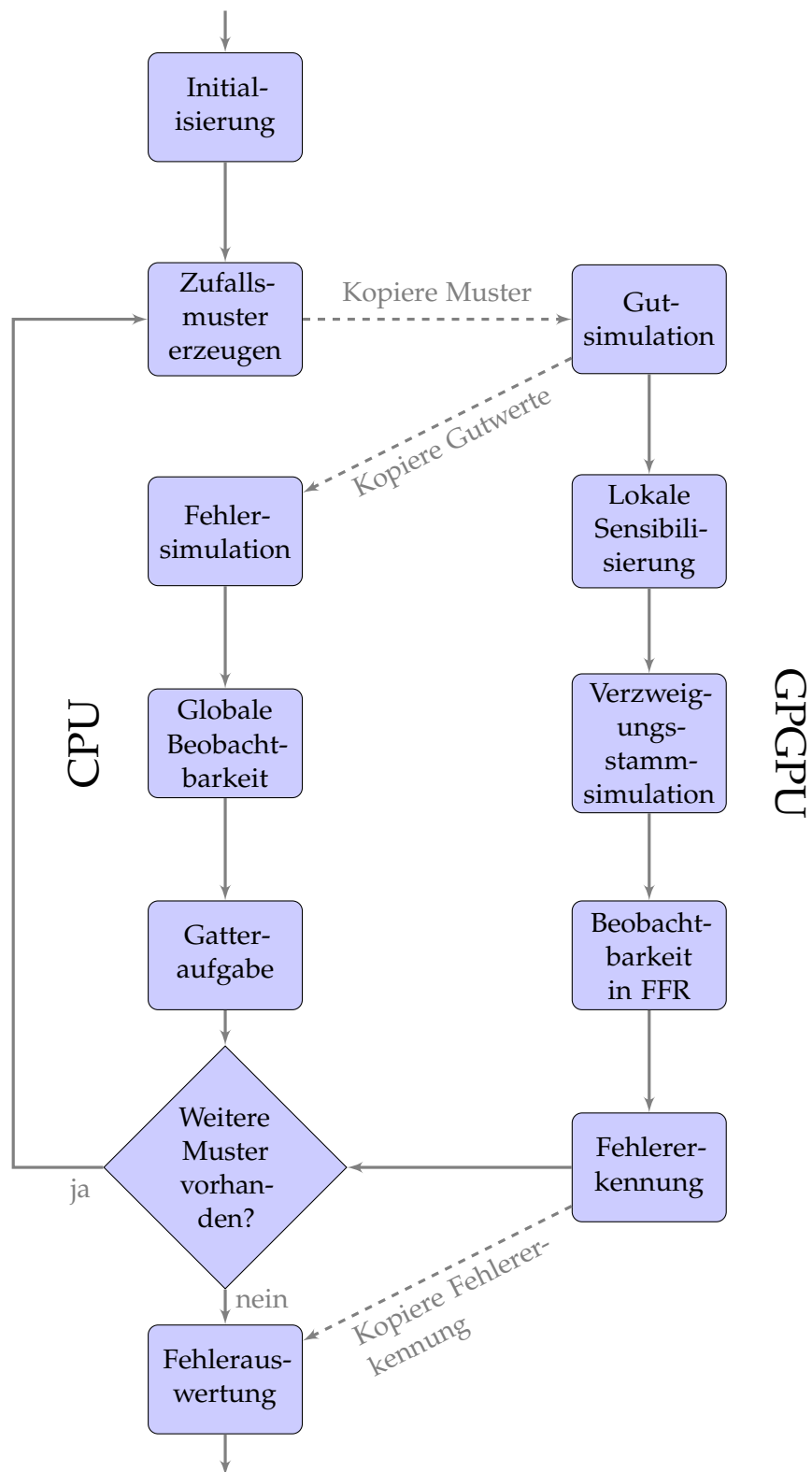


Abbildung 6.1: Ablaufdiagramm GPGPU-Simulation

6.1 Gutsimulation und Fehleraufgabe

In der Gutsimulation wird eine Logiksimulation der Schaltung im fehlerfreien Fall durchgeführt.

Die Gutsimulation nutzt eine vereinfachte Struktur der ursprünglichen Netzliste. Diese ist in Programmauflistung 6.1 abgebildet. Es werden lediglich die Informationen über die beiden Eingangskanten sowie die Funktion benötigt. Die Auswertung erfolgt von den Primäreingängen zu den Primärausgängen, d.h. vorwärtsgerichtet. Dafür werden 512 Threads genutzt, welche sich zwischen den einzelnen Ebenen des topologisch sortierten Schaltungsgraphen synchronisieren. Auf diese Weise werden die Schaltungsinformationen zusammenhängend gelesen sowie die ausgewerteten Gutwerte zusammenhängend geschrieben. Die Werte der Eingänge sind jedoch relativ zufällig über den bereits bearbeiteten Graphen verstreut, was die Leistung reduziert.

Listing 6.1 Datenstruktur eines Gatterknotens

```

1  typedef struct CONNECTORD {
2      int in0, in1; // Vorgaengerknoten
3      logic fn;    // Funktion
4      int type;   // 1 Verzweigungsstamm, 0 Verzweigungsfrei
5      int fo;     // Falls type = 1: Index zur Partition der Verzweigungssimulation
6                 // Falls type = 0: Index zum Verzweigungsstamm
7  } CONNECTORDTYPE;

```

Zusammen mit der Gutsimulation wird die lokale Aktivierbarkeit in den verzweigungsfreien Gebieten der Verzweigungsstämme berechnet. Für jeden Verzweigungsstamm wird evaluiert, ob er in der darauf folgenden Verzweigungsstammsimulation ausgewertet werden muss oder nicht. Die Information wird mittels „fo“ im Feld der Verzweigungsstämme abgelegt.

Nach der erfolgten Gutsimulation werden die Werte wieder auf den Host kopiert, so dass dieser die Verzweigungsstammsimulationen für die nicht auf der GPGPU vorgesehenen Verzweigungsstämme durchführen kann. Es wurden keine weiteren Optimierungen bezüglich der GPGPU-Architektur, wie z.B. Nutzung von Shared oder Texture Memory, durchgeführt.

6.2 Verzweigungsstammsimulation

Um Atalanta wesentlich zu beschleunigen, muss die explizite Verzweigungsstammsimulation als rechenzeitaufwendigste Operation beschleunigt werden. Die Simulation eignet sich sehr gut zur Parallelisierung, denn jeder Verzweigungsstamm kann unabhängig von den anderen Verzweigungsstämmen berechnet werden. Es muss dafür Sorge getragen werden, dass sich die parallelen Simulationen nicht gegenseitig beeinflussen. Dies lässt sich bewerkstelligen, indem jede Simulation eine Kopie der Gutsimulationswerte in einem eigenen Speicherbereich zur Verfügung gestellt bekommt. Eine einzelne Verzweigungsstammsimulation kann nur

den Ausgangskegel beeinflussen. Es ist also ausreichend, die Kopie der Gutwerte nur auf diesen Kegel des Verzweigungsstamms zu beschränken.

Der Schaltungsgraph wird dementsprechend in Partitionen von Ausgangskegeln aufgeteilt. Die nötige Datenstruktur findet sich in Programmauflistung 6.2. Eine Partition wird darüber hinaus in drei Teile aufgeteilt. Diese umfassen jene Signale, welche von außen „offPart“ in die Partition hineinkommen. Darauf folgen die eigentlichen Logikgatterknoten „subgraph“. Abschließend werden die Primärausgänge „outputs“ gespeichert.

Listing 6.2 Datenstruktur einer Partition

```
1 type Partition {
2     int offPart[]; // Indices zur Partitions-grenze
3     int subgraph[]; // Kodierte Knoten
4     int outputs[]; // Ausgaenge der Partition
5     Fanout fo[]; // Verzweigungsstaemme in
6                 // umgekehrter Ordnung
7                 // zu subgraph[]
8 }
```

In Programmauflistung 6.3 findet sich die Beschreibung eines Verzweigungsstamms. Dieser besteht aus zwei Teilen. Zum einem aus dem Index, der angibt, an welcher Stelle sich der entsprechende Verzweigungsstamm in der Partition befindet „inPart“. Wenn die Beobachtbarkeit berechnet wurde, muss diese in den globalen Schaltungsgraphen zurückgeschrieben werden, wofür der Eintrag „global“ dient.

Listing 6.3 Datenstruktur eines Verzweigungsstamms

```
1 type Fanout {
2     int inPart; // Stamm in Partition
3     int global; // Eintrag in sensitivities[]
4 }
```

In Programmauflistung 6.4 ist der Pseudocode für die Auswertung einer Partition abgedruckt. Dazu werden zuerst alle Eingangswerte der Partition gelesen sowie eine Gutsimulation durchgeführt und die Werte zu den Primärausgängen kopiert. Anschließend werden alle Verzweigungsstämme innerhalb der Partition ausgewertet. Zuerst wird überprüft, ob der Verzweigungsstamm lokal aktiviert wurde. Falls dies nicht passiert ist, wird der Verzweigungsstamm nicht ausgewertet. Dann wird der Fehler durch Invertierung des Gutwerts injiziert und die Fehlerfortpflanzung berechnet. Abschließend wird die Beobachtbarkeit ermittelt und im globalen Beobachtbarkeitsfeld gespeichert.

Um die Verzweigungsstammsimulation möglichst effizient ausführen zu können, werden immer 32 Gatterknoten von einem Warp gleichzeitig evaluiert. Um Abhängigkeiten zwischen den einzelnen Gatterknoten aufzulösen, wurden die topologisch sortierten Ränge der Schaltung auf Vielfache von 32 aufgefüllt. Dies hat den Vorteil, dass Gatterknoteninformationen zusammenhängend gelesen werden können.

