Embedded Systems & Applications

TECHNISCHE UNIVERSITÄT DARMSTADT

Optimiertes Floorplanning für CGRAs

Bachelorthesis von

Patrick van Halem

Juli 2018

Erstgutachter:
Prof. Dr.-Ing. Andreas Koch

Zweitgutachter:
Dr.-Ing. Andreas Engel

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Optimiertes Floorplanning für CGRAs

Bachelorthesis von Patrick van Halem
Eingereicht am 31.07.2018
Erstgutachter: Prof. Dr.-Ing. Andreas Koch
Zweitgutachter: Dr.-Ing. Andreas Engel

Eigenständigkeitserklärung

Hiermit versichere ich, Patrick van Halem, die vorliegende Bachelorthesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Bachelorthesis hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Bachelorthesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, 31. Juli 2018

____

(Patrick van Halem)

Kurzfassung

Die Arbeit befasst sich mit einer Teiloptimierung des Floorplans für die durch den CGRA Generator, aus der Arbeit [5], erstellten Designs. Ziel der Optimierung ist eine höhere Taktrate zu erreichen.
Zu Beginn wird auf die Modellierung das Layout des FPGAs, aus der Zynq 7000 Serie, in einem Koordinatensystem in Java eingegangen. Auf den Koordinaten werden die Informationen über die verbauten Ressourcentypen vermerkt. Damit bestimmte Module bei der Implementierung nur auf festgelegten Zellen platziert werden dürfen, wird ein PBlock erstellt. Dem PBlock können ein oder mehrere Module zugeordnet werden. Anhand des Koordinatensystems kann die Ressourcenanzahl für einen bestimmten Bereich ausgelesen und dem PBlock übergeben werden. Der Bereich richtet sich nach den benötigten Ressourcen, für die im PBlock enthaltenen Module. Anhand der Ablaufsteuerung der CGRA Architektur werden mögliche kritische Pfade analysiert. Ausgewählte Module, die auf dem kritischen Pfad liegen, werden anschließend mit Hilfe von ein oder zwei PBlöcken auf dem FPGA platziert. Der Suchraum wird mit dem Downhill Simplex Verfahren durchsucht. Die Auswahl der Module wird abhängig von dem generierten CGRA Model gemacht. Die Auswirkungen auf die Optimierung, durch die gewählten Module und Direktiven, werden in den Versuchen untersucht.
Für die Optimierung wurden insgesamt zehn Versuche auf vier verschiedenen CGRAs ausgeführt. In allen Versuchen wurde eine höhere Taktrate, gegenüber der Implementierung ohne Optimierung, erreicht. Die Verbesserung der Laufzeit, des kritischen Pfades, lag zwischen 3,41% und 9,48%. Jede Suche nach einer Optimierung hat zwischen 18 und 27 Stunden gedauert.

Inhaltsverzeichnis

1. Motivation und Vorgehensweise **1**

2. Technischer Hintergrund **3**

2.1. Coarse-Grained Reconfigurable Array 3
2.2. Synthese eines FPGA mithilfe von Vivado 5
2.3. Schnittstelle zwischen Vivado und Java 6

3. Stand der Forschung **9**

4. Vorgehen der Design Space Exploration **11**

4.1. Implementierung 11
4.1.1. Einbinden des FPGA Layouts in Java 11
4.1.2. Erstellen von PBlöcken 13
4.1.3. Analyse des kritischen Pfads 15
4.2. Algorithmen 17
4.3. Implementierung der Design Space Exploration 20

5. Ergebnisse **25**

5.1. Systeminformationen 25
5.2. Grundaufbau des CGRAs 25
5.3. Versuche 27
5.3.1. Heterogener CGRA mit einem Status Signal 28
5.3.2. Heterogener Coarse-Grained Reconfigurable Array (CGRA) mit drei Status Signalen 30
5.3.3. Heterogener CGRA mit allen Status Signalen 31
5.3.4. Homogener CGRA 34
5.3.5. Graphische Abbildungen ausgewählter Suchläufe 35

6. Diskussion **37**

7. Fazit **39**

A. Appendix **I**

Abbildungsverzeichnis **V**

Tabellenverzeichnis **VII**

Abkürzungsverzeichnis **IX**

Literatur **X**

1. Motivation und Vorgehensweise

Ein immerwährendes Ziel bei der Entwicklung eines Integrated Circuits (ICs) ist eine kompakte platzsparende Bauweise, ein geringer Energieverbrauch, sowie eine möglichst hohe Rechengeschwindigkeit. Der in dieser Arbeit verwendete IC ist ein Field-Programmable Gate Array (FPGA). Ein FPGA verbaut eine wiederprogrammierbare Logik. Durch die Bauart dieser ICs steht die Größe des ICs und die darauf verbauten Zellen schon im vorherein fest.
Die Optimierung eines Designs erfolgt in mehreren Schritten. Einer dieser Schritte ist das Floorplanning und Placement des Designs, auf den sich die vorliegende Arbeit konzentriert. Des Weiteren wird sich hauptsächlich mit einer Beschleunigung der Rechengeschwindigkeit befasst. Eine Möglichkeit das Optimierungsziel zu erreichen, ist das maximieren der möglichen Taktrate.
Für die Optimierung des Floorplannings werden verschiedene Anordnungen der in Hardware Desciption Language (HDL) beschriebenen Module verglichen, welche den festgelegten Anforderungen entsprechen.
Am Institut für Rechnersysteme an der Technischen Universität Darmstadt wird an einem Generator und der Architektur für heterogene CGRA Architekturen geforscht [5]. Auf die, durch den Generator erstellten, Architekturen wird die Teiloptimierung angewandt. Der Grundaufbau der verschiedenen CGRAs ist dabei gleich und wird über mehrere Verilog Dateien beschrieben. Dieser wird näher in Kapitel 2.1 beschrieben. Anhand des Grundaufbaus werden Module festgelegt, auf die Floorplanning angewandt wird. Für die Module werden anschließend Floorplanning Direktiven gesucht, welche einen effektiven Einfluss auf die Module haben, die die Geschwindigkeit des Designs erhöhen.
Für die standardmäßige Optimierung sollen die Floorplanning Option vom Synthesewerkzeug Vivado verwendet werden. Vivado wird dabei eingesetzt, um den in Verilog Code beschriebene CGRA auf einen FPGA zu implementieren. Damit die unterschiedlichen Konfigurationen der Floorplanning Direktiven getestet werden können, wird eine Schnittstelle zwischen Vivado und Java benötigt, um automatische Synthesen auszuführen. Zu der Auswahl der Direktiven soll das Wissen über das CGRA Model mit einfließen. Die Ergebnisse der verschiedenen Konfigurationen sollen anschließend verglichen werden, damit eine geeignete Konfiguration, die die höchste Frequenz erreicht, ausgewählt werden kann. Die Suche nach einer geeigne-

ten Konfiguration kann durch einen Suchraum, von ein paar Tausend bis Millionen Möglichkeiten, sehr lange dauern, wenn jede Konfiguration getestet wird. In Anbetracht das jede Synthese mehrere Minuten braucht. Daher wird nach einem Algorithmus gesucht, durch den die Anzahl an getesteten Konfigurationen geringgehalten wird.

Die Arbeit ist in fünf Kapitel aufgeteilt. Kapitel 2 befasst sich mit den technischen Grundlagen und Eigenschaften des CGRAs und der Synthese in Vivado. Als Vorbereitung auf diese Arbeit wurde eine Schnittstelle zwischen Vivado und Java geschrieben. Diese wird in dieser Arbeit erweitert. Die Funktionsweise wird in dem Unterkapitel 2.3 näher beschrieben. Das darauffolgende Kapitel 3 gibt einen kurzen Überblick über den Stand der Forschung und die Veröffentlichungen von Forschungsergebnissen, die sich mit Optimierungsansätzen für das Floorplanning beschäftigen. Außerdem werden Funktionen von Vivado, die sich auf eine Verbesserung der Frequenz auf einem FPGA beziehen, vorgestellt. Abschließend werden diese Ansätze hinsichtlich ihrer Eignung für die vorliegende Forschungsarbeit bewertet.
In Kapitel 4 wird die Auswahl und Implementierung der Floorplaning Direktiven behandelt. Dieses Kapitel ist in vier Unterkapitel gegliedert. In 4.1 geht es zunächst um den Aufbau eines FPGA. Danach wird das Erstellen und Platzieren von PBlöcken auf dem vorher in einem Koordinatensystem modellierten Aufbau des FPGA behandelt. Der nachfolgende Abschnitt 4.1.3 behandelt den kritischen Pfad des CGRA. In Abschnitt 4.2 werden zwei Algorithmen vorgestellt und anschließend die Implementierung des Downhill Simplex Verfahrens näher erläutert. Abschließend wird im letzten Abschnitt 4.3 der grundlegende Ablauf der Design Space Exploration zusammengefasst.
Kapitel 5 beschreibt die Auswahl und Evaluation von verschiedenen CGRA Designs. In dem Kapitel werden verschiedene CGRA Strukturen vorgestellt, welche für die Design Space Exploration verwendet wurden. Im Abschnitt 5.3 werden daraufhin die erzielten Ergebnisse in Tabellen präsentiert.
Im letzten Kapitel 6 werden die im vorherigen Kapitel erstellte Evaluation mit einer Optimierungsoption von Vivado verglichen und besprochen.

2. Technischer Hintergrund

Im folgenden Kapitel werden auf die Grundlagen und Merkmale eines CGRAs dargestellt. Danach werden die Eigenschaften eines FPGA und die Verwendung von Vivado beschrieben.

2.1. Coarse-Grained Reconfigurable Array

Dieser Überblick umfasst den Aufbau des hier verwendeten CGRAs. Der Ablauf ist für diese Thesis interessant, da über diesen der kritische Pfad definiert wird.
Ein Coarse-Grained Reconfigurable Array (CGRA) ist eine Schaltkreis-Architektur, die entwickelt wurde, um unterschiedliche Prozesse zu beschleunigen. Üblicherweise werden CGRAs in eingebetteten Systemen eingesetzt [4], in denen einem Haupt-Prozessor ein Rechenbeschleuniger zur Verfügung gestellt wird. Mithilfe eines CGRAs werden Berechnungen auf Word Level ausgeführt [4]. Word Level bedeutet, dass die Bitbreite für die Berechnung vorher festgelegt wurde. Die Beschleunigung wird erzielt, indem mehrere Berechnungen gleichzeigit ausgeführt werden. Diese Parallelität ist dabei beliebig skalierbar, solange die Anwendung dies zulässt, und wird durch die Größe des CGRAs und der Anzahl an verbauten Processing Elements (PEs) beschränkt. Ein PE verfügt über vordefinierte Rechenoperationen. Damit ein Austausch an Daten erfolgen kann, sind die PEs untereinander verschaltet. Dadurch können Ergebnisse oder Operatoren schneller an ein weiteres PE übergeben werden. Normalerweise unterscheiden sich die PEs nicht in den verbauten Operatoren und der CGRA besitzt einen homogenen Aufbau. Durch den homogenen Aufbau kann jede PE gleiche Konfiguriert werden, wordurch die Entscheidung, welches PE die nächste Berechnung ausführt, leichter ist [4].
Das CGRA [5], das vom Institut Rechnersysteme entwickelt wird, unterscheidet sich zu herkömmlichen CGRAs im Wesentlichen durch einen heterogenen Aufbau. Dies bedeutet, dass jede PE unabhängig von den anderen mit unterschiedlichen Rechenoperatoren designet werden kann. Des Weiteren können die Verbindungen zwischen verschiedenen PEs beliebig gewählt oder auch weggelassen werden. Da nicht jede PE die gleichen Rechenoperationen besitzt, kann Platz für weitere PEs oder andere Module eingespart und der FPGA energiesparender werden. Module oder auch

HDL-Module beschreiben die verschiedene Bauteile, wie ein PE oder ein Operator, einer kompletten Architektur. Das Einsparen von Modulen führt dazu, dass nicht alle PEs miteinander verbunden sind und alle die gleichen Operatoren ausführen können. Deshalb können nicht alle PEs gleich angesteuert werden. Für einen kontrollierten Ablauf aller Funktionen des CGRAs, wurde eine Steuereinheit entwickelt.
Zum einfachen Erstellen verschiedener CGRAs, wurde ein Generator in Java geschrieben. Mit diesem Generator ist es möglich über die Vorgabe von festgelegten Parametern eine funktionstüchtige Architektur in einer HDL zu generieren [5]. Zu den Parametern gehören die Anzahl der PEs mit ihren zugehörigen Operatoren, die Verbindung zwischen dem Register File eines PE mit dem Rechenmodul eines anderen PE und die Größe der Speicher.
Wie in Abbildung 2.1 dargestellt besitzt jedes PE ein Context Memory, welche die Konfiguration des PE für jeden Takt steuert. In der Abbildung werden die verschiedenen PEs in einem Array zusammengefasst. Das Context Memory wird durch einen globalen Context Counter indiziert, welcher wiederum von der Context Control Unit (CCU) erzeugt wird. Die CCU wird von der Control-Box (cBox) kontrolliert. Die cBox verarbeitet die Status Signale der PEs. Ob ein PE ein Status Signal besitzt kann davor individuell festgelegt werden. Mit Hilfe der Status Signale kann die cBox definierte Branches für die CCU festlegen. Durch die festgelegten Branches wird so dem PE ein Control-Flow ermöglicht [5]. Außerdem kann über das Predication Signal den PEs übermittelt werden, wann ein Ergebnis gespeichert werden soll.

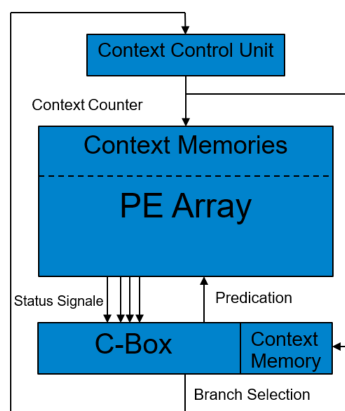


Abbildung 2.1.: Kopie aus [5]: abstrakter Überblick über die CGRA Struktur

In dem PE werden die vom Context Memory gespeicherten Befehle ausgeführt. Durch das Context Memory wird die Arithmetische Recheneinheit eines PE in jedem Takt aufs neue konfiguriert, sodass das PE in jedem Takt eine Rechenoperation ausführt. Abbildung 2.2 zeigt die Grundaufbau eines PE auf. Die Arithmetic Logic

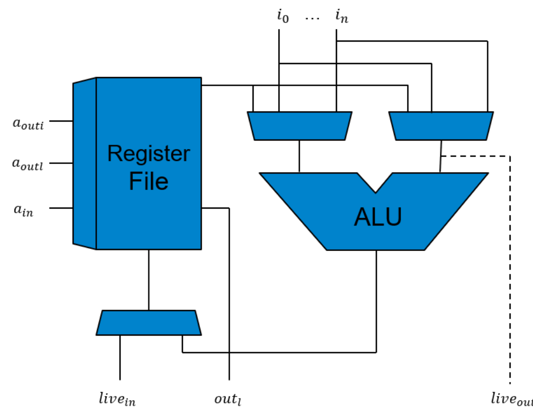


Abbildung 2.2.: Kopie aus [5]: Überblick über ein Processing Element

Unit (ALU) ist über ein Multiplexer entweder mit dem Register File oder über $i_0 - i_n$ mit dem Register File eines weiteren PE verbunden. In der ALU sind alle Rechenoperationen eines PE enthalten. Die Register Files sind die Zwischenspeicher für die Operanden und Ergebnisse. out_l ist der Ausgang des Registers, wodurch Operanden an andere PEs oder Module übertragen werden. Ein abgespeichertes Ergebnis oder Status Signal wird über $live_{out}$ geschickt.

2.2. Synthese eines FPGA mithilfe von Vivado

Die Architektur des CGRA wird auf einem FPGA implementiert. Die CGRA Architektur kann auch auf einen Full-Custom Application-Specific Circuit (ASIC) übertragen werden. Der Vorteil eines FPGAs ist allerdings die Wiederprogrammierbarkeit von Logik Bausteinen. Durch die Wiederprogrammierbarkeit kann der FPGA auf die benötigten Rechenoperationen angepasst und damit als flexibler Hardware Beschleuniger verwendet werden [5]. Ein Full-Custom ASIC ist nicht wieder programmierbar, ist aber bei Herstellung einer großen Stückzahl billiger und kann mit einer höheren Frequenz betrieben werden, wodurch schnellere Berechnungen möglich sind.
Um den in HDL beschriebenen Code auf einem FPGA auszuführen, muss der Code zuerst für den jeweils verwendeten FPGA synthetisiert und implementiert werden. Für die Synthese und Implementierung wird das von Xilinx entwickelte Programm Vivado Design Suit verwendet. Durch die von Vivado ausgeführten Synthese wird der vom CGRA-Generator erzeugte HDL-Code auf Gate-Level Ebene transformiert[8]. Vivado berechnet aus dem Code eine Anordnung, wie die Logik Bausteine eines

FPGA verwendet werden und wie diese untereinander Verschaltet sind. Vivado will für die Implementierung eine Frequenzvorgabe für den FPGA haben. Diese Frequenzvorgabe versucht Vivado mit der Synthese zu erreichen und ordnet dementsprechend die Logik Bausteine an. Nach einer abgeschlossenen Implementierung wird der kritische Pfad ermittelt und mit der vorgegebenen Zeitvorgabe verglichen. Der kritische Pfad gibt an, wie viel Zeit das längste Signal braucht, um zwischen zwei Flip-Flops/Registern gesendet zu werden. Über den Kehrwert der Frequenz kann die gegebene Laufzeit berechnet werden. Die Differenz zwischen der vorgegebenen Laufzeit minus der Zeit, welche für den kritischen Pfad benötigt wird, stellt den Slack dar. Wenn dieser Slack kleiner als null zurückgibt, ist die vorgegebene Frequenz nicht erreichbar. Dadurch kann der FPGA nicht mit der vorgegebenen Frequenz betrieben werden. Vor der Synthese können in Vivado weitere Einschränkungen (Constraints) festgelegt werden. Ein Constraint ist eine vom Nutzer festgelegte Bedingung die die Synthese einhalten muss. Die vorher beschriebene Frequenzvorgabe ist ein solches Constraint. Die Bezeichnung dieser Constraints heißen in Vivado Xilinx Design Constraint (XDC). Diese basieren auf einem Standard Synopsys Design Constraint und werden wie die Tool Command Language (TCL) Befehle spezifiziert [10]. TCL bezeichnet die Umgebungssprache, welche von Vivado ausgelesen werden kann. Vivado erweitert die Sprache, um eine Reihe eigener Befehle [7]. Dadurch können zum Beispiel verschiedene Verilog Dateien einem Projekt hinzugefügt werden oder eine Synthese gestartet werden. Auch können über die Konsole Constraints erstellt oder ausgelesen werden. Durch Vivado wird so eine einfache Bedienung für die Synthese eines CGRA zur Verfügung gestellt. Über den Befehl

```
report_timing_summary
```

können nach abgeschlossener Synthese Informationen zum erreichten Slack ausgelesen werden.
In der Thesis wird auf Placement Constraints eingegangen. Durch Placement Constraints werden Module durch räumliche Einschränkung auf festgelegte Teile des FPGA beschränkt.

2.3. Schnittstelle zwischen Vivado und Java

Diese Arbeit baut auf einem bestehenden Java-Framework auf, mit welcher der Verilog Code der CGRA Architektur erzeugt werden kann. Der generierte CGRA wird in 2.1 näher behandelt. Für die Arbeit werden über einen längeren Zeitraum unterschiedliche Synthesen in Vivado ausgeführt und die berechneten Ergebnisse anschließend ausgewertet. Daher wurde als Vorbereitung auf diese Arbeit das Java-Framework mit einer Schnittstelle zwischen Java und Vivado erweitert. Die Schnittstelle, die Vivado verwendet, basiert auf der TCL. Die Sprache wird von einer Viel-

zahl weiterer Programme verwendet. Die Sprache wurde in Vivado um weitere Befehle erweitert, welche in den Handbüchern von Xilinx dokumentiert sind. Zur Kommunikation wird über die vom Betriebssystem verwendete Konsole Vivado gestartet und eine TCL Datei zum Start übergeben. In dieser Datei stehen alle Befehle, die für eine Synthese benötigt werden.

Dazu gehören ein neues Projekt zu erstellen und alle benötigten Verilog Dateien dem Projekt hinzu zufügen, sowie alle Startparameter festzulegen. Vivado wird somit beim Start alle auszuführenden Befehle übergeben. Für eine erfolgreiche Synthese wird die Vorgabe eines Taktes mit der der FPGA betrieben werden soll, benötigt. Das Constraint wird vor dem Start von Vivado in einer eigengen Datei gespeichert und über die TCL im Projekt eingebunden. Die laufende Synthese wird über eine Methode jede Konsolen Zeile gelesen (lineReader) und kann dadurch abgebrochen werden, wenn Vivado einen Fehler (ERROR) zurückgibt. Damit anschließend Ergebnisse ausgelesen werden können, müssen diese erstellt werden. Die Befehle sind in der TCL Datei enthalten und geben das Zeitverhalten und die verwendeten Ressourcen in verschiedenen Text Datei zurück. Die Text Dateien haben immer den gleichen Aufbau, sodass die benötigten Elemente über Java ausgelesen werden können. Die ausgelesenen Werte können in Java verwendet. Zusätzlich werden die Werte in einer Text Datei gespeichert. Das Methode zum Ausführen von Vivado wurde in einer Unterklasse von Runnable implementiert und somit ist es möglich, mehrere Threads gleichzeitig zu starten. Die Anzahl an Threads kann über eine Methode festgelegt werden.

3. Stand der Forschung

Floorplanning wird angewandt, um ein Design optimal auf einem IC zu platzieren. Dabei ist das Ziel, so platzsparend wie möglich die Module auf dem IC anzuordnen und die Performance des ICs zu verbessern. In den Konzepten, die in den Veröffentlichungen [1][2] dargestellt sind, werden verschiedene Herangehensweisen erforscht, um den Floorplan für ein Design auf einem FPGA zu entwerfen. Für eine optimale Platzierung werden die benötigten Ressourcen, die ein Modul braucht, und die Verbindungen zwischen den Modulen in Betracht gezogen [6], um über diese Informationen eine optimale Anordnung der Module zu finden. Eine effektive Aufteilung wird in [2] vorgestellt. Hier wird auf den Nachteil von der Verwendung rechteckiger Anordnung der Module auf FPGAs eingegangen. Durch die rechteckige Annordnung kann es vorkommen, dass einige Ressourcen eingeschlossen werden, welche in dem Modul nicht gebraucht werden. Floorplanning wurde zuerst für ein ASICs entwickelt, allerdings unterscheidet sich aufgrund der neueren FPGA Architekturen das notwendige Floorplanning von den beim ASICs eingesetzten Verfahren. Der Unterschied liegt dabei bei der Definition des Bereiches für ein Modul. Ein Modul wird auf einem ASICs meist nur durch die Höhe und Breite bestimmt und kann auf dem ganzen IC mit den gleichen Eigenschaften verbaut werden. Bei einem FPGA ist das Problem, dass verschiedene Ressourcentypen in einer schon festen Annordnung verbaut sind und die Platzierung der Module sich an dieser Annordnung orientieren müssen. Die Ausmaße lassen sich aus der Dichte der verbauten ASICs Ressourcen ermitteln [2]. Dieser homogene Aufbau von ASICs macht Floorplanning für diese Art von IC einfacher als beim FPGA. In [1] wird ein Verfahren für das Floorplanning von FPGAs vorgestellt. In diesem Verfahren wird Floorplanning auf jedes Modul angewandt. Die Beziehungen zwischen zwei Modulen wird über eine vertikale oder horizontale Trennung festgelegt. In einem Slicing Tree werden die Module aufgehängt und die Verbindung zwischen zwei Zweigen gibt die Trennungsrichtung an. Durch das Tauschen zweier Module im Baum, wird eine neue Anordnung geschaffen. Als Bewertungsfunktion wird die maximal verwendete Ressourcen, als auch der Abstand zwischen Modulen, die miteinander verbunden sind, bewertet. Module, die viele Verbindungen zueinander haben, werden in dem Bewertungskriteriums höher gewertet.

In [11] werden Direktiven für ASICs vorgestellt, welche sich auf die Beziehungen zwischen Modulen beziehen. Obwohl Floorplanning für ASICs nicht direkt mit dem Floorplanning für FPGAs vergleichbar ist, können manche dieser Techniken auch auf einen FPGA übertragen werden. In [11] wird für eine gewisse Anzahl von Modulen eine Beziehung festgelegt. Zum einen gibt es bestimmte Module, die fest nebeneinander platziert werden können, da diese viele Verbindungen zueinander haben. Anderseits können bestimmte Module in nur einem vorgegebenen Bereich getestet werden, wodurch der Suchalgorithmus eingeschränkt wird. Weiter kann man Module welche mit den I/O Ports des FPGA verbunden sind, in dessen Nähe platzieren. Dadurch wird vor dem Suchen eines geeigneten Floorplan mit dem Vorwissen über die Architektur schon gewisse Einschränkungen festgelegt, die beim floorplanning beachtet werden müssen. In der Thesis wird über die Verbindung zweier Module eine Zusammengehörigkeit bestimmt, wodurch die beiden Module in einem PBlock zusammengeführt werden. Eine Verbesserung der Performance kann über Befehle, welche Einfluss auf die Synthese und Implementierung nehmen, von Vivado erzielt werden [9]. Durch die Befehle kann vor, aber auch nach der Synthese, Einfluss auf das platzierte Design genommen werden. Hierdurch kann, wie beim eigentlichen Floorplanning, der Platz optimiert, sowie die Performance verbessert werden. Eine Platzoptimierung ist hier allerdings weniger von Bedeutung. Für eine Performance Optimierung werden nach abgeschlossener Synthese die schlechtesten kritischen Pfade betrachtet und ermittelt, welche Ressourcen auf diesem Pfad liegen und wie diese unplatziert werden können. Falls eine bessere Platzierung möglich ist, wird die Zelle verschoben. Mit einer anderen Optimierungsoption phys_opt_design können weiter entfernte Ressourcen, die auf einem kritischen Pfad liegen, repliziert werden. Die replizierten Ressourcen werden daraufhin näher an der Last gesetzt ohne die anderen Zellen zu beeinflussen. Durch die Optimierung werden dem Design zusätzliche Zellen hinzugefügt, durch welche die Ausnutzung des FPGA steigt.
Eine naheliegende Optimierung besteht hauptsächlich in den von Vivado für die Implementierung zur Verfügung gestellten Befehlen . Aus der Arbeit [1] können Ansätze für die Teiloptimierung entnommen werden.

4. Vorgehen der Design Space Exploration

Die Design Space Exploration ist eine Analyse der verschiedenen Designs in der eine Architektur angeordnet werden kann. Dafür werden bestimmte HDL-Module, welche über den kritischen Pfad und dem in Kapitel 2 vorgestellten Aufbau der CGRAs, an unterschiedlichen Zellen des FPGAs platziert und die Auswirkung auf die Performance getestet. Bei der Analyse wird speziell nach Möglichkeiten zur Erhöhung der Frequenz geforscht.
Das folgende Kapitel 4.1 beschreibt zuerst den Suchraum und die Möglichkeit Placement Constraints mithilfe von PBlöcken zu erstellen, wodurch die Implementierung des Designs beeinflusst wird. Darauf folgt in 4.1.3 eine Analyse des kritischen Pfad und welche Module in diesem Enthalten sind. Um im Suchraum gezielt nach Ergebnissen zu suchen, wird in 4.2 ein Algorithmus vorgestellt, mit dem durch möglichst wenige Iterationen ein geeignetes Ergebnis gefunden werden soll.

4.1. Implementierung

4.1.1. Einbinden des FPGA Layouts in Java

Wie schon in Kapitel 2.2 erwähnt, besteht ein FPGA aus mehreren verschiedenen Ressourcen. Die in dieser Thesis verwendete Zynq 7000 Serie baut sich aus drei verschiedenen Ressourcen auf. Zu den Ressourcen gehören Slices, Digital Signal Processing (DSP) und Random-Access Memory (RAM). Auf dem FPGA wird ein Dual-Port 18 kBRAM verbaut, dadurch kann der RAM auch in zwei vollwertige 18 kBRAM Blöcke geteilt werden. Da in Vivado diese beiden RAM Größen getrennt betrachtet werden, erscheinen sie als zwei unterschiedliche Ressourcen. Der Speicher wird allerdings zwischen den Blöcken geteilt, wodurch entweder zwei 18 kBRAM oder ein 36 kBRAM verschaltet wird. Diese Eigenschaft ist beim späteren Generieren von PBlöcken im Abschnitt 4.1.2 wichtig.
Die Ressourcen sind in einem gewissen Muster auf einem FPGA verbaut. In Abbildung 4.1 wird ein FPGA Layout von einem Xilinx Zynq XC7Z030 gezeigt [12]. Auf

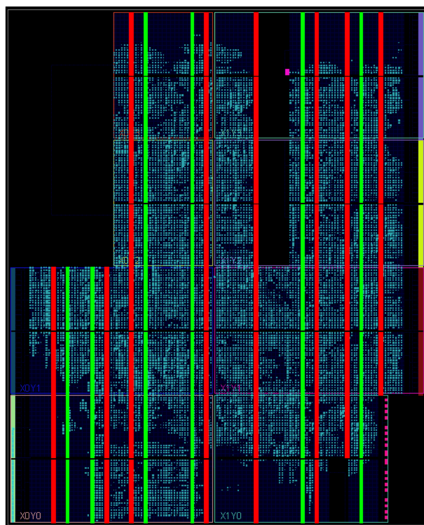


Abbildung 4.1.: eines in Vivado synthetisierten Designs auf einem Xilinx XC7Z030 FPGA

diesem ist ein CGRA implementiert, welcher 92% der verfügbaren Slices verwendet. Auf der Abbildung sind die verwendeten Slices (blau), DSP (grün) und RAM (rot) zu erkennen. Aus der Abbildung kann entnommen werden, dass DSP und RAM in einer Reihe in horizontaler Richtung angeordnet sind. Auf diesem FPGA haben die DSP und 18 kBRAM Blöcke eine Höhe von zweieinhalb Slices. Ein 36 kBRAM umschließt zwei 18 kBRAM und ist so hoch wie 5 Slices. Der FPGA hat eine Rechteckige Form. Allerdings wird in Teilen des FPGA nicht programmierbare Logik verbaut. Auf diesen freien Abschnitten befinden sich bei dem verwendeten Zynq FPGA ein ARM-Prozessor, I/O Connect und weitere Systeme, die für diese Thesis nicht wichtig sind. In Abbildung 4.2 ist ein Ausschnitt der Ressourcen eines FPGA in Einfacher Form dargestellt. Das Farbschema für die Ressourcentypen aus Abbildung 4.1 wurde übernommen. In Vivado wird jeder Ressourcentyp einem eigenen Koordinatensystem zugeteilt. Aus diesem Grund kann über das Koordinatensystem nur die Position zwischen zwei gleichartigen Ressourcen ermittelt werden, jedoch nicht die räumliche Zuordnung zwischen den unterschiedlichen Ressource. Durch einen zur Koordinate angemerkten Zusatz, die Bezeichnung der Ressource, wird das entsprechende Koordinatensystem bestimmt.