Es ergeben sich allerdings auch Nachteile. Zum Beispiel evaluieren die Gatterknoten unterschiedliche Funktionen. Würde die Funktion durch eine Fallunterscheidung ausgewertet

Listing 6.4 Berechnung der Beobachtbarkeit eines Verzweigungsstamms

```

1 procedure evaluate_partition(Partition p)
2   int values[];
3   int outputs[];
4   for i = 0 .. |p.offPart|-1
5     values[i] = faultFreeValues[p.offPart[i]];
6
7   // fehlerfreie Simulation
8   for i = 0 .. |p.subgraph|-1
9     evaluate(i);
10
11  // kopiere fehler-freie Ausgaenge
12  for i = 0 .. numOutputs-1
13    outputs[i] = values[|p.subgraph|-i]
14
15  for all f in p.fo[]
16    continue if sensitivities[f.global] == 0;
17
18    values[f.inPart] = not values[f.inPart];
19
20    // Fehlerfortpflanzung
21    for i = f.inPart+1 .. |p.subgraph|-1
22      values[i] = evaluate(i);
23
24    // Auswertung der Beobachtbarkeit
25    int sens=0;
26    for i = 0 .. p.numOutputs-1
27      sens |= (outputs[i] xor values[|p.subgraph|-i]);
28    sensitivities[f.global] = sens;

```

werden, so würde es zur Divergenz der einzelnen Threads führen. Prinzipiell könnte die bedingte Ausführung, welche in CUDA vorhanden ist, dafür genutzt werden. Allerdings setzt der NVCC dies nicht um. Es werden keine bedingten Ausführungen inferiert, so dass diese Möglichkeit leider entfällt. Statt dessen wird die Select-Instruktion und ein 1-aus-n-kodiertes Muster genutzt, um die Gatterknoten auszuwerten. Dies erfordert mehr Instruktionen, aber Sprünge und Divergenzen werden vermieden.

Um dies genauer erklären zu können wird zuerst die Datenstruktur eines Gatterknotens und die Kodierung der Informationen vorgestellt. Die Information wird in einem 32-Bit-Datenwort gespeichert, wie es in Tabelle 6.1 dargestellt ist. Die Funktionsinformationen sind am Anfang des Datenworts kodiert. Darauf folgen die Indices der beiden Vorgängerknoten.

Bitposition	31	30	29	28	27 .. 16	15 .. 0
Funktion	AND	OR	XOR	INV	IN ₁	IN ₀

Tabelle 6.1: Datenstruktur eines Gatterknotens

Die Kodierung der einzelnen Boole'schen Funktionen ist in Tabelle 6.2 dargestellt. Durch die Nutzung einer 1-aus-4-Kodierung entsteht Redundanz. Diese vermindert aber den Dekodieraufwand.

Funktion	Kodierung
AND	1000
NAND	1001
OR	0100
NOR	0101
XOR	0010
XNOR	0011
BUF	0000
NOT	0001

Tabelle 6.2: Kodierung der Boole'schen Funktionen

Die Auswertungsroutine mit Select-Instruktion für Gatterknoten ist in Programmauflistung 6.5 gegeben. Zuerst werden beide Eingangswerte gelesen. Anschließend werden für alle möglichen Funktionen die Ergebnisse berechnet. Danach filtern mehrere Select-Instruktionen das passende Ergebnis. Ein Puffergatterknoten ist als „Durchreichen“ durch alle Select-Instruktionen implementiert. Falls die Funktion eine Negation vorsieht, wird das vor der Speicherung des Werts ausgeführt.

Listing 6.5 Auswertungsroutine der Fehlersimulation

```
1 void fevald(GATEx gx, const int gid, level *values) {
2     // Laden aller Funktionen
3     GATEx op = gx.adr;
4     level in0 = values[op_in0];
5     level in1 = values[op_in1];
6
7     // Berechne alle Funktionen
8     level and_val = in0 & in1;
9     level or_val = in0 | in1;
10    level xor_val = in0 ^ in1;
11
12    // Filtere Wert
13    level val = in0;
14    val = op.f_and ? and_val : val;
15    val = op.f_or ? or_val : val;
16    val = op.f_xor ? xor_val : val;
17
18    // Negation
19    level not_val = ~val;
20    val = op.f_not ? not_val : val;
21
22    // Speichern
23    values[gid] = val;
24 }
```

Ein Problem ist das Lesen der Eingabestimuli. Diese Zugriffe liegen relativ zufällig im Speicherbereich der Gatterknotenwerte. In einer Implementierung wurden die Gatterknotenwerte in den Shared Memory kopiert. Dies führte allerdings zu Bankkonflikten, welche sich

mit einem vorherigen Scheduling reduzieren lassen. Das Scheduling kann dabei mehrere Freiheitsgrade ausnutzen.

Platzierung des Rangs Besteht für ein Gatterknoten beim Zugriff auf die Eingabestimuli ein Bankkonflikt mit einem anderen Gatterknoten, so kann einer dieser beiden unter Umständen ein anderer Rang zu gewiesen werden.

Halb-Warp-Tausch Ein Warp wird bei der Ausführung in zwei Halb-Warps aufgesplittet. Diese führen die gleichen Instruktionen aus, aber sind unabhängig beim Zugriff auf den Shared Memory. Haben zwei Gatterknoten in einem Halb-Warp einen Konflikt, so kann einer der Beiden in den anderen Halb-Warp verschoben werden.

Vorgängertausch Die beiden Eingabestimulis eines Gatterknotens werden nicht gleichzeitig gelesen. Wird die Lesereihenfolge der beiden Vorgänger getauscht, so kann ein Bankkonflikt unter Umständen vermieden werden.

Bei Anwendung einer der oben genannten Freiheitsgrade kann es zu weiteren Bankkonflikten kommen. Es handelt sich hierbei um ein Optimierungsproblem mit mehreren Variablen. Dieses ist entsprechend komplex und wurde hier nicht weiter realisiert bzw. untersucht.

Durch die Beschränkung auf den Shared Memory werden die Kegelgröße zu sehr beschnitten, so dass zu wenig Kegel vorhanden sind um alle Multiprozessoren der GPGPU auszulasten. Deshalb wurde dieses Konzept aufgegeben und ein Kompromiss der langsamen Zugriffen auf den Global Memory eingegangen. Es stellt sich heraus, dass dies letztendlich performanter ist. Das Multi-Threading von CUDA versteckt die zusätzlichen Latenzen des Speicherzugriffs effizienter.

Ein Vorteil der vollständigen Auswertung ist, dass die Funktion `restore_fault_free_values` aus Atalanta überflüssig ist. Die Verzweigungsstämme werden in der umgekehrten Reihenfolge ihres Ranges ausgewertet. So werden modifizierte Werte aus einer vorherigen Simulation neu ausgewertet und zurückgesetzt.

Optimierung der Registerauslastung

Als weiteres Problem stellte sich die Zahl der Register, die pro Thread genutzt wurde, heraus. Ein einzelner Thread allokierte bis zu 26 Register, so dass zu wenige Blöcke gleichzeitig ausgeführt werden konnten, denn der Registerpool wurde dadurch zu schnell aufgebraucht. Ein Teil der Register in einem Warp enthält redundante Daten, z.B. speichert jeder Thread die Anzahl der insgesamt vorhandenen Gatterknoten in einem Kegel oder an der Stelle, an denen eigentlichen Logikgatterknoten beginnen usw. Diese und weitere Informationen wurden in den Shared Memory ausgelagert, so dass innerhalb eines Warps die Daten nur ein einziges Mal gespeichert werden müssen. Dies hat den Nachteil, dass die Daten bei Verwendung erst aus dem Shared Memory explizit geladen werden müssen. Allerdings belegen sie bei Nichtbenutzung auch keine Register, so dass mehr Blöcke parallel ausgeführt werden konnten und die zusätzlichen Instruktionen durch Speicherzugriffe versteckt wurden.

Verschiebung von Gatterknoten im Kegel

Weitere Freiheitsgrade können ausgenutzt werden, um die Leistung zu verbessern. Eine Möglichkeit besteht in der Ausnutzung der Positionierung jener Gatterknoten, die nicht auf dem kritischen Pfad liegen. Solche Gatterknoten können in unterschiedlichen virtuellen Ebenen ausgewertet werden, speziell wirkt sich die Positionierung auf die Verzweigungsstämme außerhalb des kritischen Pfades aus. Da alle virtuellen Ebenen nach dem Verzweigungsstamm ausgewertet werden, sollte der Verzweigungsstamm in der letztmöglichen Ebene platziert werden. Verzweigungsfreie Regionen dahingegen sollten in die ersten virtuellen Ebenen gebracht werden.

Anhand des Beispiels in 6.2 wird dies verdeutlicht. Dort ist ein Schaltungsgraph mit drei Verzweigungsstämmen, welche orange eingefärbt sind, unter drei verschiedenen Positionierungsalgorithmen dargestellt. Bei dem ASAP-Ansatz (as soon as possible), der allen Gatterknoten den kleinstmöglichen Rang vergibt, müssen bei der Auswertung des Verzweigungsstamms im rechten Zweig alle Rot gefärbten Gatterknoten zusätzlich ausgewertet werden. In diesem Fall müssen darüber hinaus mehrere Ränge berechnet werden.

Wird dagegen der ALAP-Ansatz (as late as possible) genutzt, so reduziert sich die Zahl der unnötig ausgewerteten Gatterknoten auf ein einzelnes. Bei diesem Ansatz wird jedem Gatterknoten der größtmögliche Rang zugeordnet, ohne weitere Ränge zu erzeugen. Dieser Ansatz ist besser, allerdings auch nicht optimal, denn der Verzweigungsstamm im linken Zweig muss den Gatterknoten des rechten Zweigs mit gleichem Rang ebenfalls auswerten. Dieser liegt allerdings nicht im Ausgangskegel des simulierten Verzweigungsstamm und wird somit nicht durch die Fehlerinjektion beeinflusst.