Aufgrund der räumlichen Darstellungsschwierigkeit zwischen den verschiedenen Ressourcen, werden diese in einem neuen Koordinatensystem in Java modelliert. Die Modellierung muss manuell erfolgen, da ein einzelnes Koordinatensystem nicht aus Vivado erstellt werden kann oder diese vom Hersteller zur Verfügung gestellt werden. Für die Modellierung wird ein zweidimensionales Array erstellt. Damit die unterschiedlichen Größen von Slices und RAM, welcher 2,5 Slices hoch ist, richtig dargestellt werden können, wird des Koordinatensystem um einen Faktor von

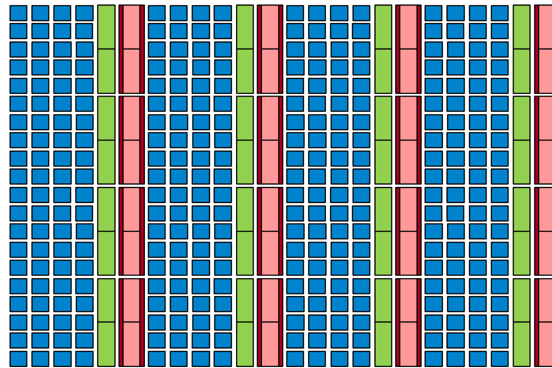


Abbildung 4.2.: Kopie aus [1]: Vereinfachung eines FPGA Layouts

zwei vergrößert. Jede Koordinate im Array erhält die Informationen über den Ressourcentyp und die in Vivado verwendete Koordinate. Um eine Ressource richtig im Koordinatensystem darzustellen, wird ein Block definiert. Der Block enthält die Anfangskoordinate als auch seine Höhe und Breite. Diese Information wird in jeder vom Block eingeschlossenen Koordinate gespeichert. Obwohl ein 36 kBRAM Block zwei 18 kBRAM Blöcke umschließt, wird dieser nicht extra definiert, sondern beim Auslesen kann der 36 kBRAM Block anhand der 18 kBRAMs und seiner Höhe ermittelt werden. Jedem Block kann der Name eines PBlocks, der im nächsten Kapitel erklärt wird, eingetragen oder gelöscht werden. Stellen, auf denen keine programmierbare Logik platziert ist, werden mit einem leeren Marker (EMPTY) gekennzeichnet.
Anhand des in Java modellierten Koordinatensystem, kann nun eine bessere Zuordnung zwischen zwei unterschiedlichen Ressourcentypen erfolgen. Dadurch kann dem Koordinatensystem entnommen werden, wie viele unterschiedliche Ressourcentypen in einem bestimmten Bereich des FPGA verbaut sind, oder wo die Position der PBlöcke aufgespannt sind.

4.1.2. Erstellen von PBlöcken

In Vivado werden Placement Constraints mit PBlöcken dargestellt. Über die PBlöcke wird definiert, in welchem Bereich des FPGA das Synthesewerkzeug bei der Implementierung des Designs für ein HDL-Modul Ressourcen verwenden darf. Außerhalb des definierten Bereich dürfen dem PBlock keine Ressourcen zugeteilt werden. Dadurch ist der PBlock auf die festgelegten Ressourcen fixiert.
Der folgende Pseudo Code stellt eine XDC Datei für einen PBlock dar:

```
create_pblock Name_for_PBlock
add_cells_to_pblock [get_pblocks Name_for_PBlock]
    [get_cells -quiet [list Constrained_Module]]
resize_pblock [get_pblocks Name_for_PBlock]
```

```
    -add {SLICE_X22Y80:SLICE_X49Y110}
resize_pblock [get_pblocks Name_for_PBlock]
    -add {DSP18_X1Y32:DSP18_X2Y43}
resize_pblock [get_pblocks Name_for_PBlock]
    -add {RAM18_X1Y32:RAM18_X2Y43}
resize_pblock [get_pblocks Name_for_PBlock]
    -add {RAM36_X1Y16:RAM36_X2Y21}
```

In der ersten Zeile wird ein PBlock erstellt und mit einem Namen definiert. Darauf werden einem PBlock über get_cells ein oder auch mehrere Zellen eines HDL-Module zugeteilt. Aufgrund der verschiedenen Koordinatensysteme muss der Raum für jede einzelne Ressource definiert werden. Falls eine Ressource einem PBlock nicht zugeordnet wurde, wird diese bei der Synthese nicht eingeschränkt und kann unabhängig von den definierten Ressourcen auf dem ganzen IC platziert werden. Wenn einem PBlock für die Synthese zu wenige Ressourcen zugeordnet wurden, kommt es zu einem Fehler in der Synthese. Dieser Fehler wird aber erst gegen Ende der Synthese ausgegeben, was aufgrund der langen Synthese mehrere Minuten dauern kann. Da der Bereich für die Ressourcen eines PBlocks über zwei Koordinaten, die untere linke und die obere rechte, bestimmt wird, ist der Bereich immer rechteckig. Da eine vordefinierte Größe des PBlocks aufgrund der unterschiedlichen Ressourcen des FPGA nicht beibehalten werden kann, muss für jede Ausgangskoordinate ein neuer Bereich bestimmt werden. Ein Vektor über die verschiedenen Ressourcentypen wird mit $\theta = (CLB, RAM18, RAM36, DSP)$ festgelegt. Dieser Vektor gibt die benötigte Anzahl der verschiedenen Ressourcentypen an, die ein Modul benötigt. Ausgehend von der Ausgangskoordinate wird ein Rechteck gesucht, welche alle Elemente des Moduls einschließt, um die Elemente des Moduls darauf abzubilden. Zur Veranschaulichung wird in Abbildung 4.3 ein PBlock gesucht, welches die folgenden Ressourcen benötigt $\theta = (16, 1, 1, 1)$. Als Ausgangskoordinate werden $(3, 4)$ oder $(16, 5)$ gewählt. Wie man nun sieht, gibt es mehrere Möglichkeiten ein Rechteck aufzuspannen, in dem mindestens alle Ressourcen enthalten sind. Da im Rahmen dieser Thesis nicht alle Module auf dem IC aufgeteilt werden müssen, wird ein PBlock mit einer möglichst kompakten Größe gesucht (gelbes Rechteck). Als kompakt kann ein PBlock angenommen werden, wenn das aufgespannte Rechteck einem Quadrat nahe kommt. Die maximale Distanz zwischen allen eingeschränkten Ressourcen ist bei einem Quadrat am kleinsten, was aber nicht immer das optimale Design liefern muss. Aufgrund der Anordnung von RAM und DSP in einer horizontalen Spalte, ist es nicht immer möglich ein Quadrat zu bilden. Daher wird festgelegt, dass das Rechteck nicht breiter wird, sobald die Anzahl der benötigten Slices erreicht ist. Zusätzlich wird die Breite und Höhe durch die Ausmaße des FPGA Layouts oder durch einen benachbarten PBlock eingeschränkt. PBlöcke dürfen sich in Vivado überlappen. In der Implementierung ist dies nicht der Fall. Überlappte PBlöcke müssen sich die

Ressourcen teilen, wodurch die PBlöcke wiederum größer definiert werden müssen. Die PBlöcke werden in der Implementierung nacheinander auf dem Layout platziert.

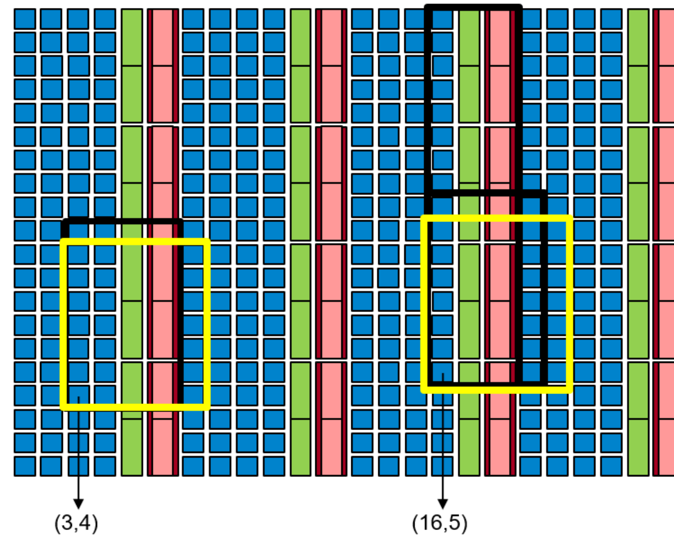


Abbildung 4.3.: Kopie aus [1]: Das Modul braucht 16 CLBs, ein 36 kBRAM und ein DSP. Es werden PBlöcke von den Startpunkten (3,4) und (16,5) definiert, welche alle benötigten Ressourcen beinhalten. Die gelben Rechtecke geben die implementierte Lösung an.

4.1.3. Analyse des kritischen Pfads

In Abschnitt 2.1 wurde das Grundkonstrukt des CGRAs vorgestellt. Wie bereits in der Einleitung ausgeführt, ist es das Ziel dieser Arbeit, ein Design zu finden, das zu einer Verbesserung der Frequenz führt. Dazu sollen bestimmte Module zur Verkürzung des kritischen Pfad gezielt platziert werden. Faktoren für die Zeit sind die Distanz zwischen zwei Registern und der Verzögerung durch kombinatorische Logik, wie Multiplexer und Look-Up Table. Durch Floorplanning kann die Anzahl an verwendeten Ressourcen nicht beeinflusst werden. In Vivado ändert sich die Anzahl der verwendeten Ressourcen pro Synthese für die gleiche Architektur in einem nicht merkbaren Bereich und wirkt sich somit nicht auf das Floorplanning aus. Daher muss für eine hohe Frequenz die Distanz des kritischen Pfades verkürzt werden. Solange sich der kritische Pfad nicht innerhalb eines Modules befindet, sondern sich über mehrere Module erstreckt, kann dieser verbessert werden. Durch eine nähere Platzierung zweier Module die auf dem kritischen Pfad lagen, wird dieser verkürzt. Allerdings kann dadurch ein anderes Modul weiterweg platziert werden, was zu

einem neuen kritischen Pfad führt. Die richtige Anordnung der Module ist daher entscheidend. Die Bewertungsfunktion aus der Arbeit [1] werten eine enge Platzierung von Modulen, die viele Verbindungen zueinander haben, höher.

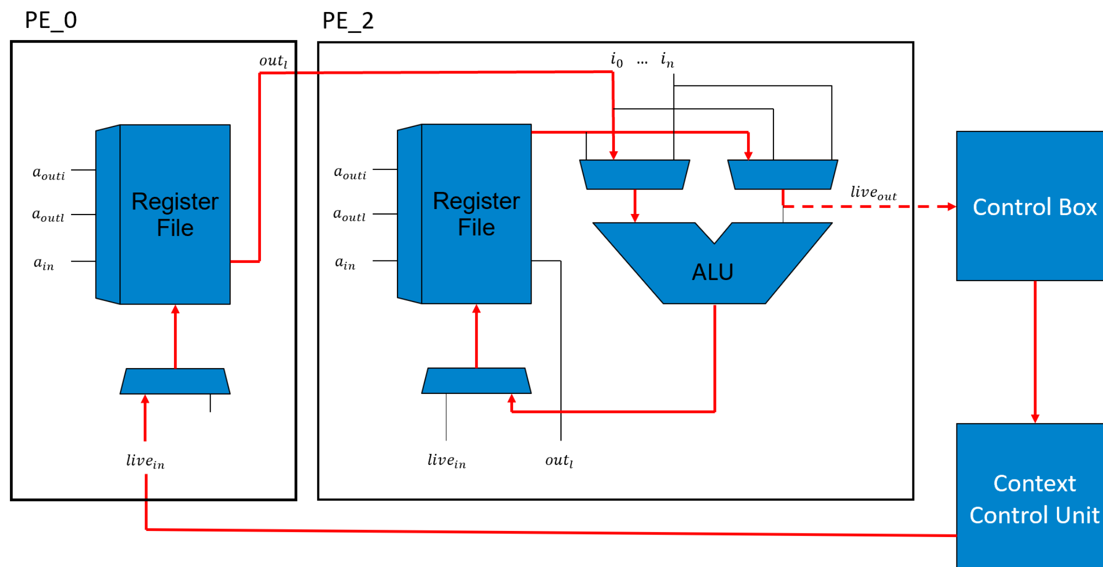


Abbildung 4.4.: Stellt einen vereinfachte aus Vivado dargestellten kritischen Pfad auf dem CGRA[5] dar

Die im Kapitel 2 gezeigte Ablaufsteuerung 2.1 kann Grundlage für mögliche kritische Pfade sein. Ein möglicher kritischer Pfad kann das von dem PE zur cBox verlaufende Status Signal sein, das verrechnet und dann zur CCU weitergeleitet wird, um anschließend wieder im Context Memory des PE zu enden. Ein weiterer möglicher kritischer Pfad kann auch der Austausch von Daten zischen zwei PEs sein. Um diese Annahme zu bestätigen wurden verschiedene CGRA Konfigurationen in Vivado ohne Einwirkungen von Constraint synthetisiert. Dabei kann entweder einer der beiden kritischen Pfade auftreten als auch beide zusammen. In Abbildung 4.4 ist eine vereinfachte Version einer durch Vivado erzeugten Schematik, des kritischen Pfades, abgebildet, welche beide Annahmen vereinen. Der kritische Pfad wurde aus einem synthetisierten CGRA, bei welchem alle PEs ein Status Signal hatten, entnommen. Der kritische Pfad ist hier rot markiert und startet im Register File vom PE_0. Das Signal schickt Daten von einem PE an eine anderes PE, die ALU berechnet aus den Daten ein Status Signal und leitet dieses zur cBox weiter. Die cBox berechnet aus diesem Signal eine neuen Branch und schickt die Daten weiter an die CCU. Die CCU verteilt darauf die Daten auf die Context Memorys der verschiedenen PEs, welche in der Abbildung nicht dargestellt werden.
Aus diesem kritischen Pfad erschließt sich die zentrale Position der cBox, sowie

der CCU. Ferner sollten diejenigen PEs, die ein Status Signal berechnen können, möglichst nah an der cBox liegen. Das Platzieren aller PEs neben der cBox ist meist aus Platzgründen schwierig und könnte zu eher nachteiligen Formen der verschiedenen Module führen. Daher wird für eine homogenere Verteilung der Module die Platzierung von Vivado verwendet. Allerdings wird versucht über ein Placement Constraint die cBox auf unterschiedliche Orte des FPGA einzuschränken. Damit soll erreicht werden, dass Vivado die Zellen der cBox nur in dem festgelegten Bereich platzieren darf. Mit der Vorgabe werden die Zellen der anderen Module ausgehend von dem gesetzten Module platziert. Für das Constraint wird einem PBlock das Modul der cBox übergeben.
Bei heterogenen CGRAs kann die Anzahl der verschiedenen Status Signale von PE zu PE variieren oder bestimmte PEs können gar kein Status Signal besitzen. Anhand dieser Eigenschaften wird ein PE ermittelt, welche die meisten Status Signale besitzt und wenn zwei PEs gleich viele Besitzen, das PE die mehr Rechenoperationen verbaut hat. Dadurch wird ein zweites Modul ausgesucht, welches einen hohen Einfluss auf den kritischen Pfad haben kann, wegen der Anzahl der Verbindungen zur cBox und der Größe des Moduls. Das PE wird entweder dem PBlock der cBox hinzugefügt, um die beiden Module möglichst eng beieinander zu platzieren oder ein eigener PBlock erstellt, welcher unabhängig vom PBlock der cBox bewegt wird. Im Kapitel 5 wird erforscht, welche Variante mehr Einfluss auf den FPGA hat. Bei einem homogenen CGRA sind alle PEs gleich, wodurch eine spezielle Platzierung eines PE nicht relevant ist.
Auch wird erforscht wie stark sich die Frequenz verbessert, wenn der PBlock der cBox mit der CCU erweitert wird oder ob dies keine Auswirkung hat.
Zu den gesetzten PBlöcken kann ein Spielraum festgelegt werden. Der Spielraum gibt dabei an, wie die prozentuale Verteilung von benötigten Slices zu denjenigen Slices ist, die von einem PBlock eingeschlossen sind. Dies soll Vivado ermöglichen ein wenig mehr Spielraum in der Synthese zu haben, damit das Modul nicht in der definierten Form eines Rechtecks platziert werden muss.

4.2. Algorithmen

Mit jedem Modul, welches explizit platziert werden soll, steigt die Anzahl der verschiedenen Möglichkeiten exponentiell an. Aufgrund eines immer größer werdenden Suchraums ist es nicht mehr möglich, alle Konfigurationen zu testen. Bei diesem spezifischen Problem muss die Zahl der Iterationen sogar noch weiter reduziert werden. Da für jede Konfiguration eine neue Synthese gestartet wird, dauert jeder einzelne Schritt mehrere Minuten. Eine Synthese nimmt meist mehr als 20 Minuten in Anspruch. Daher werden in diesem Kapitel eine heuristische Methode vorgestellt, welche für die Suche nach der optimalen Konfiguration geeignet ist.

Simulated Annealing [6, pp. 254-259] wird bei vielen Floorplanning Problemen angewandt, da dieses Suchverfahren vergleichsweise robust gegen lokale Minima ist. Jeder Schritt in diesem Suchverfahren wird zufällig bestimmt, wobei die maximale Schrittweite immer weiter reduziert wird. Daher wird der Suchraum am Anfang großflächig untersucht und wird dann immer kleiner. Das Verfahren sucht solange, bis die vorgegebene Anzahl an Iterationen erreicht ist. Umso höher die angegebene Anzahl ist, umso genauer wird auch das anschließende Ergebnis. Allerdings wird das Suchverfahren für sehr große Suchräume von mehreren Millionen Möglichkeiten angewandt.

Downhill Simplex Verfahren

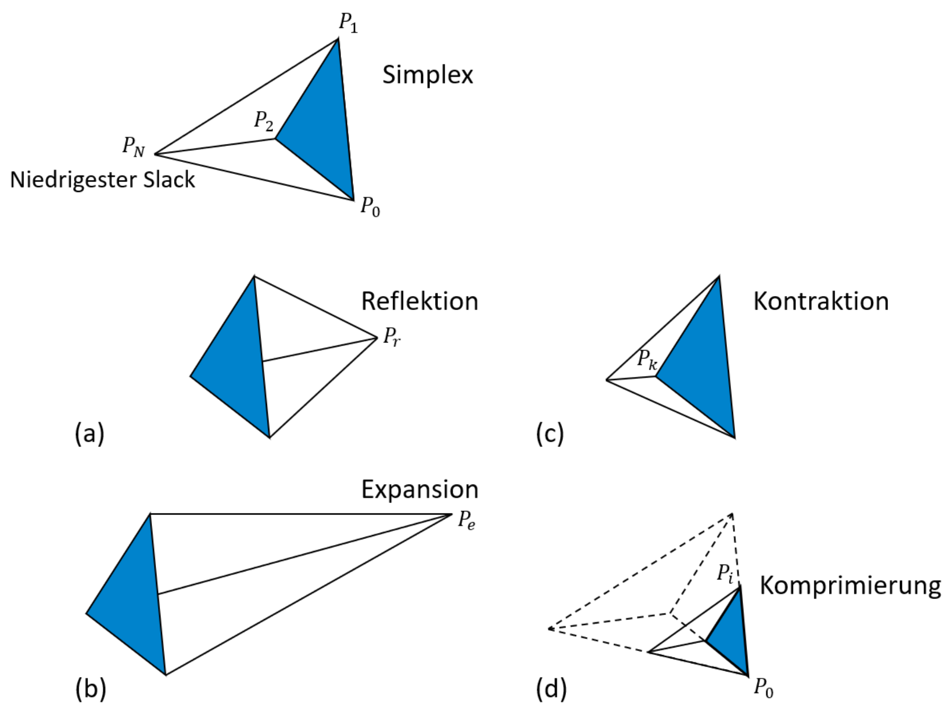


Abbildung 4.5.: Kopie aus [3]: Verschiedene Schritte des Downhill Simplex Verfahren für einen dreidimensionalen Suchraum. Das oberste Bild zeigt ein Tetraeder, welches den Simplex vor jedem Schritt aufspannt. Das Simplex nach Ende des ersten Durchlaufes wird durch (a) als Reflexion vom schlechtesten Punkt, (b) als Erweiterung vom Reflexion Punkt, (c) eine Kontraktion vom schlechtesten Punkt oder (d) eine Komprimierung jeder Dimension zum besten Punkt, ersetzt.

Das Downhill Simplex Verfahren [3, pp. 502-507] ist eine Erweiterung der Hill-

Climb Verfahren und kann auch in einem multidimensionalen Raum genutzt werden. Das Verfahren kann in einem unbekannten Raum nach einem Maximum gesucht werden, ohne, wie bei dem Gradienten verfahren [6, pp. 251-254], auf eine Ableitung angewiesen zu sein. Ein Simplex beschreibt einen Körper, der in N Dimensionen aufgespannt wird und aus N + 1 Eckpunkten besteht. Als Ausgangspunkt des Algorithmus werden zuerst N + 1 Punkte im Raum gebraucht, auf denen der Simplex aufgespannt wird. Für die Suche wird der erste dieser Punkte P_0 zufällig bestimmt. Die nachfolgenden Punkte können nun über

$$P_i = P_0 + \Delta e_i$$

festgelegt werden. e_1 bis e_n sind die Einheitsvektoren des Suchraums und die Schrittweite Δ wird in der Thesis unterschiedlich definiert. Die Konfiguration für ein Placement Constraint wird über die x- und y-Achse des Layouts bestimmt. Die Schrittweite Δ für die benötigten Anfangswerte ist ein zufälliger Wert zwischen 5% bis 10% der maximalen Breite und Höhe des Suchraums. Der Schritt wird in eine zufällige Richtung gemacht. Für die Schrittweite eines eindimensionalen Raum, wie dem Constraint für die prozentualen Verteilung von benötigten zu verwendeten Slices, wird ein zufälliger Schritt von 1% bis 10% anhand der vorhandenen Konfiguration ermittelt. Jedes einzelne Constraint erweitert den Raum somit um jeweils ein oder zwei Dimensionen. Der durch die Synthese ermittelte Slack wird als Bewertungsfunktion für einen Punkt des Simplex gewählt. Das Ziel ist es den maximal möglichen Slack zu bekommen. Nachdem die ersten N + 1 Punkte bestimmt und ein Ergebnis für jeden Punkt berechnet wurde, werden verschiedene Schritte durchlaufen. Die Schritte werden in der Abbildung 4.5 veranschaulicht. Der dargestellte dreidimensionale Raum zeigt im obersten Bild den aus vier Punkten aufgespannten Simplex. Die Punkte werden im nächsten Schritt nach der Bewertungsfunktion geordnet. P_0 wird der Punkt mit dem besten Slack zugeordnet und P_N stellt den Punkt mit dem schlechtesten Slack dar. Die Punkte 0 bis (N - 1) schließen den mit blau gefüllten Bereich ein. Über diese Punkte wird ein Mittelwert $m = 1/N \sum_{i=0}^{N-1} P_i$ gebildet. An dem Mittelwert wird nun der schlechteste Punkt reflektiert (a) $P_r = 2m – P_N$. Wenn der Slack an dem reflektierten Punkt P_r besser, als der Slack in P_0 ist, wird die Reflexion um den Faktor zwei erweitert (b) $P_e = 3m – 2P_N$. Der expandierte Punkt wird ein weiteres Mal mit dem Slack aus P_0 verglichen. Wenn der expandierte Punkt besser ist, wird der Punkt P_{N+1} durch die Expansion P_e ersetzt, sonst wird er durch den reflektierten Punkt P_r ersetzt. Falls P_r schlechter als P_0, aber besser als P_1 ist, wird P_{N+1} direkt durch P_r ersetzt. Wenn in beiden Fällen P_r schlechter ist, wird eine Kontraktion $P_k = 0,5 + (1 – 0,5) P_h$ berechnet. P_h stellt den besseren der beiden Punkte zwischen P_{N+1} und P_r dar. P_{N+1} wird ersetzt, wenn P_k einen besseren Slack aufweist. Wenn keiner der berechneten Punkte besser war, wird der Simplex komprimiert, um den Suchraum zu verkleinern. Für die Komprimierung werden alle Punkte an P_0 angenähert.

$$P_i = 0,5 * P_0 + (1 - 0,5)P_i \text{ für } i \in \{1, \ldots, N+1\}$$

Nachdem einer oder alle Punkte ersetzt wurden, werden diese wieder neu nachdem Slack sortiert und der Ablauf startet von vorne. Terminiert ein Suchvorgang, sobald der Abstand zwischen den Punkten im Simplex kleiner als vier wird. Vier ist aus dem Grund gewählt, da das die Größe von vier Slices widerspiegelt und somit an einem Punkt hängen bleibt.
Ein Problem und der Nachteil gegenüber Simulated Annealing liegt darin, in einem lokalen Maximum stecken zu bleiben. Um diesem Problem entgegen zu wirken, wird nach der Terminierung des Suchvorgangs, dieser an einem zufälligen Punkt neu gestartet, damit nach einem neuen Maximum gesucht wird. Um die Suche zu beschränken wird eine maximale Anzahl an Iterationen eingestellt, die die Suche durchlaufen soll. Dadurch entsteht ein ähnliches Problem wie bei Simulated Annealing, da die Suche genauer mit der Anzahl an Iterationen wird. Die Iteration wird nicht erhöht, wenn bei der Synthese ein Fehler aufgetreten ist. Dann wird für die Konfiguration ein unendlich negativer Slack zurückgegeben. Um die Zeit, die der Algorithmus braucht, zu verkürzen, werden in unterschiedlichen Threads mehrere Suchvorgänge unabhängig voneinander gestartet. Die Anzahl an Threads sollte die Anzahl an vorhandenen Rechenkernen nicht überschreiten. Falls eine Konfiguration der Constraints in einem Threads getestet wurde, kann das schon vorhandene Ergebnis direkt aus dem Speicher gelesen werden.
Die Anzahl an Iterationen wird auf 180 beschränkt, um eine zeitliche Einschränkung von ungefähr 20 Stunden bei der Verwendung von 6 Threads einzuhalten. Die Zeit schwankt stark abhängig von der Größe der CGRAs um mehrere Stunden.
Ein weiteres Problem liegt darin, dass jedes Constraint eine gewisse Anzahl an Konfigurationen hat. In diesem Verfahren ist die Ausbreitungsrichtung sehr starr am Simplex orientiert. Wenn ein Schritt außerhalb des Suchbereichs gemacht wird, entsteht dadurch eine nicht synthetisierbare Konfiguration. In diesem Fall wird dem Algorithmus ein unendlich negativer Slack zurück gegeben und der Zähler für die Anzahl an Iterationen nicht erhöht.
Mit Hilfe dieses Algorithmus soll mit nur wenigen Iterationen ein guten Ergebnis erzielt werden.