Durch den Hybrid-Ansatz werden keine unnötigen Gatterknoten evaluiert. Dieser Ansatz ist eine Kombination aus dem ASAP- und dem ALAP-Ansatz. Für Verzweigungsstämme wird der ALAP-Ansatz genutzt und für die Gatterknoten in den verzweigungsfreien Gebieten der ASAP-Ansatz.

Kegelduplizierung

Um die volle Leistung der GPGPU zu erhalten, müssen sehr viele Threads vorhanden sein. Nun kann es passieren, dass durch die Wiederverwendung eines Kegels für mehrere Verzweigungsstämme zu wenige Stämme vorhanden sind und ein Teil der Leistung nicht genutzt wird. Die Kegel können wieder aufgebrochen und mehrere virtuelle Kegel erzeugt werden. Bei diesen Kegeln wird die Netzliste des ursprünglichen Kegels wiederverwendet.

Effizientes Speichern der Beobachtbarkeit

Aus der Fehlersimulation eines Verzweigungsstamms wird die Beobachtbarkeit abgeleitet. Diese ist als ein 32-Bit-Datenwort implementiert. Jeder der 32 Threads berechnet für einen

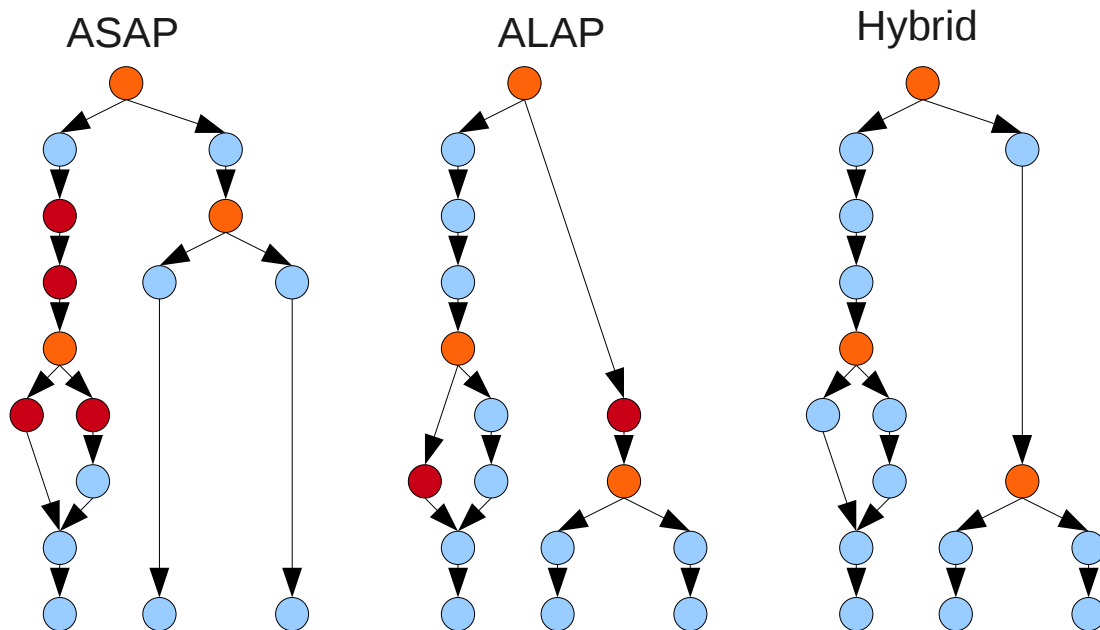


Abbildung 6.2: Verschiedene Ansätze zur Positionierung beweglicher Gatterknoten

Teil der Primärausgangsknoten die Beobachtbarkeit. Anschließend wird über den Shared Memory die Beobachtbarkeit ausgetauscht.

Das Datenwort wird im Global Memory gespeichert um später bei der Bestimmung der Beobachtbarkeiten in den verzweigungsfreien Gebieten und der Fehlerdetektion genutzt zu werden. Anstatt dieses nach jeder Auswertung zu speichern, werden mehrere Datenworte inklusive der Speicheradresse zwischengespeichert und in einem zusammenhängenden Schreibzugriff effizient abgelegt.

6.3 Fehlerauswertung

Nach der Verzweigungsstammsimulation wird abschließend der Graph rückwärts traversiert und dabei für die verzweigungsfreien Gebiete die Beobachtbarkeit festgestellt. Darüber hinaus werden basierend auf der Beobachtbarkeit die Fehler erkannt. Die Traversierung läuft dabei rückwärts, ähnlich wie die Gutsimulation ab. Ein Multiprozessor wertet mit 512 Threads rangweise alle Gatterknoten aus. Bei der GPGPU-Implementierung wertet ein Warp immer 32 nebeneinander liegende Gatterknoten aus, so dass Lese- und Schreibzugriffe für Gatterknoteninformationen, Gutwerte des auszuwertenden Gatterknotens und Beobachtbarkeiten auf den Global Memory zusammenhängend ausgeführt werden können. Dahingegen wertet ein Warp bei der ereignisgesteuerten Simulation immer nur einen Gatterknoten aus, so dass alle Zugriffe zusammenhängend erfolgen.

Für die Auswertung der Beobachtbarkeit werden die Formeln aus Tabelle 3.2 genutzt. Diese wurden wie die Gatterknotenauswertung so implementiert, dass sie ohne divergierende Sprünge auskommen.

Bei der Fehlererkennung, welche direkt nach der Auswertung der Beobachtbarkeit statt findet, wird für jeden Gatterknoten zuerst der Vektor der bereits erkannten Fehler geladen und geprüft, ob alle Fehler bereits erkannt wurden und eine weitere Fehlererkennung unnötig ist. Sind noch unerkannte Fehler vorhanden, so werden für diese Fehler die entsprechenden Bedingungen untersucht und falls der Fehler erkannt wird, wird er dem entsprechend markiert.

6.4 Scheduling

Durch die verkleinerten Kegel ist es möglich, die Gatterknoteninformationen so zu komprimieren, dass sie nur noch ein einzelnes Datenwort belegen. Dies optimiert den Zugriff auf die Informationen im Speicher, hat aber einen erhöhten Rechenaufwand für die Dekompression zur Folge.

Die Knoten eines Schaltungsgraphen lassen sich in drei Klassen einteilen. Quellen als Primäreingänge, Fluss als Logikgatterknoten und Senken als Primärausgänge. Für die Partitionen der Verzweigungsstämme entspricht dies:

Quellen Die Quellen bestehen aus dem eigentlichen Verzweigungsstamm sowie eingehenden Kanten in den Ausgangskegel. Für diese Knoten besteht die Auswertung darin, die Werte aus den Gutwerten zu kopieren. Eine Auswertung im eigentlichen Sinne findet nicht statt. Als Information zum Kopieren werden lediglich ein Zeiger in das Gutwertfeld benötigt. Das Ziel ergibt sich aus der aktuellen Position der Gatterknoteninformation im Kegel.

Fluss Die Gatterknoten im Fluss sind die eigentlichen Logikknoten. Sie nehmen einen oder mehrere Werte und transformieren diese in einen neuen Wert. Für diese Transformation werden Informationen benötigt, nämlich welche Operation auf welchen Operanden ausgeführt wird.

Senken Zum Schluss müssen die Beobachtbarkeitswerte errechnet werden. Diese lässt sich durch die Kontravalenz des Gut- und des Falschwerts erhalten.

Liegen im Ausgangskegel eines Verzweigungsstamms weitere Verzweigungsstämme und stehen darüber hinaus noch hinreichend viele weiteren Verzweigungsstämme zur Verfügung um alle Multiprozessoren vollständig auszulasten, so bietet es sich an nur einen Kegel für die Simulation zu erzeugen und die Gutwerte nur ein einziges Mal zu kopieren. Allerdings müssen dann die fehlerbehafteten Werte neu kopiert werden, weil die Simulation zu einem falschen Ergebnis kommen würde. Sind die Ausgangskegel von mehreren Verzweigungsstämmen paarweise disjunkt, so kann dieser Schritt entfallen. Ist die Schaltung ausreichend groß um die Multiprozessoren auszulasten, so lässt sich dieses Prinzip ausreizen und ausschließlich Primäreingänge als Startpunkte der Verzweigungsstammsimulation nutzen.

Auswertung

Im Verlauf dieses Kapitels wird die im Kapitel 6 vorgestellte Implementierung für GPGPUs mit den Implementierungen [GKo8], [LH91] verglichen. Zusätzlich wird auf die einzelnen Benchmark-Schaltungen eingegangen und untersucht, welche Merkmale zu welchen Auswirkungen bei der Implementierung führen.

7.1 Bisherige Ansätze

Für Vektor-[Ily90], [DÖ89], [NNN⁺94] und Parallelsysteme [MTSDA93], [NP92] wurden mehrere Ansätze zur Beschleunigung bereits vorgeschlagen. Allerdings gilt es für die GPGPUs die kleinen, lokalen Speicher zu beachten.

2008 ist auf der DAC¹ [GKo8] eine Implementierung der Fehlersimulation auf GPGPUs bereits vorgestellt worden. In diesem Paper wurde gezeigt, dass sich durch deren Implementierung die GPGPU gegen einen nicht weiter genannten kommerziellen Fehlersimulator behaupten kann. Es wurden keine algorithmischen Optimierungen vorgenommen, so dass dieser Ansatz alleine aus der Parallelisierung durch die GPGPU ihre Geschwindigkeitssteigerung erlangt.