4.3. Implementierung der Design Space Exploration

In Abbildung 4.6 wird der Ablauf der Implementierung veranschaulicht. Im ersten Schritt werden die Verilog Dateien für einen CGRA generiert. Darauf folgt eine erste Synthese des Designs, weil für die Generierung eines PBlocks die genaue Anzahl der Ressourcen für ein HDL-Modul benötigt werden. Diese können nicht direkt aus dem generierten HDL-Code ausgelesen werden. Die Anzahl unterscheidet sich

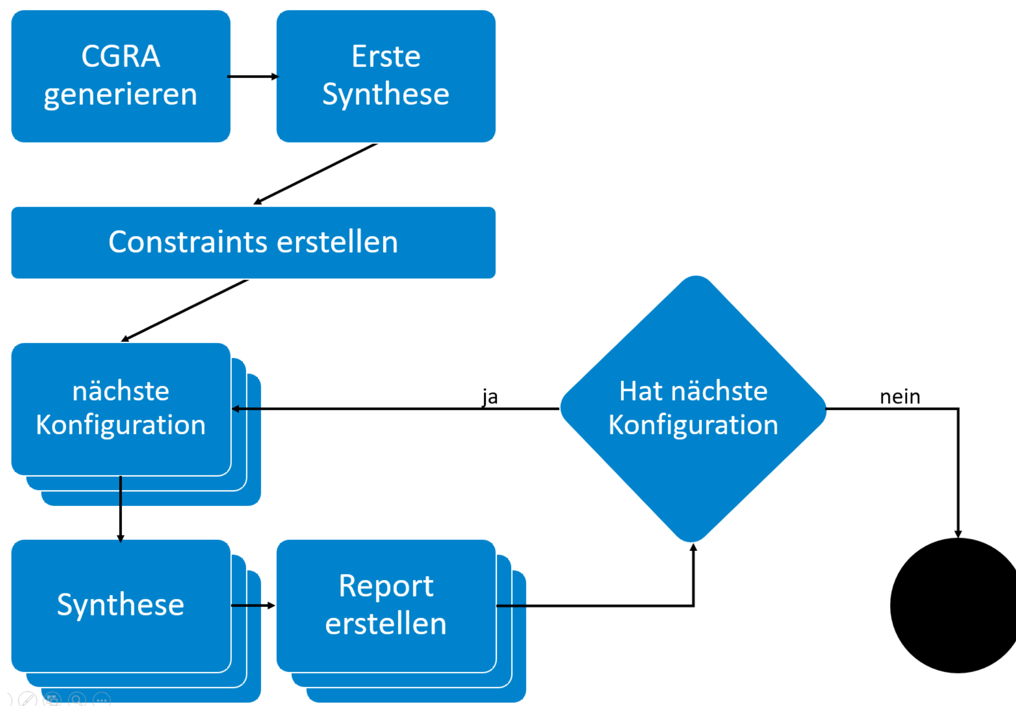


Abbildung 4.6.: Ablauf der Design Space Exploration

zusätzlich für unterschiedliche FPGAs und ist außerdem Abhängig von der konkreten CGRA Komposition. Um eine genaue Anzahl der Ressourcen zu bekommen, wird der CGRA ohne Placement Constraints in Vivado mit einer Zielfrequenz implementiert. Die Zielfrequenz ist für die erste Implementierung frei wählbar. Damit die Ressourcen für jedes einzelne Modul ausgelesen werden können, wird für jedes Modul ein eigener PBlock erstellt. Sobald ein PBlock in Vivado erstellt und implementiert wurde, kann für diesen ein Bericht erstellt werden. Das Erstellen der PBlöcke erfolgt über eine in der TCL-Datei übergebenen Schleife. Die Schleife sucht nach allen Modulen, deren Parent Modul das CGRA ist. Abschließend müssen die PBlöcke über save_constraints noch in einer XDC Datei gespeichert werden.

```
foreach x [get_cells "!IS_PRIMITIVE && (PARENT == CGRA)"] {
    create_pblock $x
    add_cells_to_pblock [get_pblocks $x] [get_cells -quiet [list $x]]
}
save_constraints
```

Nach der Implementierung kann die Laufzeit des kritischen Pfades ausgelesen werden. Die ermittelte Laufzeit stellt eine realistischere Zielfrequenz dar und wird anschließend für die spätere Suche übernommen. Aus den Berichten werden die

benötigten Ressourcen und die Bezeichnung für jedes einzelne Modul ausgelesen und in einer Liste gespeichert. Anschließend werden aus den ermittelten Ressourcen und dem verwendeten CGRA Model unterschiedliche Constraints erstellt. Für alle CGRA Modelle wird ein Placement Contraint für die platzierung eines PBlock erstellt und das Modul der cBox hinzugefügt. Der PBlock kann mit dem Modul der CCU erweitert werden. Daraufhin werden die PEs des CGRA Model miteinander verglichen. Wenn sich ein PE von den anderen Unterscheidet hat der CGRA einen heterogenen Aufbau. In diesem Fall kann zwischen drei verschiedenen Einstellungen ausgewählt werden. Diese sind nicht bestimmt, da noch nicht gesagt werden kann, wie sich die Einstellungen auf die unterschiedlichen CGRAs auswirken. Für die ersten beiden Einstellung wird das PE mit den meisten unterschiedlichen Operatoren, die ein Status Signal zurückgeben, ausgewählt. Falls dies auf mehrere PEs zutrifft, wird die Anzahl aller Operatoren verglichen. Durch diese Wahl wird das PE ermittelt, die die größte Wahrscheinlichkeit hat auf dem kritischen Pfad zu liegen. Das PE kann entweder ein eigenes Placement Constraint zugewiesen bekommen oder das PE erweitert den PBlock der cBox. Durch die Erweiterung des PBlock wird das PE immer neben der cBox platziert und führt somit zu einem kürzeren kritischen Pfad zwischen den beiden Modulen. Allerdings kann es passieren, dass ein anderes PE nun weiter von der cBox entfernt platziert werden muss und so der kritische Pfad negativ beeinflusst wird. In der letzten Einstellung wird auf keine PEs Einfluss genommen. Diese Einstellung wird auch auf homogene CGRAs angewandt. Die Placement Constraints werden noch durch eine mögliche Konfiguration für den Spielraum der PBlöcke erweitert, in Kapitel 4.1.3 beschrieben.
Über die Methode getNextConstraint(), der Klasse Contraints, wird für die erste Synthese eine zufällige Konfiguration zurückgegeben. Mit dieser Konfiguration und den dazugehörigen Constraints wird eine XDC Datei erstellt. Zu der Konfiguration wird eine eigene TCL Datei erstellt, aus Kapitel 2.3. Vivado wird über die Konsole gestartet und die TCL Datei übergeben. Nachdem die Synthese abgeschlossen ist, werden die von Vivado erstellten Berichte ausgelesen. Vivado erzeugt einen Bericht über das Zeitverhalten und einen Bericht über die Auslastung des CGRA. Aus den Berichten werden der Slack, Laufzeit, verwendeten Look-up Tables und Register ausgelesen. Zusätzlich zu diesen Daten, werden noch die jeweilige Konfiguration, die Iteration und der Thread in einer Datei gesammelt. Nach Abschluss der Synthese wird abgefragt, ob eine nächste Konfiguration vorhanden oder die festgelegte Anzahl an Iterationen erreicht ist. Wenn eine weiter Konfiguration getestet werden soll, wird der von Vivado ausgelesene Slack der Methode getNextConstraint() übergeben und anhand des verwendeten Algorithmus aus Kapitel 4.2 die nächste Konfiguration berechnet. Dieser Vorgang wird in einer Schleife wiederholt bis die festgelegte Anzahl an Iterationen erreicht ist. Die Schleife kann in mehreren Threads gleichzeitig durchlaufen werden. Mit jedem Thread wird eine eigene Schleife gestartet. Dadurch

ist die Anzahl der Threads frei wählbar. Die Daten der einzelnen Threads werden alle in der gleichen Datei gespeichert. Falls es vorkommt, das eine Konfiguration schon einmal getestet wurde, werden die gespeicherten Daten wieder ausgelesen und für die nächste Konfiguration verwendet. Wenn die gleiche Konfiguration ein drittes Mal getestet wird, startet der Thread eine neue Suche.

5. Ergebnisse

5.1. Systeminformationen

Für die Suche wurde Vivado Version 2018.2 verwendet. Vivado wurde auf einem System mit zwei Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz Prozessoren ausgeführt, welche jeweils 8 physische Kerne besitzen. Der verfügbare Arbeitsspeicher beläuft sich auf ~252 GB, wovon durchschnittlich ~110 GB ungenutzt sind. Das verwendete Betriebssystem ist Scientific Linux Carbon Version 6.8. Die Design Space Exploration wurde für einen Xilinx XC7Z030FFG676-2 FPGA, welcher zur Zynq 7000 Serie gehört, ausgeführt. Der FPGA gehört zu den größeren FPGAs der Familie. Der XC7Z030 hat 19650 Slices, 400 DSP und 265/530 36/18 kB BRAMs verbaut. Die in Java Modellierten räumlichen Ausmaße betragen 268x400.

5.2. Grundaufbau des CGRAs

Für die verschiedenen getesteten CGRAs werden die Operatoren der PEs, die Anzahl der PEs und deren Interconnect variiert. Dabei wird versucht einen einheitlichen Ressourcenbedarf der verschiedenen CGRAs zu gewährleisten. Allerdings werden im Laufe der Versuche verschiedenen Ressourcen Auslastungen untersucht. Die restlichen mit dem CGRA verbundenen Module werden nicht weiter verändert.
Der folgende Code erhält die Einstellungen des CGRA zu der verwendeten cBox und PEs. Unter „PEs:“ werden die verwendeten PE Module aufgelistet, welche in eigenen JSON Dateien beschrieben werden. Über Interconnect wird eine Datei übergeben, welche die Verbindung zwischen den PEs auflistet. Die weiteren Befehle setzten die Einstellung der cBox.

```
{
    "PEs":
    {
        "(PE_Number)" : "random_PE.json"
    },
```

```
    "Context_memory_size" : 4096,
    "Interconnect" : "File_Interconnect.json",
    "CBox_slots" : 1024,
    "CBox_evaluation_blocks" :1,
    "CBOX_output_ports_per_evaluation_blocks" : 1,
    "Pipelined" : false,
    "SecondRFOutput2ALU" : false,
    "Branch_selection_mode" : "default",
    "Branch_selection_sources" : ["regOutOrNegative"],
    "RomSize" : 32
}
```

In der Hauptdatei werden die Busweite und die Speichergröße festgelegt. Der CGRA Generator erstellt noch zusätzliche Peripherie, welche auf dem FPGA an dem CGRA verschaltet sind. Die Peripherie wird für den Versuch allerdings nicht weiter betrachtet. Die Einstellungen für die Speichergröße der Module kann aus dem JASON-Code entnommen werden.

```
{
    "name" : "nameofFile",
        "composition" : "CGRA_composition.json",
        "runCounterWidth" : 32,
        "cycleCounterWidth" : 32,
        "parameterBufferSize" : 4,
        "parameterBufferMaxExpectedParameterCount" : 3,
        "IDTableSize" : 1024,
        "actorCount" : 128,
        "sensorCount" : 128,
        "ocmDataWidth" : 64,
        "ocmBufferSize" : 1024,
        "globalLogContextSize" : 1024,
        "logAxiOutputContextSize" : 32,
        "ocmOutputContextSize" : 256,
        "ocmAxiOutputContextSize" : 256,
        "logs" :
        {
        "logSize(PE_Number)" : 64
        }
}
```

Für die Versuche wurden sechs verschiedene PE Konfigurationen erstellt. Drei der Konfigurationen besitzen sechs verschiedene Module für ein Status Signal. Die an-

deren drei Konfigurationen sind das Äquivalent zu diesen, nur verwenden diese kein Status Signal. Die PEs verwenden alle 64 Bit Operatoren und sind aufgeteilt in Floating Point, Integer und einer gemischten Implementierung.

5.3. Versuche

In diesem Kapitel werden für vier verschiedene CGRAs insgesamt 10 Versuche gemacht. In den Unterkapiteln wird der Aufbau des CGRAs kurz beschrieben. Darauf folgen ein paar Informationen, die aus der ersten Synthese des CGRAs entnommen wurden. Zu jedem Versuch werden die verwendeten Direktiven und Module aufgelistet und wie viele Ressourcen jeder PBlock braucht. Darauf folgt die beste Laufzeit und die prozentuale Verbesserung zur Laufzeit ohne Optimierung. In den Tabellen werden die vier besten Ergebnisse der Suche aufgelistet.
In den Tabellen werden immer der Slack, in welcher Iteration und der Thread in dem die Suche erfolgt ist und die Ausgangslaufzeit gezeigt. Zusätzlich werden die Module, die verändert wurden, aufgelistet. Jede Spalte, in denen die Koordinaten x und y enthalten sind, stellt einen freien PBlock dar. Falls eine Spalte zwei unterschiedliche Namen von Modulen enthält, gehören die Module nur einem PBlock an. In der Spalte sind die PBlock-Koordinaten des PBlocks aufgelistet. In dem Versuch ist die Ausnutzung der PBlöcke veränderbar. Der angegebene Wert stellt den prozentualen Anteil an Slices dar, die ein PBlock benötigt, um synthetisiert zu werden. Die Anordnung der PEs ist in jedem Versuch ein Kreis. Dadurch ist ein PE direkt mit zwei anderen PEs verbunden.
In den ersten Synthesen werden die CGRAs ohne Optimierung mit einer Zeiteinschränkung von 25 ns implementiert. Jede Suche wurde auf sechs Threads gleichzeitig ausgeführt.

5.3.1. Heterogener CGRA mit einem Status Signal

Der CGRA ist Heterogen aufgebaut und besteht aus zehn PEs. Von den 10 PEs sind sechs PEs mit Integer und vier PEs mit Floating Point Operatoren bestückt. Von den PEs besitzt nur PE fünf, Module zur Berechnung eines Status Signals. Von den auf dem FPGA vorhandenen Look-Up Tables werden 66% verwendet. Der CGRA braucht für die Synthese 14653 Slices, 82 36 kBRAM, 24 18 kBRAM und 220 DSP. Nach der ersten Synthese wurde eine Laufzeit von 14.996 ns ermittelt, welcher für die nachfolgenden Synthesen als Frequenzvorgabe übernommen wurde. Für diese Frequenz liegt der Slack bei 2.218 ns und die Laufzeit bei 12.778 ns. Der CGRA wurde in drei Varianten getestet.

Floorplanning der cBox

In diesem Versuch wurde die Position der cBox gesucht. Die cBox benötigt auf dem CGRA 817 Slices, 6 36 kBRAM und 1 18 kBRAM. Es wurden 214 Konfigurationen getestet. Mit dem besten Slack von 3.178 ns wird eine Laufzeit von 11.818 ns erreicht. Der Slack ist dadruch um 43,28% höher und die Laufzeit um 7,5% kürzer. Die Ausnutzung des PBlock liegt bei 52%. Der Versuch hat 18 Stunden und drei Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.1 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		Modul Größe	Timing
			x	y	Ausnutzung	
3,178	209	5	30	155	0,521	14,996
3,17	168	0	80	275	0,816	14,996
3,142	140	2	166	132	0,675	14,996
3,135	34	4	148	164	0,762	14,996

Tabelle 5.1.: Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position eines PBlock verändert. In dem PBlock ist die cBox enthalten.

Floorplanning mit cBox und freiem PE

Für diesen Versuch wurde ein weiterer PBlock der Suche hinzugefügt. In diesem PBlock ist PE fünf enthalten. Das PE benötigt 3707 Slices, 4 36 kBRAM, 1 18 kBRAM und 37 DSP. Es wurden 196 verschiedene Konfigurationen getestet. Mit dem besten Slack von 3.006 ns wird eine Laufzeit von 11.99 ns erreicht. Der Slack ist dadruch um 35,52% höher und die Laufzeit um 6,16% kürzer. Die Ausnutzung des PBlock liegt bei 75%. Der Versuch wurde unerwartet unterbrochen, daher ist keine Länge des Versuches bekannt. Allerdings wurde aufgrund der Anzahl an schon getestet

Konfigurationen, der Versuch nicht wiederholt. Die besten vier Ergebnisse können aus Tabelle 5.2 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		i_pe5		Module Größe	Timing
			x	y	x	y	Ausnutzung	
3,006	142	4	90	37	72	116	0,751	14,996
3,001	81	3	38	6	26	128	0,741	14,996
2,974	20	2	119	47	53	92	0,5	14,996
2,938	109	1	131	192	58	44	0,614	14,996

Tabelle 5.2.: Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE5 enthalten.