Durch einen Vergleich der Laufzeiten von [GKo8] mit [LH91], wie in Tabelle 7.1 dargelegt, lässt sich schließen, dass die algorithmischen Optimierungen auf einem herkömmlichen Prozessor dennoch schneller sind als die parallele Auswertung durch die GPGPU aus [GKo8]. Lediglich bei zwei Schaltungen, nämlich `b17` und `b17_1`, ist [GKo8] schneller als [LH91].

Die ausgewählten Schaltungen in [GKo8] sind vergleichsweise klein. Das liegt unter anderem daran, dass [GKo8] sehr viel Speicher für die parallelen Auswertungen benötigt und daher nur kleine Schaltungen betrachtet werden können. Es ist allerdings zu bezweifeln, ob dieser Ansatz bei größeren Schaltungen skalieren würde.

Auf Grund dessen, dass [GKo8] langsamer ist als [LH91] und darüber hinaus nur für kleine Schaltungen geeignet, wird der Ansatz aus dieser Studienarbeit nicht mit [GKo8], sondern

¹<http://www.dac.com>

ausschließlich mit [LH91] verglichen. Dieser ist mit der Laufzeit von aktuellen kommerziellen Fehlersimulatoren vergleichbar.

Als Testumgebung wurde ein AMD Athlon 64 X2 2,4GHz mit 512 kB L2-Cache sowie einer NVIDIA GTX-285 eingesetzt. Der NVIDIA GTX-285-Grafikprozessor bietet dabei 30 Multiprozessoren und damit 240 Kerne sowie 1 GB Hauptspeicher. Alle Auswertungen beziehen sich ausschließlich auf den Fehlersimulationsteil ohne Einlesen der Schaltungsinformation und Aufbauen der Datenstrukturen. Dies liegt zum Einen daran, dass sich dieser Teil nicht durch die GPGPU effizient beschleunigen lässt, als auch daran, dass dies in [GKo8] ebenfalls vernachlässigt wurde.

7.2 Verzweigungsstammsimulation

In Tabelle 7.4 sind mehrere Merkmale, der als Benchmark-Schaltungen genutzten Schaltungsgraphen, aufgelistet. Darin sind neben der Zahl der Ein- und Ausgänge, die Zahl der äquivalenten Fehler, sowie die Zahl der Verzweigungsstämme sowie der Dominatoren gezeigt.

In Abbildung 7.2 sind die Laufzeiten von GPGPU-Implementierung sowie [LH91] und die erreichte Geschwindigkeitssteigerung für die Verzweigungsstammsimulation aufgelistet. Die durchschnittliche Beschleunigung beträgt 9.24 und zeigt damit, dass die Fehlersimulation durch Many-Core-Prozessor durchaus beschleunigt werden kann, auch gegenüber aktuellen Fehlersimulatoren.

Diese Implementierung wurde als Beschleuniger für einen bereits vorhandenen Fehlersimulator gebaut. Daher werten GPGPU und CPU beide gemeinsam das Problem der Fehlersimulation aus. Im besten Fall arbeiten beide Recheneinheiten genau die gleiche Zeit an ihrem Teilproblem. Notfalls muss das Teilproblem für eine Recheneinheit größer sein, als für die andere um Geschwindigkeitsvorteile auszugleichen. Dadurch wird die Rechenkapazität optimal ausgelastet.

Allerdings hängt die Geschwindigkeitssteigerung vom Aufbau der Schaltung ab. Folgende Merkmale schränken das Potential zur Beschleunigung in der GPGPU-Implementierung ein:

Ausgangskegel von Verzweigungsstämmen Sind die Ausgangskegel der Verzweigungsstämme sehr groß und passen nicht in die optimierten Datenstrukturen der GPGPU, so kann hier keine Beschleunigung statt finden. Dadurch werden dem Host weitere Kegel überlassen. Dies gilt z.B. für die Schaltung p469k.

Topologisch lange Pfade in Verzweigungsstämmen Topologische sehr lange Pfaden in den Verzweigungsstämmen führen zu einer erhöhten Ausführungszeit. Die einzelnen Ebenen der topologisch sortierten Kegel können nur seriell abgearbeitet werden. Sind diese nun dünn besetzt, so kann die Rechenleistung eines ganzen Warps unter Umständen nicht effizient genutzt werden.

Dominatoren In der Implementierung werden die Dominatoren nicht auf der GPGPU berechnet, sondern der CPU überlassen. Da dies relativ wenige sind, siehe dazu Tabelle 7.4, schränkt dies die Beschleunigung nicht maßgeblich ein, wurde aber Zwecks eines einfacheren Programmablaufs auf der GPGPU der CPU belassen.

Scheduling Das Scheduling hat eine starke Auswirkung auf die Ausführungszeit der Verzweigungstammsimulation, denn es müssen alle Gatterknoten in einem Kegel evaluiert werden. So ist das ASAP-/ALAP-Scheduling gut geeignet um bereits ausgewählte Kegel zu optimieren. Allerdings kann es bei Wiederverwendung eines Verzweigungstamms in einem Kegel zu ineffizienten Variationen kommen, z.B. ist es möglich, dass ein sehr kleiner Verzweigungstamm in einem sehr großen Kegel wieder benutzt wird. Liegt dieser kleine Verzweigungstamm nun relativ am Anfang des Kegels, so müssen alle Gatterknoten und vor allem Ebenen aus dem Kegel mit simuliert werden.

7.3 Beobachtbarkeit und Fehlererkennung

Wird die Betrachtung weiter detailliert und werden die Laufzeiten der einzelnen Kernel separiert, siehe Abbildung 7.3, so folgt daraus, dass aktuell die Verzweigungstammsimulation nicht mehr den Flaschenhals darstellt. Da in der aktuellen Implementierung nur ein einzelnen Multiprozessor mit 512 Threads die Gutsimulation und Beobachtbarkeit sowie Fehlerdetektion auswertet, kann hier durch eine triviale Parallelisierung über verschiedene Muster eine weitere Geschwindigkeitssteigerung erreicht werden. Konkret bedeutet dies, dass mehrere Gutsimulationen parallel von mehreren Multiprozessoren ausgewertet werden, und anschließend mehrere Verzweigungstammsimulationen seriell auf unterschiedlichen Gutwerten arbeiten, welche wieder von allen zur Verfügung stehenden Multiprozessoren bearbeitet wird. Dies wird in Abbildung 7.1 graphisch dargestellt. Abschließend wird die Beobachtbarkeit und Fehlerdetektion analog zur Gutsimulation ausgeführt. Für große Schaltungen stellt hier der nötige Speicherbedarf den Flaschenhals dar. Bei der Auswertung der Gutsimulation sowie Beobachtbarkeit und Fehlerdetektion können diese auch parallel zu einander ausgeführt werden, wie in einer Pipeline. Lediglich die erste Gutsimulation und die letzte Beobachtbarkeit und Fehlerdetektion müssen gesondert behandelt werden. Die Auswertung durch mehrere Multiprozessoren parallel skaliert anähernd linear, so dass sich eine große Leistungssteigerung für die Auswertung erreichen lässt.

7.4 Zusammenfassung

Es hat sich gezeigt, dass die Fehlersimulation mit Hilfe von GPGPUs beschleunigt werden kann. Der Fokus lag dabei auf der Optimierung der Verzweigungstammsimulation, welche nun im Laufzeitbereich der Gutsimulation bzw. der Berechnung der Beobachtbarkeit liegt. Diese beiden Funktionen lassen sich einfach beschleunigen, so dass eine weitere Beschleunigung um mindestens den Faktor zwei zu erwarten ist.

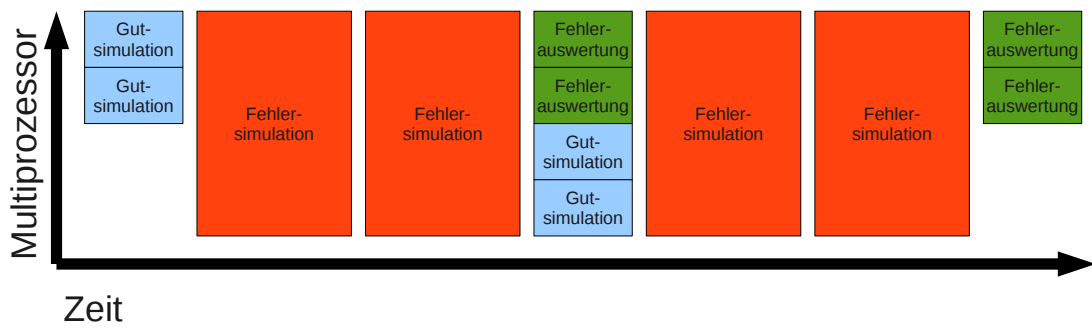


Abbildung 7.1: Parallele Auswertung von Gutsimulation und Fehlerauswertung

Schaltung	# Gatter	[LH91] [s]	[GKo8] [s]	Beschl.
s5378	1106	0.06	1.96	0.03
s9234.1	5944	1.69	2.04	0.83
s13207	1290	0.08	0.66	0.12
s15850	596	0.04	0.42	0.10
s35932	7152	0.03	5.43	0.01
s38417	10125	2.41	8.23	0.29
s38584	8571	0.54	7.88	0.07
b17	37446	49.35	19.03	2.59
b17_1	44544	51.63	17.87	2.89
b21	23100	11.94	46.58	0.26
b22	33569	19.81	60.49	0.33

Tabelle 7.1: Laufzeiten der Fehlersimulation aus [LH91] und [GKo8]