Floorplanning der cBox mit eingeschlossenem PE

Die Module beider PBlöcke wurden in diesem Versuch in einen PBlock zusammengeführt. Dadurch addiert sich die benötigte Anzahl der Ressourcen für den PBlock auf 4524 Slices, 10 36 kBRAM, 2 18 kBRAM und 37 DSP. Es wurden 198 Konfigurationen getestet. Mit dem besten Slack von 3.17 ns wird eine Laufzeit von 11.826 ns erreicht. Der Slack ist dadruch um 42,92% höher und die Laufzeit um 7,45% kürzer. Die Ausnutzung des PBlock liegt bei 86%. Der Versuch hat 17 Stunden und 57 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.3 entnommen werden.

Slack	Iteration	Thread	i_pe5cBoxWrapper		Module Größe	Timing
			x	y	Ausnutzung	
3,17	97	1	65	186	0,862	14,996
3,02	9	3	38	163	0,689	14,996
3,011	134	2	63	91	0,725	14,996
2,97	106	4	78	148	0,714	14,996

Tabelle 5.3.: Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position eines PBlock verändert. In dem PBlock sind die cBox und PE5 enthalten.

5.3.2. Heterogener CGRA mit drei Status Signalen

Der CGRA ist fast gleich dem CGRA aus dem vorherigen Kapitel. Diesem CGRA wurden zwei Module hinzugefügt, die ein Status Signal besitzen. Die PEs die ein Status Signal besitzen sind eins, fünf und sieben. Vom FPGA werden in dieser Konstellation 72,2% der vorhandenen Look-Up Tables verwendet. Von den verfügbaren Ressourcen werden 15620 Slices, 82 36 kBRAM, 24 18 kBRAM und 220 DSP gebraucht. Bei der ersten Synthese wurde eine Laufzeit von 17.88 ns ermittelt. Für diese Laufzeit liegt der Slack bei 2.441 ns und die Laufzeit bei 15.439 ns.

Floorplanning mit cBox und freiem PE

Für die Suche wurde die cBox und das PE eins getrennt in jeweils einem PBlock hinzugefügt. Das PE benötigt 484 Slices, 4 36 kBRAM, 1 18 kBRAM und 12 DSP und die cBox 808 Slices, 6 36 kBRAM und 1 18 kBRAM. Es wurden 189 Konfigurationen getestet. Mit dem besten Slack von 3.906 ns wird eine Laufzeit von 13.974 ns erreicht. Der Slack ist dadurch um 60% höher und die Laufzeit um 9,48% kürzer. Die Ausnutzung des PBlock liegt bei 56,8%. Die Suche hat 18 Stunden und 19 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.4 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		i_pe1		Module Größe	Timing
			x	y	x	y	Ausnutzung	
3,906	166	4	121	149	59	60	0,568	17,88
3,676	15	3	121	214	102	336	0,8	17,88
3,655	79	1	179	177	130	244	0,688	17,88
3,573	42	0	126	202	144	318	0,921	17,88

Tabelle 5.4.: Besten Ergebnisse der Teiloptimierung eines CGRAs mit drei Status Signalen. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE1 enthalten.

Floorplanning mit CCU, cBox und freiem PE

Der PBlock für die cBox wurde mit dem Module der CCU erweitert. Die CCU braucht 73 Slices und 2 36 kBRAM. Für diesen Versuch wurde eine Zeitvorgabe auf 9.5 ns gesetzt. Dadurch ist der Slack bei der Synthese ohne Optimierung negativ bei −1.207 ns und die Laufzeit beträgt 10.707 ns. Es wurden 206 Konfigurationen getestet. Mit dem besten Slack von −0.841 ns wird eine Laufzeit von 10.341 ns erreicht. Der Slack ist dadurch um 43% höher und die Laufzeit um 3,42% kürzer. Die Ausnutzung des PBlock liegt bei 73,2%. Die Suche hat 22 Stunden und 52 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.5 entnommen werden.

Slack	Iteration	Thread	control_cBox		i_pe1		Module Größe	Timing
			x	y	x	y	Ausnutzung	
-0,841	83	5	40	85	129	83	0,732	9,5
-0,857	155	5	31	41	119	106	0,715	9,5
-0,962	69	3	77	155	171	119	0,596	9,5
-0,967	172	4	143	172	37	108	0,555	9,5

Tabelle 5.5.: Besten Ergebnisse der Teiloptimierung eines CGRAs mit drei Status Signalen. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und die CCU, und im Anderen das PE1 enthalten.

5.3.3. Heterogener CGRA mit allen Status Signalen

Der CGRA hat einen heterogenen Aufgebaut und besteht aus acht PEs. Von den acht PEs sind fünf PEs mit Integer und drei PEs mit Floating Point Operatoren bestückt. Um die verwendeten Look-Up Tables in einem ähnlichen Bereich zu bringen, wie bei den vorherigen Versuche, wurden zwei PEs weniger verbaut. Die Ausnutzung des FPGA liegt für diesen CGRA bei 66,5% der vorhandenen Look-Up Tables. Von den verfügbaren Ressourcen wurden 14013 Slices, 68 36 kBRAM, 20 18 kBRAM und 171 DSP benutzt. Bei der ersten Synthese wurde ein Laufzeit von 18.528 ns ermittelt. Für diese Laufzeit liegt der Slack bei 2.717 ns und die Laufzeit bei 15.811 ns. Der CGRA wurde mit zwei verschiedenen Optionen getestet.

Floorplanning der cBox mit eingeschlossenem PE

Für die Suche wurde die cBox und das PE null getrennt in jeweils einem PBlock hinzugefügt. PE0 benötigt 487 Slices, 4 36 kBRAM, 1 18 kBRAM und 12 DSP und die cBox 789 Slices, 6 36 kBRAM und 1 18 kBRAM. Es wurden 208 Konfigurationen getestet. Mit dem besten Slack von 3.793 ns wird eine Laufzeit von 14.735 ns erreicht. Der Slack ist dadurch um 39,6% höher und die Laufzeit um 6,8% kürzer. Die Suche hat 20 Stunden und 50 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.6 entnommen werden.

Slack	Iteration	Thread	i_pe0cBoxWrapper		Module Größe	Timing
			x	y	Ausnutzung	
3,793	134	2	43	35	0,797	18,528
3,724	120	0	161	68	0,65	18,528
3,698	38	2	44	34	0,829	18,528
3,692	108	0	147	93	0,629	18,528

Tabelle 5.6.: Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von einem PBlock verändert. In dem PBlock ist die cBox und das PE0 enthalten.

Floorplanning mit cBox und freiem PE

Für die Suche wurde die cBox und das PE null in ein PBlock gesteckt. Der PBlock benötigt somit 1276 Slices, 10 36 kBRAM, 2 18 kBRAM und 12 DSP. Es wurden 169 Konfigurationen getestet. Mit dem besten Slack von 4.016 ns wird eine Laufzeit von 14.512 ns erreicht. Der Slack ist dadurch um 47,8% höher und die Laufzeit um 8,2% kürzer. Die Suche hat 17 Stunden und 15 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.7 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		i_pe0		Module Größe	Timing
			x	y	x	y	Ausnutzung	
4,016	100	4	77	64	134	36	0,516	18,528
4,016	184	4	77	64	134	35	0,517	18,528
4,016	202	4	77	64	134	35	0,516	18,528
3,873	114	0	82	73	127	297	0,617	18,528

Tabelle 5.7.: Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE0 enthalten.

Floorplanning mit CCU

In diesem Versuch wurde dem PBlock der cBox die CCU hinzugefügt. PE null ist Bestandteil eines getrennten PBlock. Die CCU benötigt 20 Slices und 2 36 kBRAM. Es wurden 204 Konfigurationen getestet. Mit dem besten Slack von 4.078 ns wird eine Laufzeit von 14.45 ns erreicht. Der Slack ist dadurch um 50% höher und die Laufzeit um 8,6% kürzer. Die Suche hat 20 Stunden und 30 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.8 entnommen werden.

Slack	Iteration	Thread	control_cBox		i_pe0		Module Größe	Timing
			x	y	x	y	Ausnutzung	
4,078	28	4	58	43	173	107	0,5	18,528
3,904	196	4	53	35	167	48	0,584	18,528
3,815	100	4	34	32	172	85	0,781	18,528
3,715	165	3	39	69	103	169	0,62	18,528

Tabelle 5.8.: Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und die CCU, und im Anderen das PE0 enthalten..

Floorplanning mit cBox und freiem PE, CGRA mit 90% Auslastung

In diesem Versuch wurde der CGRA durch eine weiteres Floating Point PE ergänzt. Die Auslastung des PEs steigt damit auf 92,93% verwendete Look-Up-Tables an. Dadurch benötigt der CGRA 19455 Slices, 76 36 kBRAM, 21 18 kBRAM und 208 DSP. Die cBox und das PE null wurden zwei getrennten PBlöcken zugeordnet. PE null braucht 935 Slices, 4 36 kBRAM, 1 18 kBRAM und 12 DSP und die cBox 784 Slices und 7 36 kBRAM. In der ersten Synthese wurde eine Laufzeit von 19.372 ns ermittelt. Der Slack liegt ohne Optimierung bei 2.797 ns und die Laufzeit bei 16.575 ns. Für diesen CGRA wurden 219 Konfigurationen getestet. Mit dem besten Slack von 4.072 ns wird eine Laufzeit von 15.3 ns erreicht. Der Slack ist dadurch um 45,58% höher und die Laufzeit um 7,69% kürzer. Die Suche hat 27 Stunden und 9 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.9 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		i_pe0		Module Größe	Timing
			x	y	x	y	Ausnutzung	
4,072	77	5	131	215	37	112	0,805	19,372
4,019	204	0	63	123	18	74	0,572	19,372
4,016	203	5	128	251	35	146	0,786	19,372
3,995	174	0	97	139	51	87	0,763	19,372

Tabelle 5.9.: Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE0 enthalten.

5.3.4. Homogener CGRA

Im letzten Versuch wurde ein homogener CGRA getestet. Die PEs auf einem homogenen CGRA sind alle gleich und besitzen somit alle ein Status Signal. Auf dem CGRA wurden 9 PEs verbaut. Die PEs besitzen alle eine Mischung von Integer und Floating Point Operatoren. Die Auslastung des FPGA liegt bei 67,72%. Der CGRA benötigt 14920 Slices, 76 36 kBRAM, 21 18 kBRAM und 396 DSP. Die cBox braucht 802 Slices und 7 36 kBRAM. Der FPGA wird mit einer Laufzeit von 20.619 ns betrieben. Ohne Optimierung liegt der Slack bei 3.456 ns. In diesem Versuch wurde nur die Position der cBox optimiert. Dafür wurden 190 Konfigurationen getestet. Mit dem besten Slack von 4.544 ns wird eine Laufzeit von 17.163 ns erreicht. Der Slack ist dadurch um 31,4% höher und die Laufzeit um 6,3% kürzer. Die Suche hat 20 Stunden und 19 Minuten gedauert. Die besten vier Ergebnisse können aus Tabelle 5.10 entnommen werden.

Slack	Iteration	Thread	cBoxWrapper		Module Größe	Timing
			x	y	Ausnutzung	
4,544	87	3	83	160	0,786	20,519
4,278	151	1	155	178	0,861	20,519
4,189	41	5	66	132	0,541	20,519
4,142	127	1	123	61	0,595	20,519

Tabelle 5.10.: Besten Ergebnisse der Teiloptimierung eines homogenen CGRAs. Es wurde die Position von einem PBlock verändert. In dem PBlock ist die cBox enthalten.

5.3.5. Graphische Abbildungen ausgewählter Suchläufe

In Abbildung 5.1 wird ein Ausschnitt der in Kapitel 5.3.4 getätigten Suche dargestellt. Die x-Achse gibt die Iterationen in der durchgeführten Reihenfolge von Thread drei an. In der y-Achse ist der erreichte Slack angegeben.

Die Graphen aus Abbildung 5.2 zeigen die gesetzten PBlock-Koordinaten der dreidimensionalen Suchräume an. Die Abbildungen sind im Anhang in einer größeren Darstellung angefügt. Der erreichte Slack für die jeweiligen PBlock-Koordinaten wird in einem Farbschema dargestellt. Wenn ein Punkt rot markiert ist, liegt der Punkt in der Nähe des höchsten Slack der Suche. Die Ausnutzung der PBlöcke fließen nicht in die Informationen der Graphen ein.
In den Graphen 5.2a und 5.2b stellen zwei aus dem Kapitel 5.3.1 enthaltene Suchen dar. Graph 5.2a enthält alle PBlock-Koordinaten für einen PBlock in dem die Module der cBox und das PE fünf eingeschlossen sind. In Graph 5.2b enthält der PBlock nur die cBox.
Graph 5.2c stellt die PBlock-Koordinaten für den PBlock, in dem die Module der cBox und das PE null enthalten sind, auf dem CGRA aus Kapitel 5.3.3 dar. Der letzte Graph 5.2d bezieht sich auf die Suche des homogenen CGRA aus 5.3.4.