Schaltung	# Gatter	[LH91] [s]	GPGPU [s]	Beschl.
b17	37446	49.35	4.95	9.97
b17_1	44544	51.63	3.89	13.27
b18	130949	235.97	18.35	12.86
b19	263547	498.44	30.72	16.23
b20	22557	11.29	2.65	4.26
b21	23100	11.94	2.65	4.51
b22	33569	19.81	3.4	5.83
p100k	84356	44.07	6.22	7.09
p141k	152808	93.39	15.84	5.90
p239k	224597	94.43	13.79	6.85
p259k	298796	118.71	18.97	6.26
p267k	238697	133.13	12.46	10.68
p269k	239771	139.19	12.57	11.07
p279k	257736	173.99	19.34	9.00
p286k	332726	274.53	28.7	9.57
p295k	249747	225.96	14.38	15.71
p330k	312666	214.19	18.71	11.45
p388k	433331	223.73	29.57	7.57
p418k	382633	258.93	29.84	8.68
p469k	96408	2212.71	-	-
p483k	444664	249.33	25.65	9.72
p500k	431439	389.03	36.39	10.69
p533k	586819	333.74	37.67	8.86
p874k	629723	476.98	46.22	10.32
p951k	816072	327.31	59.49	5.50
p1522k	1104085	912.07	76.46	11.93
p2927k	2408328	2340.14	369.79	6.33
Avg.				9.24

Tabelle 7.2: Laufzeiten von [LH91] gegenüber Beschleunigung mit GPGPU

Schaltung	# Gatter	Init [sec]	FSIM [sec]	TM [%]	CP PI [%]	Gutsim [%]	CP GV [%]	VSsim [%]	Obs. [%]
b17	37446	14.420	10.43	0	0	16	4	66	12
b18	130949	44.750	36.67	0	0	12	4	74	9
b19	263547	42.320	187.23	0	0	16	6	64	11
b20	22557	7.130	3.38	0	0	25	6	49	19
b21	23100	7.340	3.48	0	0	24	6	49	19
b22	33569	10.380	4.55	0	0	24	6	49	18
p100k	84356	8.170	6.09	2	0	31	11	33	21
p141k	152808	39.720	15.6	1	0	24	12	44	17
p239k	224597	16.520	15.72	2	0	31	13	29	21
p259k	298796	22.440	19.66	2	0	32	13	28	22
p267k	238697	25.930	17.03	2	0	29	14	32	20
p269k	239771	26.450	17.78	2	0	28	15	33	19
p279k	257736	42.720	28.67	1	0	23	10	46	17
p286k	332726	56.230	41.75	1	0	25	11	43	17
p295k	249747	19.960	27.86	2	0	26	12	38	19
p330k	312666	31.110	24.76	1	0	27	12	40	17
p388k	433331	41.150	34.85	1	0	28	12	38	19
p418k	382633	38.600	39.25	2	0	30	13	32	20
p483k	444664	29.050	33.52	3	0	32	15	23	24
p500k	431439	39.430	73.9	2	0	25	11	40	19
p533k	586819	38.800	75.6	2	0	30	14	29	22
p874k	629723	59.840	76.29	3	0	27	12	34	21
p951k	816072	50.480	86.17	4	0	32	16	19	25
p1522k	1104085	95.180	214.0	2	0	31	14	29	20
p2927k	2408328	153.180	1734.64	2	0	37	18	12	29

Tabelle 7.3: Laufzeitanteile der GPGPU-Implementierung

7 Auswertung

Schaltung	Gatter	Eingänge	Ausgänge	Fehler	Stämme	Dominatoren
b17	37446	1452	1512	81330	8145	760
b17_1	44544	1452	1512	92794	8767	1746
b18	130949	3357	3364	277976	31044	3194
b19	263547	6666	6713	560696	63050	6194
b20	22557	522	512	47376	4645	969
b21	23100	522	512	48182	4613	982
b22	33569	767	757	70464	6876	1286
p100k	84356	5902	5829	166960	18486	2507
p141k	152808	11290	10502	287552	31083	9365
p239k	224597	18692	18495	455992	46465	4035
p259k	298796	18713	18495	607536	65450	21154
p267k	238697	17332	16621	372140	38705	4124
p269k	239771	17333	16621	374296	38706	4124
p279k	257736	18074	17827	493744	52393	3879
p286k	332726	18351	17835	648044	71449	19807
p295k	249747	18508	18521	478996	41705	3298
p330k	312666	18010	17468	547808	56441	9061
p388k	433331	25005	24065	856678	94439	24595
p418k	382633	30430	29809	688808	72481	7915
p469k	96408	635	403	169364	14286	2620
p483k	444664	33264	32610	922950	106598	23756
p500k	431439	30768	30840	843286	92303	9067
p533k	586819	33373	32610	1221432	141450	51270
p874k	629723	61977	70863	1050534	103017	8858
p951k	816072	91994	104714	1590490	168494	10492
p1522k	1104085	71392	68035	1747416	140840	12159
p2927k	2408328	101844	95159	3593886	708825	125450

Tabelle 7.4: Schaltungsmerkmale

Zusammenfassung

In der hier erarbeiteten Studienarbeit wurde gezeigt, dass sich GPGPUs zur Beschleunigung von Fehlersimulation nutzen lassen und dass die algorithmischen Optimierungen für herkömmliche Prozessoren auch auf GPGPUs umsetzbar sind. Die durchschnittliche Beschleunigung um circa den Faktor neun demonstriert, dass sich diese Abbildung effizient nutzen lässt.

Bei den GPGPUs und Many-Core-Prozessoren stehen als nächste Entwicklungsschritte NVIDIA Fermi und Intel Larrabee[SCS⁺08] auf der Agenda. Der Vorteil dieser beiden Ansätze ist, dass Zwischenspeicher für den Hauptspeicherzugriff eingeführt werden. Dies verbessert die Latenz beim Zugriff auf den Hauptspeicher und die Speicherzugriffsmuster bei der GPGPU-Implementierung sind relativ gut vorraussagbar, da zum Einen die Werte aus einem kleinem räumlichen Bereich, dem Kegel, gelesen werden und zum Anderen wird durch die topologisch sortierte Auswertung ein Wert immer aus der vorherigen Ebene gelesen, welcher zeitnah zuvor ausgewertet wurde. Somit sind zeitliche wie auch räumliche Lokalitäten ausnutzbar.

Darüber hinaus werden sich die Hauptspeichergröße und die Anzahl der zur Verfügung stehenden Recheneinheiten weiter erhöhen. Beide Größen ändern sich für diese Implementierung positiv, denn durch mehr Hauptspeicher können mehr Kegel gespeichert werden und durch mehr Recheneinheiten können mehr Kegel parallel ausgewertet werden.

An vielen Stellen der Implementierung könnten noch weitere architekturenspezifische Optimierungen, wie z.B. Shared Memory in der Gutsimulation genutzt werden. Durch diese Optimierungen könnten weitere Leistungssteigerungen erreicht werden.

Es wurde in dieser Studienarbeit mit dem einfachen Haftfehlermodell gearbeitet. Durch dieses Fehlermodell konnten gewisse Eigenschaften ausgenutzt werden, welche z.B. im PPSFP-Algorithmus realisiert wurden. Würden andere Fehlermodelle betrachtet werden, so müssen auf einige Optimierungen verzichtet werden. Durch andere Modelle ergeben sich andere Optimierungsverfahren.

Ereignisgesteuerte Methode

Als Alternative zu der in Abschnitt 6.2 beschriebenen Fehlersimulation, soll hier kurz eine mögliche Abbildung des ereignisgesteuerten Ansatzes auf die GPGPU beschrieben werden.

Um die Vorteile der ereignisgesteuerten Fehlersimulation zu nutzen, wird eine Methode zum Verwalten der Ereignisse benötigt. Im Atalanta Fehlersimulator gibt es hierfür eine Warteschlange für jeden Rang. So ist es möglich, effizient die Ereignisse topologisch sortiert auszuwerten. Auf der GPGPU ist diese Form der Verwaltung nicht effizient möglich, denn ein Warp führt 32 Threads aus, welche gemeinsam die Zugriffe auf die entsprechenden Datenstrukturen synchronisieren müssen.

Seit CUDA 1.2 gibt es Instruktionen um atomare Speicheroperationen auszuführen. Diese haben aber einen entsprechenden Mehraufwand durch Sperren zur Auflösung von Zugriffskonflikten zur Folge, welcher die Leistung einschränkt. Aus diesem Grund sollten die atomaren Operationen nicht in der kritischen inneren Schleife genutzt werden.

A.1 Parallelisierung

Bevor die ereignisgesteuerte Simulation implementiert wurde, stellte sich die Frage, wie sich die Parallelität eines Warps nutzen lässt. Folgend sind drei Methoden aufgelistet:

32 Gatterknoten Dieser Ansatz wurde in der Implementierung aus Kapitel 6.2 gewählt. Er wertet 32 Gatterknoten in einer virtuellen Ebene von gleichen Rängen aus. Ein Vorteil ist, dass Ausgangswerte zusammenhängend geschrieben werden können, aber die Eingangswerte müssen zufällig gelesen werden. Um divergierende Sprünge zu vermeiden, ist ein erhöhter Auswertungsaufwand durch die Select-Instruktion nötig.

1024 Muster parallel Mit dieser Methode werden für jeden Gatterknoten 32x 32 Muster ausgewertet, entsprechend 32fach SIMD. Ein Warp wertet damit nur einen einzelnen Gatterknoten aus. Diese Variante kann Ein- und Ausgangswerte zusammenhängend lesen bzw. schreiben. Alle Threads werten den selben Gatterknoten aus, so dass es zu keinen divergierenden Sprüngen kommt. Zur Auswertung auf dem Host sollten die Daten konvertiert werden, da sonst die Caches nicht effizient arbeiten können.