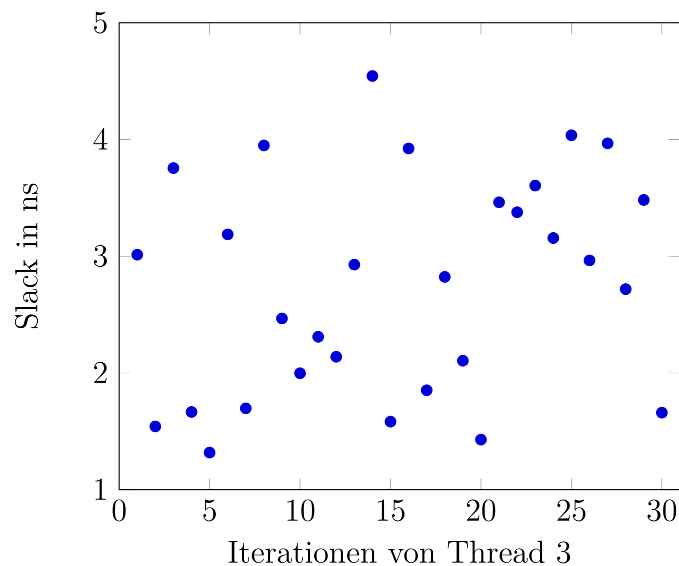
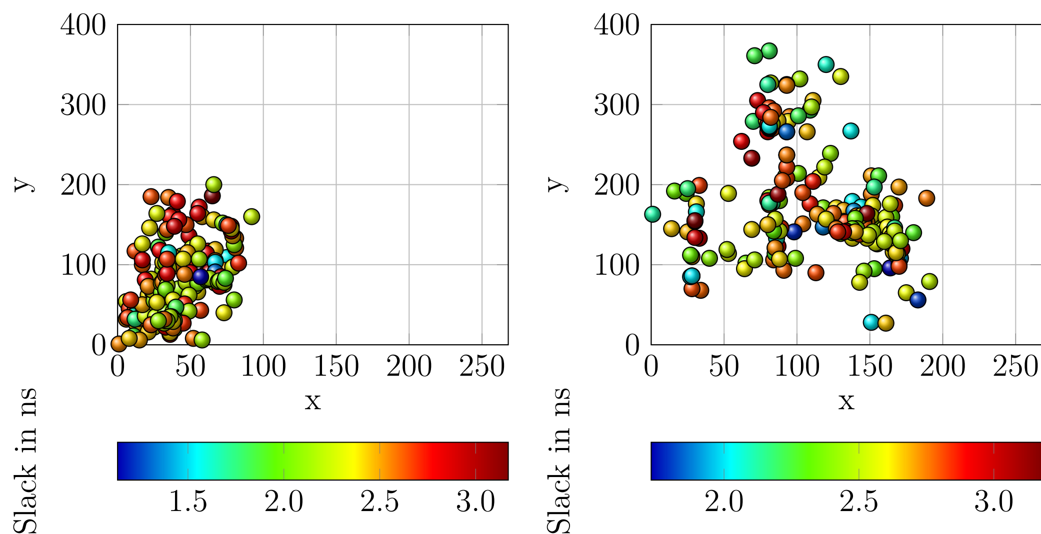
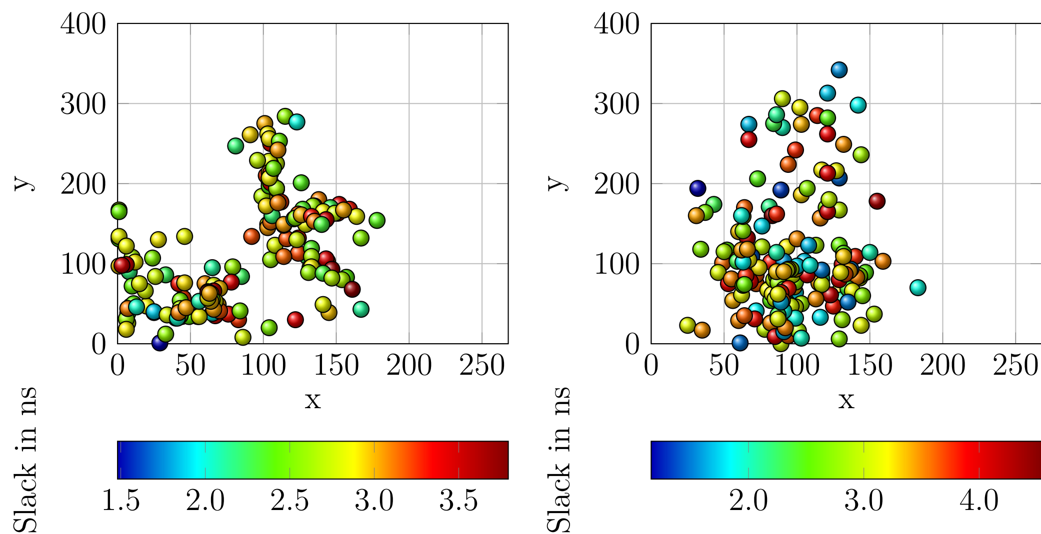


Abbildung 5.1.: Auszug der Suche des Homogenen CGRA aus 5.3.4. Der Slack wird Anhand der Schritte des Algorithmus dargestellt. Die Schritte werden nur vom dritten Thread gezeigt.



(a) CGRA mit einem Status Signal, PE und cBox in einem PBlock, aus 5.3.1

(b) CGRA mit einem Status Signal, cBox in einem PBlock, aus 5.3.1



(c) CGRA mit allen Status Singalen, PE und cBox in einem PBlock, aus 5.3.3

(d) Homogener CGRA, cBox in einem PBlock, aus 5.3.4

Abbildung 5.2.: Auf den Graphen sind alle PBlock-Koordinaten markiert, die für die Optimierung der verschiedenen CGRAs synthetisiert wurden. Über die Farbskalar wird der erreichte Slack der Koordinate gezeigt.

6. Diskussion

Durch die Teiloptimierung des CGRA wurde, gegenüber der Synthese ohne Optimierung, in jeder Suche eine kürzere Laufzeit und somit eine höhere Frequenz erzielt. Die Laufzeit wurde zwischen 3,42% und 9,48% verringert und der Slack um 30% bis 60% gesteigert.
Schon vor der 100. Iteration wurde ein Slack gefunden, der unter den besten vier Werten lag. Dies war in allen Suchräumen der Fall, obwohl sich die Anzahl der Konfigurationen zwischen den Suchräumen stark unterschieden haben. Die Möglichkeiten steigern sich mit jeder Dimension. Die Größe des Raumes ist bei fünf Dimensionen um einiges größer. Dies ist der Fall, wenn zwei verschiedene PBlöcke platziert werden. Durch den vorgegebenen Richtwert von 200 Iterationen pro Suche wurde sich an der zeitlichen Länge der Suche orientiert. Durch die Beschränkung kann nicht mit Sicherheit gesagt werden, ob die maximale Optimierung erreicht wurde. Dafür müsste jede Konfiguration getestet werden. Trotz der Limitierung werden für die unterschiedlichen CGRAs sehr ähnliche Laufzeiten erreicht.
In den Graphen 5.2 wird jede getestete PBlock-Koordinate einer Suche graphisch abgebildet. Die Graphen zeigen nur die Suchen mit einem Placement Constraint, da eine dreidimensionale Darstellung besser abzubilden ist. Über das Farbschema ist zu erkennen, wie hoch der Slack auf den jeweiligen Koordinaten ist. Über die verschiedenen Farbpunkte können die lokalen Maxima und Minima abgelesen werden. In dem Koordinatensystem werden eine hohe Anzahl an Maxima und Minima veranschaulicht. Dabei liegen diese sehr dicht aneinander. Dies erschwert die Suche nach einem besseren Slack, da eine Schwäche des Downhill Simplex Verfahren die lokalen Extremstellen sind. Der Algorithmus kann leicht in den lokalen Maxima stecken bleiben oder in den ersten Synthesen durch die große Schrittweite ein paar Maxima übergehen.
Aus den Graphen 5.2 kann erkannt werden, dass sich die meisten Punkte zwischen $x = 0$ und $x = 180$ befinden. Dies liegt allerdings daran, dass der PBlock in positive x- und y- Richtung aufgespannt wird. Eine andere Annahme kann für die Platzierung der PBlöcke anhand dieser Graphen nicht getroffen werden.
Der Graph 5.1 zeigt den Slack nach den verschiedenen Suchdurchläufen des Algorithmus aus einem Thread. Der Slack weißt ein sehr inkonsistentes Verhalten auf, wodurch die Suche nach einem Maxima erschwert wird. Die Tabellen mit den Ergeb-

nissen zeigen die Anzahl an verschiedenen Threads die zu einem guten Slack führen. Dies bestätigt die Aussage, dass der Algorithmus in lokalen Maxima stecken bleibt, sofern sich diese nicht stark voneinander unterscheiden.
Jedem PBlock können mehr Slices zugewiesen werden, als die Module benötigten. Unter den besten Laufzeiten liegt der prozentuale Anteil benötigter Slices unter 90%. Die Werte unter 90% variieren stark und es kann keine Abhängigkeit zu einer bestimmten Einstellung erkannt werden.
Für einen Vergleich wurde auf das gleiche Design von Kapitel 5.3.2 eine von Vivado bereitgestellte Optimierungsoption angewandt. Die Option, phys_opt_design, kann nur angewandt werden, wenn der Slack kleiner als null ist. Die Option wurde mit den Standard Werten ausgeführt. Der Slack lag nach der Optimierung von Vivado bei -1.047 ns. Der erreichte Slack in dieser Arbeit liegt bei -0.845 ns und ist somit niedriger. Die Ziellaufzeit liegt bei 9.5 ns und die erreichte Standardlaufzeit bei 10.707 ns. Die Laufzeit mit der Optimierung durch Vivado wurde um 1,34% verbessert. Durch die Teiloptimierung in dieser Arbeit wurde eine Verbesserung von 3,42% im Vergleich zur Standardlaufzeit erzielt. Dies entspricht einer 2,62-fachen Verbesserung zu Vivado. Jedoch muss auch die Zeit zur Berechnung der Optimierung betrachtet werden. Die Optimierung durch Vivado hat nur 24 Minuten gedauert und ist damit 56-mal schneller als die Teiloptimierung die in 22 Stunden und 52 Minuten berechnet wurde.
Für einen CGRA, in dem nur ein PE ein Status Signal besitzt, steigert die Verwendung von nur einem PBlock, für PE und die cBox, die Laufzeit um 7,46%. Im Vergleich liegt die Verbesserung, bei der Trennung der Module auf zwei PBlöcke, bei nur 6,16%. Im Gegensatz dazu ist für einen CGRA, in dem alle PEs ein Status Signal besitzen, das Gegenteil der Fall. Hier liegt die Steigerung der Laufzeit bei einem PBlock bei 6,8% und bei zwei PBlöcken bei 8,2%.
Eine erhöhte Auslastung des FPGA führt zu ähnlichen Ergebnissen, wie schon bei den anderen CGRAs. Die Synthesen brauchen bei einer höheren Auslastung länger, was eine längere Suche zur Folge hat.

7. Fazit

Ziel dieser Arbeit war es, die Frequenz eines CGRA Generators erstellten Designs zu optimieren und den Einfluss verschiedener Module auf die Frequenz zu untersuchen. Hierfür war die Programmierung eines Java-Frameworks nötig. Der Verlauf der Arbeit lässt sich wie folgt kurz zusammenfassen.
Die wichtigsten theoretischen Hintergründe wurden in Kapitel 2 erläutert. Es wurde der grundlegende Ablauf der CGRA Architektur veranschaulicht. Die zentralen Module eines CGRAs sind die cBox, die CCU und die PEs. Darüber hinaus sind die Abhängigkeiten der einzelnen Module aufgezeigt. Im weiteren Verlauf des Kapitels 2.2 wird die Synthese durch Vivado näher betrachtet und der Implementierung des CGRA auf einem FPGA. In Kapitel 2.3 auf die im Java-Framework enthaltene Schnittstelle zwischen Vivado und Java eingegangen, die im Laufe der Arbeit erweitert wurde.
In Kapitel 3 wurde in [1] ein genereller floorplanning Ansatz für einen FPGA vorgestellt. Teile der Definition des FPGA Layout wurden in der Arbeit übernommen. Des Weiteren wurde eine Option von Vivado vorgestellt, mit welcher eine Optimierung des FPGA nach der Implementierung gemacht werden kann.
Schließlich wird in Kapitel 4 der praktische Teil der Arbeit in Java erklärt. Der Ablauf lässt sich wie folgt beschreiben. Zuerst wurde das Layout eines FPGAs in Java modelliert. Hierfür wurde ein Array angelegt, welches ein Koordinatensystem darstellt, das die verschiedenen Ressourcentypen eines FPGAs auf den Koordinaten anordnet. Eine Besonderheit des gewählten FPGAs ist, dass sich zwei Ressourcentypen jeweils die gleiche Koordinate teilen. Dies erschwerte das Ansteuern der Speicherressource.
Im Anschluss an die Modellierung des FPGAs wurden die Eigenschaften der PBlöcke, einer Floorplanning Option aus Vivado, definiert. Einem PBlock werden ein oder mehrere HDL-Module zugeordnet. Durch den PBlock werden, ausgehen von einer freiwählbaren Koordinate des Layouts und der benötigten Ressourcenanzahl der zugeordneten Module, eine Auswahl an Zellen bestimmt werden, auf die die Module bei der Implementierung platziert werden müssen.
Anschließend wurde der kritische Pfad analysiert und versucht, diesen zu optimieren. Anhand der CGRA Architektur wurden drei mögliche kritische Pfade aufgelistet. Der kritische Pfad wird in den meisten Fällen durch die cBox und die CCU

beeinflusst. Teil des kritischen Pfades stellt das Status Signal dar, wodurch eine mit der cBox verbundenen PE auf dem kritischen Pfad liegt. Dieser Einfluss wurde in 4.1.3 genauer analysiert. Man fand heraus, dass eine Platzierung der Control Box großen Einfluss auf die Verbesserung des kritischen Pfades hat. Darüber hinaus wurden weitere Kombinationen von Modulen betrachtet und implementiert, die den kritischen Pfad verkürzen können.
In Kapitel 4.2 wurde das Downhill Simplex Verfahren beschrieben. Mit diesem Algorithmus wird eine gezielte Suche, für die Platzierung der PBlöcke, ermöglicht. Anhand einer besseren Frequenz wird eine Konfiguration als gut empfunden. Dieses wurde in die Programmierung nachfolgend eingebunden.
In 4.3 wurde der Ablauf der Design Space Exploration und die Wahl der Constraints, anhand des generierten CGRA, erklärt. Für jeden CGRA wird ein PBlock mit dem Modul der cBox erstellt. Wenn der Aufbau des CGRA heterogen ist, wird Anhand der Anzahl an Operatoren die ein Status Signal besitzen, ein PE bestimmt. Dieses kann entweder in den PBlock der cBox hinzugefügt werden oder die Suche um einen weiteren PBlock erweitern. Beide Einstellungen haben zu besseren Ergebnissen bei unterschiedlichen CGRAs geführt. Jede Suche kann mit einer davor festgelegten Anzahl an Threads durchlaufen werden. Die Threads durchsuchen den Raum unabhängig voneinander, sammeln die Ergebnisse allerdings in einer einzelnen Datei.
In Kapitel 5 wurden die Ergebnisse der Arbeit präsentiert. Für die Analyse wurden fünf verschiedene CGRAs generiert und mit unterschiedlichen Einstellungen der PBlock Konstellationen getestet. Jede Such hat zu einer höheren Taktrate geführt. Die Laufzeit des kritischen Pfades wurde zwischen 3,42% und 9,48% kürzer. Den meisten Einfluss auf die Implementierung wurde durch die Platzierung der cBox erzielt. Für einen CGRA mit nur einer PE mit Status Signal, wurde ein kürzere Laufzeit erzielt, wenn die cBox und das PE in einem PBlock enthalten sind. Im Gegensatz dazu ist die Laufzeit kürzer für einen CGRA in dem alle PEs ein Status Signal haben, wenn das PE unabhängig von der cBox platziert wird. Die getesteten Konfigurationen jeder Suche lagen zwischen 170 und 210 Iterationen und haben um die 20 Stunden in Anspruch genommen.
Zur weiteren Vertiefung wäre interessant zu testen, wie sich das Floorplanning des Ganzen Designs auf den kritischen Pfad auswirkt. Auch wären Versuche auf einem anderen FPGA interessant. Die vorgegebene Taktrate hat einen gewissen Einfluss auf die Implementierung, daher wäre es interessant wie sich eine Veränderung der Taktrate auf die platzierten Module auswirkt. Die Teiloptimierung könnte mit den Optimierungsoptionen für die Implementierung von Vivado erweitert werden.
Abschließend sei noch zusagen, dass durch die Teiloptimierung das Ziel einer höheren Taktrate erreicht wurde. Allerdings muss aufgrund der langen und rechenlastigen Suche evaluiert werden, ob die Verbesserung für die verwendete Anwendung ausreichend ist.