128 Muster parallel In dieser Variante werden mehrere Verzweigungstämme parallel ausgewertet, wobei vier Threads jeweils 128 Muster eines Verzweigungsstamms auswerten. Durch Nutzung von SIMD kann diese Darstellung auch auf dem Host direkt genutzt werden. Es entfällt die Konvertierung zwischen GPGPU- und Host-Kodierung, wie es bei 1024 Mustern nötig ist. Die Eingangswerte werden zufällig verteilt mit 128 Muster gelesen. Dies ist nicht so effizient wie mit 1024 zusammenhängenden Mustern, aber effizienter als nur 32 Muster zu lesen. Alle Ausgangswerte können zusammenhängend geschrieben werden. Die virtuellen Ränge können Vielfache von acht statt von 32 sein. Da in einem Warp unterschiedliche Funktionen ausgewertet werden können, wird, wie bei 32 parallelen Gatterknoten, wieder ein erhöhter Auswertungsaufwand durch die Select-Instruktion nötig.

In dieser Arbeit wurde der Ansatz mit 1024facher paralleler Musterauswertung gewählt, weil dies keine Modifikation am Algorithmus von Atalanta zur Folge hat. Deshalb wird nun näher auf die Speicherung der Muster eingegangen. Diese liegen auf der GPGPU als ein Kilobit Block pro Gatterknoten nebeneinander. Dadurch wird es möglich alle Muster zusammenhängend zu lesen und zu schreiben. Zufällige Zugriffe auf den Speicher beim Lesen der Eingabewerte werden hiermit effizient ausgeführt.

A.2 Kantensimulation

Im Gegensatz zur Implementierung aus Kapitel 6.2, welcher für jeden Gatterknoten die beiden Eingangskanten gespeichert hat, benötigt die ereignisgesteuerte Simulation die Ausgangskanten für jeden Gatterknoten. Dies ist notwendig um Ereignisse für die Nachfolgerknoten zu verzeichnen. Das hat den Nachteil, dass die Datenstruktur der Gatterknoteninformation nun nicht mehr statisch ist, denn bei Verzweigungsstämmen muss mehr als eine Ausgangskante gespeichert werden. Das Problem wird dadurch gelöst, dass keine Informationen über die zweite Eingangskanten gespeichert werden. Die neue Datenstruktur GATEx aus Programmauflistung A.1 stellt nur noch eine Eingangs- und eine Ausgangskante, sowie die Funktion des Gatterknotens und ob es der letzte Gatterknoten in einem Verzweigungsstamm ist bereit. EOG ist wahr, wenn es der letzte Gatterknoten eines Verzweigungsstamms oder gar kein Verzweigungsstamm ist.

Listing A.1 Datenstruktur einer Kante

```
1 typedef struct GATEx {
2     unsigned pad:1; // Fuellen auf 32bit
3     unsigned eog:1; // Gatterende
4     unsigned fn:3; // Funktion
5     unsigned fn_inv:1; // Funktion invertieren?
6     unsigned in:13; // Eingangskante
7     unsigned out:13; // Ausgangskante
8 };
```

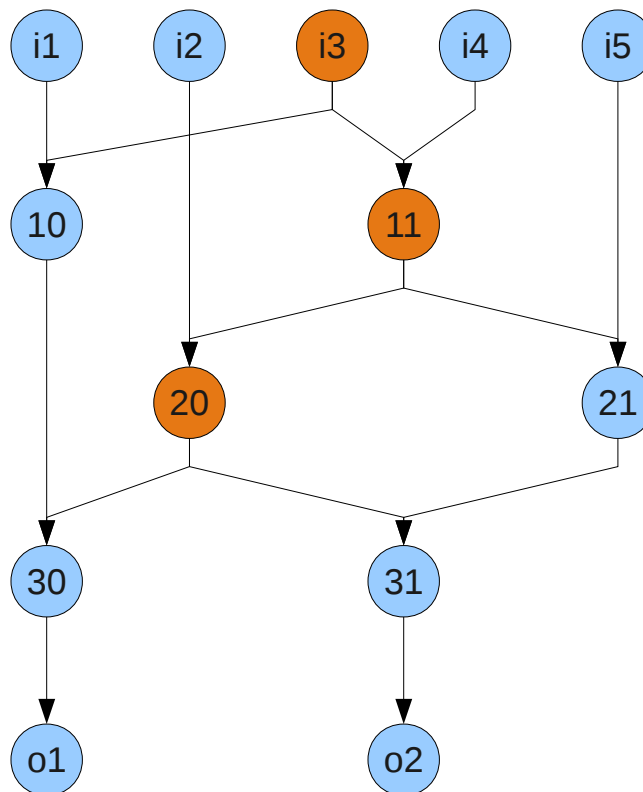
Die Datenstruktur GATEx besitzt nur einen Eingang. Statt wie bisher für einen Knoten die beiden Eingangskanten zu speichern, wird nun für jede Ausgangskante eines Knoten

die zugehörige zweite Eingangskante des Zielknotens gespeichert. Die Quelle der eigentlichen Eingangskante entspricht dabei genau der aktuellen Position in der Netzliste. Die temporären Werte für die Verzweigungsstammsimulation weist dadurch an den Stellen eines Verzweigungsstamms, wo weitere Kanten gespeichert werden, Lücken auf.

Es findet sich in Tabelle A.1 eine Übersicht über die Belegung der Datenstruktur. Als Beispielgraph wurde C17 benutzt, welcher in Abbildung A.1 dargestellt ist.

Name	In	Out	EOG
i1	i3	10	1
i2	11	20	1
i3	i1	10	0
	i4	11	1
i4	i3	11	1
i5	11	21	1
10	20	30	1
11	i2	20	0
	i5	21	1
20	10	30	0
	21	31	1
21	20	31	1
30	-	01	1
31	-	i2	1

Tabelle A.1: Repräsentation der Netzliste C17 für die ereignisgesteuerte Simulation

Abbildung A.1: Schaltungsgraph C₁₇

Ein Nachteil der Kantensimulation ist, dass alle Ereignisse simuliert werden müssen. Dies bedeutet, dass an einem Gatterknoten mit zwei Eingängen auch zwei Auswertungen statt finden können. Dies führt zu einem erhöhten Aufwand. Unter der Annahme, dass letztendlich weniger Kanten durch die ereignisgesteuerte Simulation ausgewertet werden müssen, sollte es dennoch effizienter sein.

A.3 Verwaltung von Ereignissen

Die Datenstruktur zum Speichern der Ereignisse besteht aus einem einfachen Bitfeld, welches so groß ist, wie das Maximum der Kegelgröße sein kann. Für jeden Knoten existiert ein Eintrag in diesem Feld, welcher angibt, ob ein Ereignis propagiert wurde oder nicht. Das Bitfeld wird nicht wort-adressiert sondern tatsächlich bit-adressiert. Dadurch wird das Feld um den Faktor der Maschinenwortbreite komprimiert. Die nötigen Umwandlungsschritte von Wort- nach Bitadressierung werden mit Hilfe der Konjunktion mit einem nach rechts geschobenen ALL1-Vektor sowie der Count-Leading-Zeroes (CLZ)-Instruktion realisiert. Da alle Threads die gleiche Kante betrachten und nur eine Eins geschrieben wird, wenn ein Ereignis

propagiert wird, kann es nicht passieren, dass ein Ereignis zwar in einem Thread propagiert, aber durch einen anderen Thread wieder auf Null gesetzt wird. Tatsächlich wird das Setzen nur von einem einzelnen Thread im Warp ausgeführt. Durch die Votier-Funktionen von CUDA 1.2 können sich alle Threads innerhalb des Warps darüber austauschen, ob ein Ereignis propagiert wurde oder nicht.

Nachteilig wirkt sich das Bitfeld bei wenigen Ereignissen aus, da es zuerst komplett abgesucht werden muss um festzustellen, dass keine Ereignisse mehr vorhanden sind. Dem lässt sich dadurch entgegenwirken, dass jeder Thread ein Wort des weiteren Bitfelds voraus schaut und durch die Votierfunktion gleichzeitig für 1024 Ereignisse in die Zukunft geschaut wird. Die Votierfunktion kann auch genutzt werden, um eine parallele binäre Suche zu implementieren, so dass das Bitfeld nicht vollständig sequentiell durchsucht werden muss.

A.4 Beobachtbarkeit

Die Beobachtbarkeit besteht aus einem 1024-Bit-Vektor. Dieser lässt sich effizienter berechnen als in der Implementierung aus Kapitel 6.2, bei der ein Austausch über den Shared Memory statt findet. Jeder Thread bildet die Disjunktion der Beobachtbarkeit aller Primärausgänge. Das Speichern in den Global Memory erfolgt ebenfalls zusammenhängend. Die Auswertung findet später in einem separaten Kernel statt.

A.5 Gutwerte wiederherstellen

Durch die ereignisgesteuerte Simulation ist es nicht mehr möglich, die temporär fehlerhaften Werte im entsprechenden Speicherfeld zu lassen und später zu überschreiben, denn es werden nicht alle Gatterknoten neu evaluiert. Dadurch würde es zu Fehlern in der Simulation kommen. Statt dessen wird ein Fehler, durch Invertierung, am Fehler erneut injiziert, also der Gutwert am Verzweigungsstamm wiederhergestellt und eine weitere Simulation gestartet. Dadurch werden die ursprünglichen Gutwerte wieder hergestellt, ohne das der ganze Kegel erneut kopiert werden muss.

Approximative Fehlersimulation

Bei der approximativen Fehlersimulation handelt es sich um ein beschleunigtes Verfahren der Fehlersimulation. Dabei wird einer schnelleren Auswertung Vorzug gegenüber einer genauen Fehlersimulation gegeben. Dies kann z.B. für eine erste grobe Einschätzung der Güte von einer gegebenen Testmuster Menge genutzt werden.

B.1 Funktionsweise

Statt der aufwendigen Auswertung der Beobachtbarkeit jedes einzelnen Verzweigungsstamms durch Fehlerinjektion und Logiksimulation wird eine Abschätzung der Beobachtbarkeit für jeden Verzweigungsstamm vorgenommen. Diese Abschätzung besteht darin, die Beobachtbarkeiten der Nachfolgegatter zu übernehmen. Hierfür wurden zwei verschiedene Ansätze von [WW92] vorgeschlagen:

Optimistischer Fall Der Verzweigungsstamm VS ist beobachtbar genau dann, wenn *mindestens* eines der Nachfolgegatter beobachtbar ist.

$$O_{VS} = \forall s \in successors(VS) : \exists O_s$$

Pessimistischer Fall Der Verzweigungsstamm VS ist beobachtbar genau dann, wenn *exakt* eines der Nachfolgegatter beobachtbar ist.

$$O_{VS} = \forall s \in successors(VS) : \exists! O_s$$

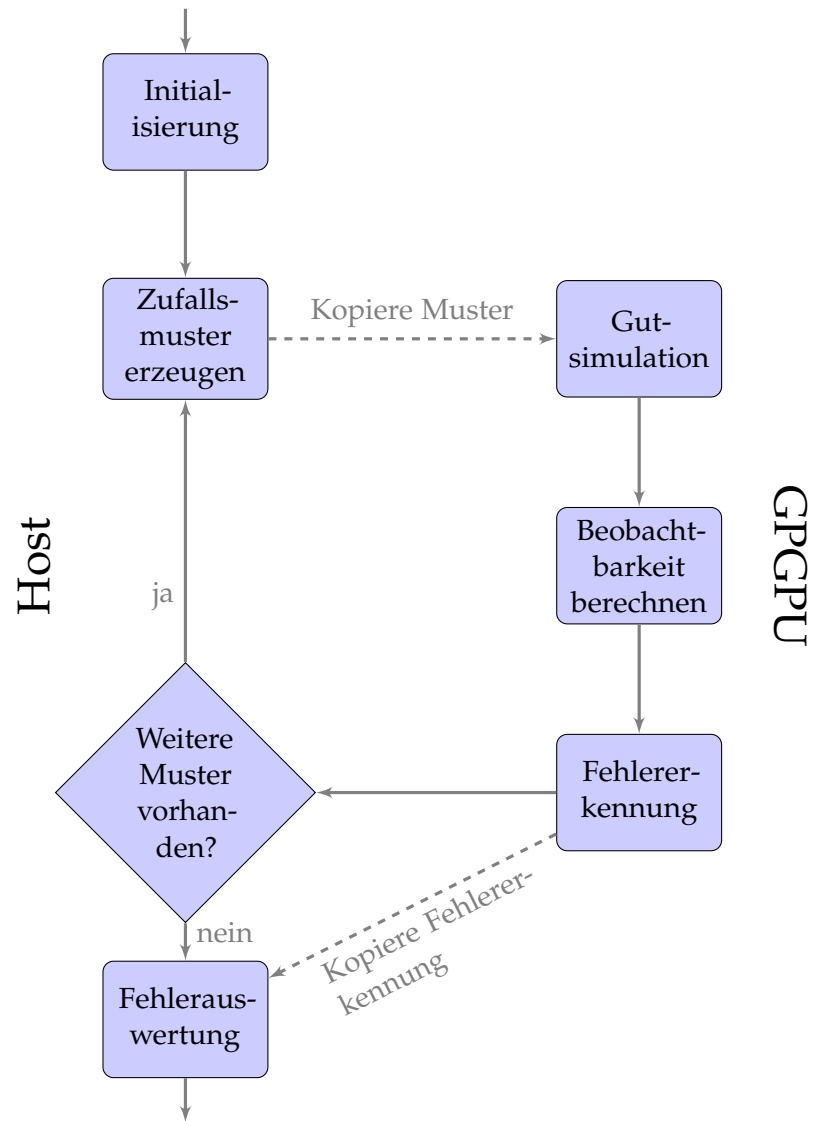


Abbildung B.1: Ablaufdiagramm approximative Fehlersimulation

Der approximative Algorithmus sollte sich sehr effizient auf der GPGPU realisieren lassen. Die Zahl der parallelen Auswertungen kann, wie bei der bisherigen ereignisgesteuerten Implementierung, mit 1024 Mustern pro Gatter beibehalten werden. Durch den Wegfall der Verzweigungsstammsimulation, können die Multiprozessoren auf mehrere Muster aufgeteilt werden, sofern genug Speicher vorhanden ist

Die Fehlerentdeckung kann mit dem bisherigen Schema allerdings nicht effizient implementiert werden. Wenn die Fehler eines Gatterknotens von mehreren Threads parallel bearbeitet werden, so müssen diese Threads entweder ihren Zugriff vorher synchronisieren oder mit atomaren Speicher-Operationen arbeiten. Beides führt zu einem bedeutenden Mehraufwand.

Statt dessen wurde ein anderer Ansatz implementiert, welcher für alle Fehler eines Gatterknotens genau einen Thread benutzt. Dadurch kommt es bei der Auswertung zu keinen Berechnungskonflikten. Um die Geschwindigkeit der Fehlerauswertung weiter zu steigern, können die Threads innerhalb eines Warps alle Werte gemeinsam für alle Threads zusammenhängend lesen und in den Shared Memory auslagern. Danach nutzt jeder Thread die Daten aus dem Shared Memory, die er benötigt. So ist es möglich, dass die einzelnen Threads keine Leistungseinbusen durch Lesen von lediglich 32 Bit Datenworten aus dem Speicher aufweisen.

Wie bisher werden nur die Eingangskanten der Gatterknoten gespeichert, so dass die Datenstruktur der Gatterknoten immer eine fixe Größe hat. Bei der Auswertung der Beobachtbarkeit wurden in der PPPSF-Implementierung nur die verzweigungsfreien Gebiete betrachtet. In diesen haben alle Gatterknoten nur einen Nachfolgerknoten, so dass nur ein Thread die Beobachtbarkeit propagieren kann. Nimmt man nun die Verzweigungsstämme hinzu, so können mehrere Threads gleichzeitig die Beobachtbarkeit auswerten und propagieren. Diese Schreibkonflikte werden mit atomaren Speicher-Operationen gelöst.

Ein weiteres Problem entsteht dadurch, dass nun die Beobachtbarkeiten der Verzweigungsstämme zurückgesetzt werden müssen, da diese nicht mehr explizit durch die Verzweigungsstammsimulation gesetzt werden. Um dies möglichst effizient zu implementieren, wird eine weitere Liste mit den Verzweigungsstämmen benötigt, so dass die Beobachtbarkeit von Gatterknoten in den verzweigungsfreien Gebieten nicht unnötig gesetzt werden. Diese werden ohnehin durch späteres Rückwärtstraversieren des Graphen gesetzt.

B.2 Auswertung

Die Laufzeiten der Verzweigungsstammsimulation wurden in Tabelle B.1 mit der Laufzeit der optimistischen und pessimistischen Fehlersimulation verglichen. Wie zu erwarten war, ist die approximative Fehlersimulation schneller als die explizite Verzweigungsstammsimulation. Die größten Leistungssteigerungen treten bei Schaltungen mit hohem Verzweigungsstammsimulationsanteil auf.

Wie sich herausgestellt hat, ist das größte Problem in der aktuellen Implementierung, der massive Speicherverbrauch. Durch die Nutzung von 1024 Mustern pro Gatter wird die Größe

der Gutwerte und Beobachtbarkeiten um den Faktor 32 vergrößert. Darüber hinaus wird der Speicherverbrauch weiter vergrößert, weil jeder Block ebenfalls eine eigene Kopie der Gutwerte und der Beobachtbarkeiten benötigt. Bei der genutzten GTX-285 GPGPU sind dies mindestens 60 Blöcke. Für jeden Multiprozessor werden zwei Blöcke mit jeweils 512 Threads gestartet, damit eine maximale Auslastung erreicht wird. Insgesamt ergibt sich der Speicherverbrauch für das Gutwert- sowie Beobachtbarkeitsfeld für die approximative Fehlersimulation zu $S = |G| * 2 * |MP| * 32$. Dabei liegt G in einer Größenordnung von hundertausend bis zehn Millionen und MP im Bereich von 20 bis 30.

In Tabelle B.2 ist unter Anderem der Speicherverbrauch aufgelistet. Dieser liegt bei einer großen Schaltung wie „p1522k“ bereits bei 3,4 GB. Die größten derzeit erhältlichen CUDA-fähigen GPGPUs stammen aus der NVIDIA High-Performance-Computing-Serie Tesla[LNOMo8]. Diese besitzen bis zu 4GB RAM womit genug Speicher für diese Auswertung zur Verfügung stand, ohne die Zahl der Prozessoren zu sehr einzuschränken. Die Ergebnisse aus Tabelle B.1 wurden auf einer entsprechenden GPGPU, Tesla C1060, erstellt. Die Laufzeitanteile sollten auf GPGPUs mit weniger Multiprozessoren oder Speicher ähnlich sein.

Aus Tabelle B.2 ergibt sich, dass die Berechnung der Beobachtbarkeit in der Größenordnung der Fehlerentdeckung liegt. Das Erzeugen der Zufallsmuster nimmt mit bis zu 25% einen erhebliche Anteil ein. Das Kopieren der Zufallsmuster vom Host auf die GPGPU hat einen zu vernachlässigbaren Anteil. Die Anteile der Erzeugung und Kopieren der Zufallsmuster kann aber vernachlässigt werden, da diese Parallel zur Berechnung auf der GPGPU auf dem Host ausgeführt werden können.

Allerdings ist die Speicherbandbreite ein weiterer limitierender Faktor. Eine Reduktion der Zahl der gleichzeitig ausgeführten Blöcke um den Faktor vier, d.h. es werden halb so viele Blöcke gestartet wie es Multiprozessoren gibt, reduziert die Laufzeit der approximativen Fehlersimulation nicht. Bei der Auswertung von Tabelle B.1 wurde diese Optimierung bereits angewandt, da sonst der Hauptspeicherbedarf um einen Faktor vier größer gewesen wäre und viele Schaltungen zuviel Hauptspeicher benötigt hätten.

Schaltung	# Gatter	Akkurat [s]	Optimistisch		Pessimistisch	
			[s]	Beschl.	[s]	Beschl.
b17	37446	10.43	0.633	16.48	0.667	15.64
b18	130949	36.67	2.05	17.89	2.1	17.46
b19	263547	187.23	4.117	45.48	4.767	39.28
b20	22557	3.38	0.3	11.27	0.333	10.15
b21	23100	3.48	0.3	11.60	0.333	10.45
b22	33569	4.55	0.45	10.11	0.467	9.74
p100k	84356	6.09	1.283	4.75	1.3	4.68
p141k	152808	15.6	2.35	6.64	3.617	4.31
p239k	224597	15.72	3.667	4.29	4.95	3.18
p259k	298796	19.66	4.35	4.52	5.55	3.54
p267k	238697	17.03	3.35	5.08	4.667	3.65
p269k	239771	17.78	3.433	5.18	3.45	5.15
p279k	257736	28.67	4.05	7.08	4.067	7.05
p286k	332726	41.75	5.15	8.11	5.117	8.16
p295k	249747	27.86	4.367	6.38	4.317	6.45
p330k	312666	24.76	4.55	5.44	4.567	5.42
p388k	433331	34.85	5.933	5.87	6.133	5.68
p418k	382633	39.25	6.15	6.38	6.167	6.36
p483k	444664	33.52	6.75	4.97	6.9	4.86
p500k	431439	73.9	6.733	10.98	6.767	10.92
p533k	586819	75.6	8.25	9.16	8.267	9.14
p874k	629723	76.29	12.05	6.33	14.833	5.14
p951k	816072	86.17	26.533	3.25	24.7	3.49
p1522k	1104085	214.0	23.65	9.05	22.433	9.54

Tabelle B.1: Laufzeiten der exakten und der approximativen Simulation

B Approximative Fehlersimulation

Schaltung	# Gatter	RAM [MB]	Muster [%]	Speicher- transfer [%]	Gut- simulation [%]	Beobacht- barkeit [%]	Fehler- erkennung [%]
b17	37446	113	9	2	14	37	36
b18	130949	385	7	1	13	38	39
b19	263547	775	6	3	17	39	38
b20	22557	65	6	2	17	46	26
b21	23100	66	7	2	16	46	27
b22	33569	96	7	3	16	46	26
p100k	84356	273	21	2	16	40	18
p141k	152808	494	15	5	23	34	24
p239k	224597	747	18	7	21	29	14
p259k	298796	953	17	5	23	33	13
p267k	238697	760	17	8	22	26	15
p269k	239771	763	22	2	27	36	21
p279k	257736	831	21	1	25	36	25
p286k	332726	1040	16	1	15	38	27
p295k	249747	802	19	1	14	35	28
p330k	312666	970	18	1	17	36	25
p388k	433331	1363	19	3	17	40	18
p418k	382633	1253	22	2	15	33	25
p500k	431439	1402	22	1	15	35	23
p533k	586819	1861	18	1	17	43	18
p874k	629723	2167	45	1	11	22	17
p951k	816072	2941	58	3	9	19	9
p1522k	1104085	3406	42	1	11	25	19

Tabelle B.2: Laufzeitanteile der GPGPU-Kernel an der approximative Fehlersimulation

Literaturverzeichnis

- [AMM83] M. Abramovici, P. R. Menon, and D. T. Miller. Critical path tracing - an alternative to fault simulation. In *DAC '83: Proceedings of the 20th Design Automation Conference*, pages 214–220, Piscataway, NJ, USA, 1983. IEEE Press. (Zitiert auf Seite 15)
- [BA05] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, 2005. (Zitiert auf den Seiten 7 und 12)
- [BHK92] Bernd Becker, Ralf Hahn, and Rolf Krieger. Fast fault simulation in combinatorial circuits: an efficient data structure, dynamic dominators and refined check-up. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 436–441, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. (Zitiert auf Seite 5)
- [DÖ89] Raja Daoud and Füsün Özgüner. Highly vectorizable fault simulation on the Cray X-MP supercomputer. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(12):1362–1365, 1989. (Zitiert auf Seite 47)
- [GK08] Kanupriya Gulati and Sunil P. Khatri. Towards acceleration of fault simulation using graphics processing units. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 822–827, 2008. (Zitiert auf den Seiten 47, 48, 51 und 73)
- [HSU86] D. Harel, R. Sheng, and J. Udell. Efficient single fault propagation in combinatorial circuits. *Communications of the ACM*, 29(4):300–311, 1986. (Zitiert auf Seite 18)
- [IYY90] Nagisa Ishiura, Masyuki Ito, and Shuzo Yajima. Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 9(8):868–875, 1990. (Zitiert auf Seite 47)
- [LH91] Hyung Ki Lee and Dong Sam Ha. An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation. In *Proceedings IEEE International Test Conference 1991, Nashville, TN, USA, October 26-30, 1991*, pages 946–955, 1991. (Zitiert auf den Seiten 19, 47, 48, 51, 52 und 73)

- [LNOMo8] Erik Lindholm, John Nickolls, Stuart F. Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008. (Zitiert auf Seite 66)
- [MR90] Fadi Maamari and Janusz Rajski. A method of fault simulation based on stem regions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 9(2):212–220, 1990. (Zitiert auf Seite 18)
- [MTSDA93] Robert B. Mueller-Thuns, Daniel G. Saab, Robert F. Damiano, and Jacob A. Abraham. VLSI logic and fault simulation on general-purpose parallel computers. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(3):446–460, 1993. (Zitiert auf Seite 47)
- [NNN⁺94] Takaharu Nagumo, Masahiko Nagai, Takao Nishida, Masayuki Miyoshi, and Shunsuke Miyamoto. VFSIM: Vectorized fault simulator using a reduction technique excluding temporarily unobservable faults. In *Proceedings of the 31st Conference on Design Automation (DAC), San Diego, California, USA, June 6-10, 1994*, pages 510–515, 1994. (Zitiert auf Seite 47)
- [NP92] Vinod Narayanan and Vijay Pitchumani. Fault simulation on massively parallel SIMD machines algorithms, implementations and results. *J. Electronic Testing*, 3(1):79–92, 1992. (Zitiert auf Seite 47)
- [NVI] NVIDIA. *NVIDIA CUDA – Programming Guide 2.3*. (Zitiert auf Seite 30)
- [Sch88] Michael H. Schulz. *Testmustererzeugung und Fehlersimulation in digitalen Schaltungen mit hoher Komplexität*, volume 173 of *Informatik-Fachberichte*. Springer, 1988. (Zitiert auf Seite 7)
- [SCS⁺08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM. (Zitiert auf Seite 55)
- [WEF⁺85] J.A. Waicukauski, E.B. Eichelberger, D.O. Forlenza, E. Lindbloom, and T. McCarthy. Fault simulation for structured VLSI. *VLSI Systems Design*, 6(12):20–32, 1985. (Zitiert auf Seite 13)
- [WW92] H.J. Wunderlich and M. Warnecke. Efficient test set evaluation. In *Design Automation, 1992. Proceedings.[3rd] European Conference on*, pages 428–433, 1992. (Zitiert auf Seite 63)
- [WWW06] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. (Zitiert auf Seite 5)

Abbildungsverzeichnis

2.1	Stuck-At-o Fehler am Ausgang eines Und-Gatters	8
2.2	Beispiel zum Fehlermodell	9
2.3	Beispiel zur Fehleräquivalenz	11
3.1	Ablaufdiagramm des PPFSP-Algorithmus	14
3.2	Beispiel der Beobachtbarkeit eines Schaltung mit wenigen Gattern.	16
4.1	Aufrufgraph – Atalanta Fehlersimulator	22
4.2	Profiling – Genutzter CPU-Zyklen-Anteil	23
4.3	Profiling – L1-Data-Cache Miss Rate	24
4.4	Profiling – L2-Cache Miss Rate	25
5.1	Aufbau der Architektur einer CUDA GPGPU	29
6.1	Ablaufdiagramm GPGPU-Simulation	38
6.2	Verschiedene Ansätze zur Positionierung beweglicher Gatterknoten	45
7.1	Parallele Auswertung von Gutsimulation und Fehlerauswertung	50
A.1	Schaltungsgraph C17	61
B.1	Ablaufdiagramm approximative Fehlersimulation	64

Tabellenverzeichnis

3.1	Rekursives Berechnungsschema für die Beobachtbarkeit	15
5.1	Vergleich von AMD Athlon64 X2 4600+ und NVIDIA GTX-285	32
6.1	Datenstruktur eines Gatterknotens	41
6.2	Kodierung der Boole'schen Funktionen	42
7.1	Laufzeiten der Fehlersimulation aus [LH91] und [GK08]	51
7.2	Laufzeiten von [LH91] gegenüber Beschleunigung mit GPGPU	52
7.3	Laufzeitanteile der GPGPU-Implementierung	53
7.4	Schaltungsmerkmale	54
A.1	Repräsentation der Netzliste C ₁₇ für die ereignisgesteuerte Simulation	60
B.1	Laufzeiten der exakten und der approximativen Simulation	67
B.2	Laufzeitanteile der GPGPU-Kernel an der approximative Fehlersimulation	68

Erklärung

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

(Marcel Schaal)