A. Appendix

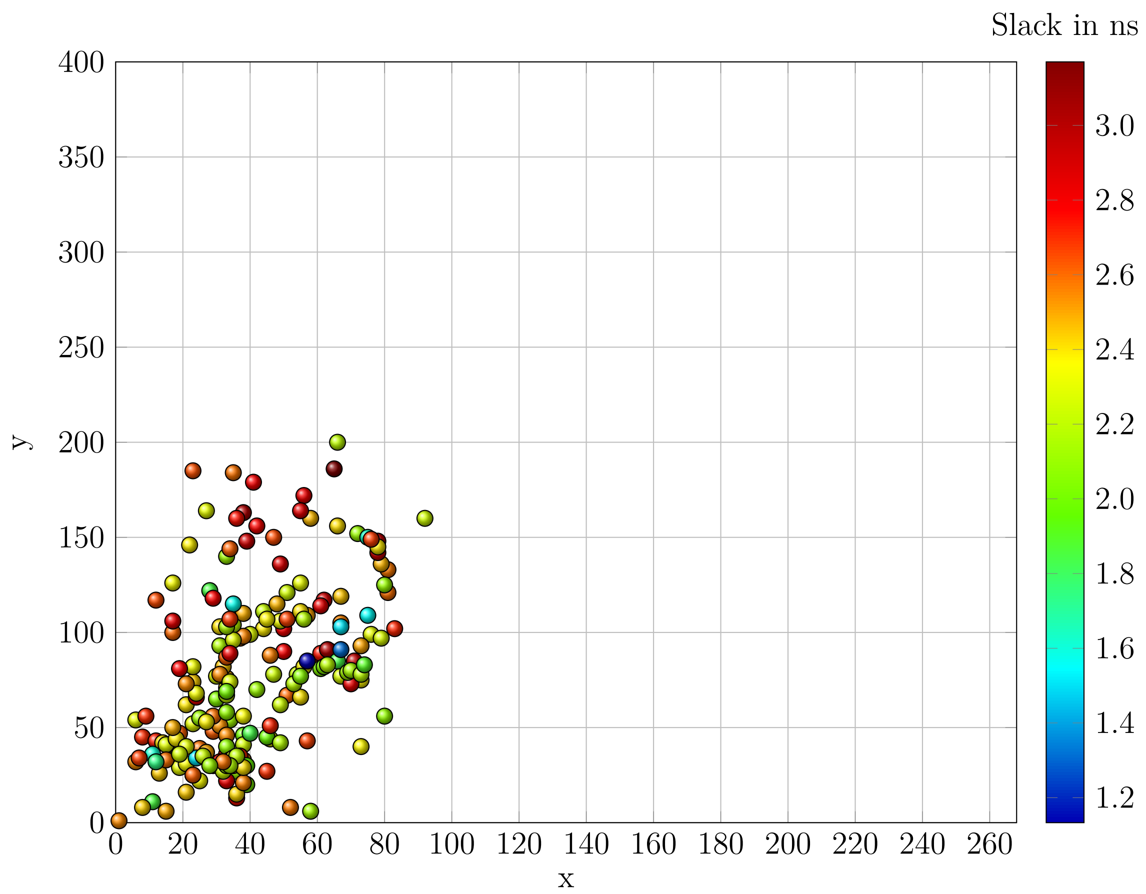


Abbildung A.1.: CGRA mit einem Status Signal, PE und cBox in einem PBlock, aus 5.3.1

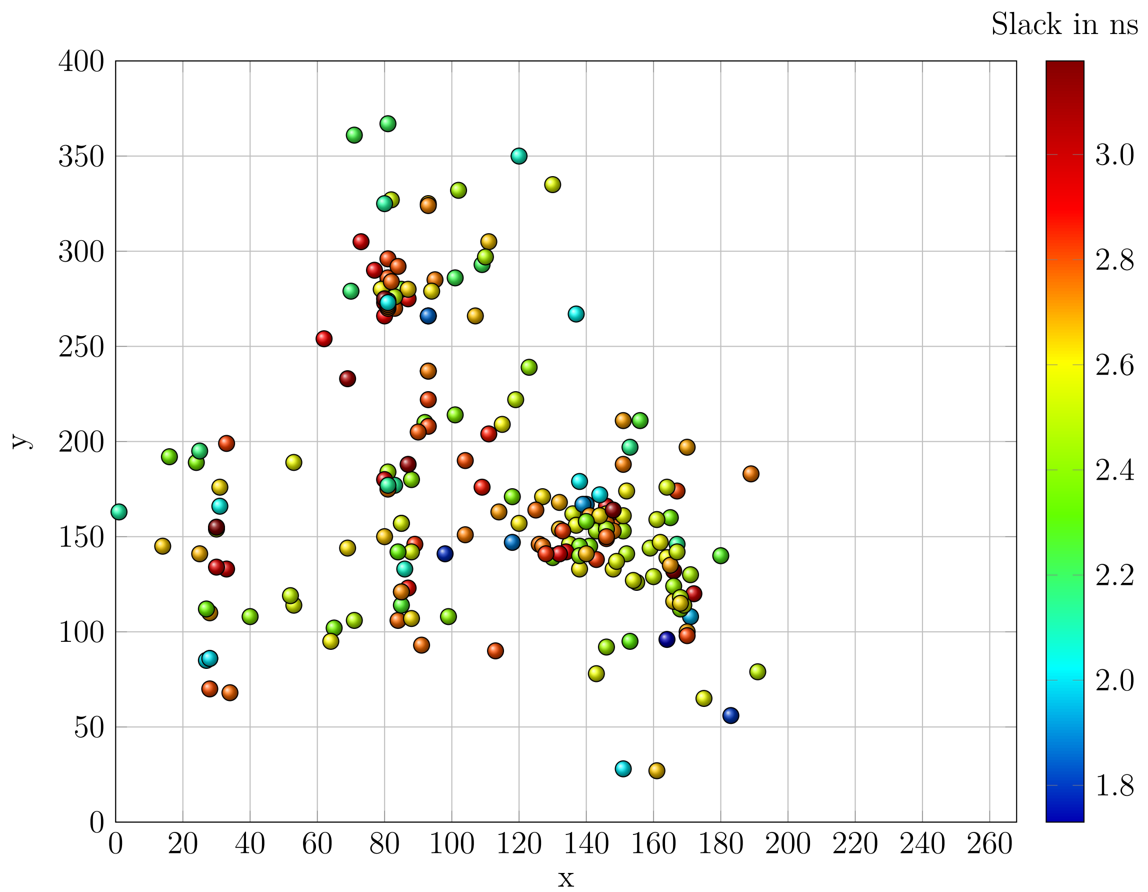


Abbildung A.2.: CGRA mit einem Status Signal, cBox in einem PBlock, aus 5.3.1

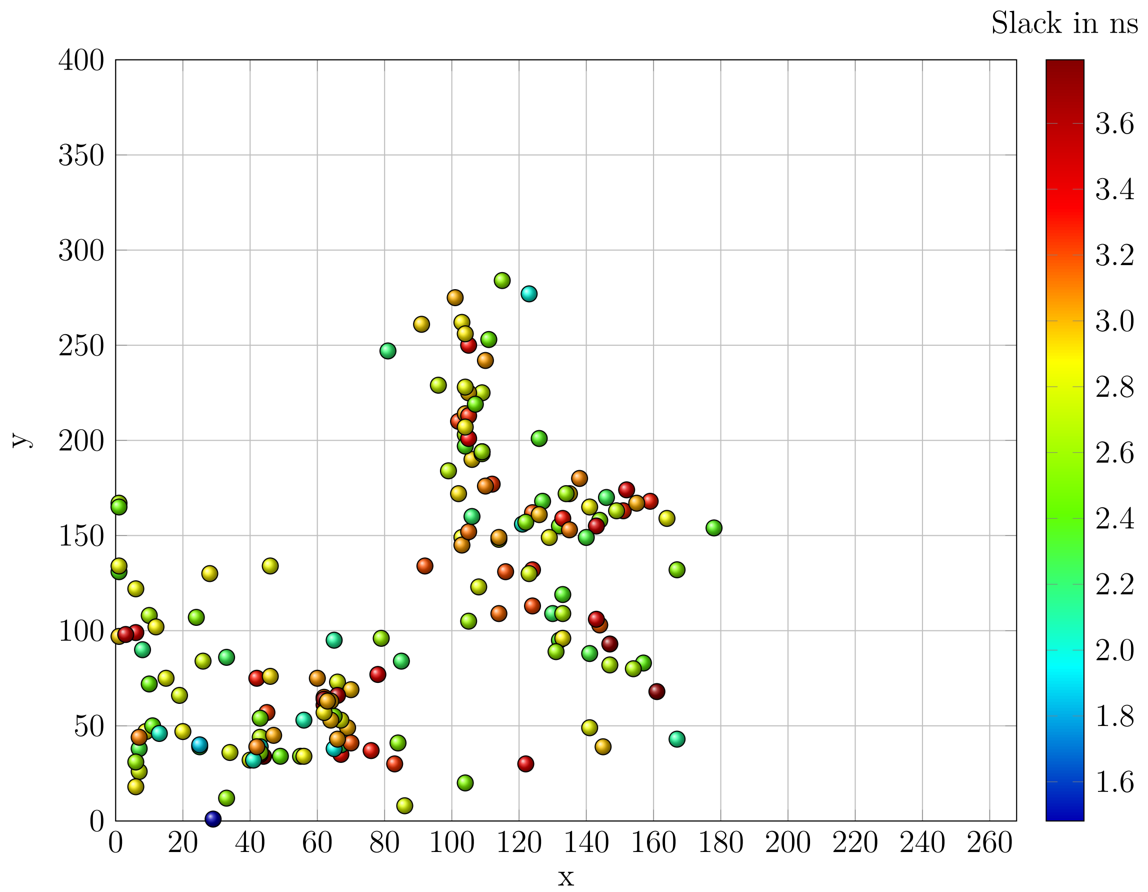


Abbildung A.3.: CGRA mit allen Status Singalen, PE und cBox in einem PBlock, aus 5.3.3

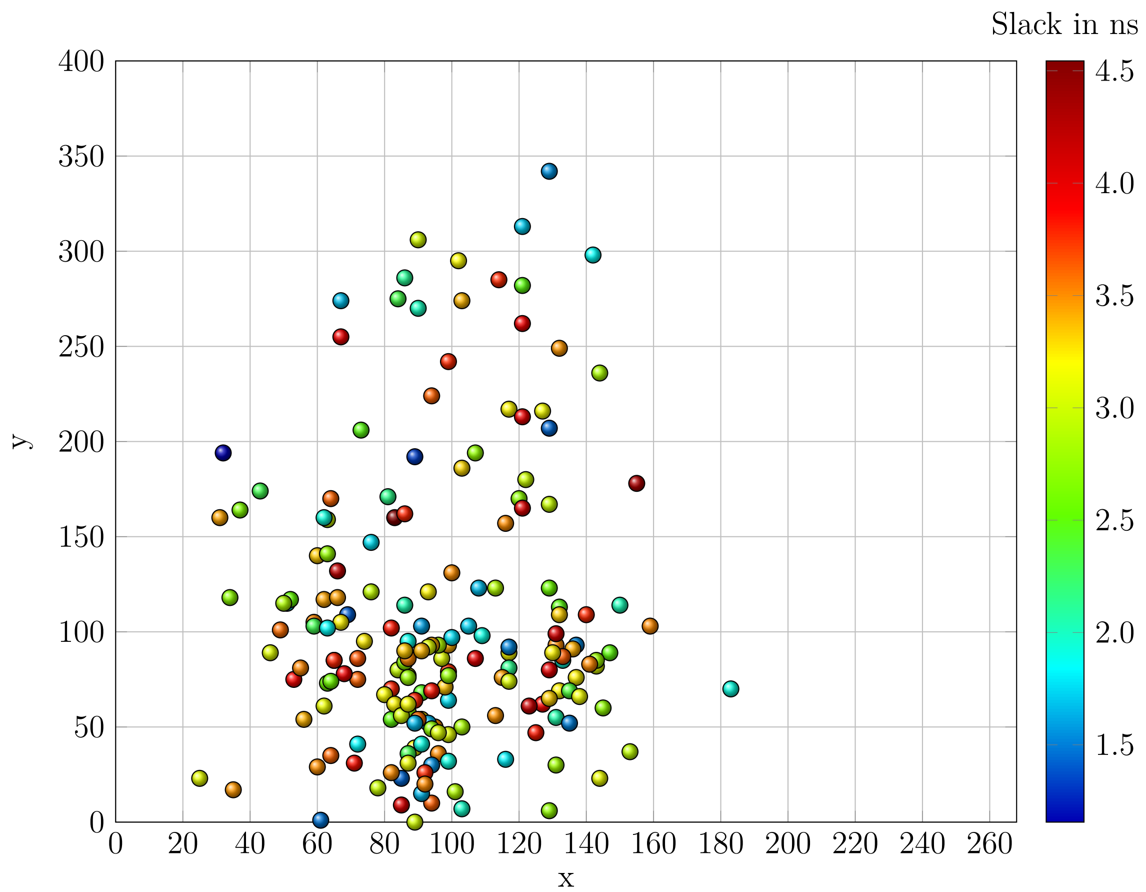


Abbildung A.4.: Homogener CGRA, cBox in einem PBlock, aus 5.3.4

Abbildungsverzeichnis

2.1. Kopie aus [5]: abstrakter Überblick über die CGRA Struktur 4
2.2. Kopie aus [5]: Überblick über ein Processing Element 5

4.1. eines in Vivado synthetisierten Designs auf einem Xilinx XC7Z030 FPGA 12
4.2. Kopie aus [1]: Vereinfachung eines FPGA Layouts 13
4.3. Kopie aus [1]: Das Modul braucht 16 CLBs, ein 36 kBRAM und ein DSP. Es werden PBlöcke von den Startpunkten (3,4) und (16,5) definiert, welche alle benötigten Ressourcen beinhalten. Die gelben Rechtecke geben die implementierte Lösung an. 15
4.4. Stellt einen vereinfachte aus Vivado dargestellten kritischen Pfad auf dem CGRA[5] dar 16
4.5. Kopie aus [3]: Verschiedene Schritte des Downhill Simplex Verfahren für einen dreidimensionalen Suchraum. Das oberste Bild zeigt ein Tetraeder, welches den Simplex vor jedem Schritt aufspannt. Das Simplex nach Ende des ersten Durchlaufes wird durch (a) als Reflexion vom schlechtesten Punkt, (b) als Erweiterung vom Reflexion Punkt, (c) eine Kontraktion vom schlechtesten Punkt oder (d) eine Komprimierung jeder Dimension zum besten Punkt, ersetzt. 18
4.6. Ablauf der Design Space Exploration 21

5.1. Auszug der Suche des Homogenen CGRA aus 5.3.4. Der Slack wird Anhand der Schritte des Algorithmus dargestellt. Die Schritte werden nur vom dritten Thread gezeigt. 35
5.2. Auf den Graphen sind alle PBlock-Koordinaten markiert, die für die Optimierung der verschiedenen CGRAs synthetisiert wurden. Über die Farbskalar wird der erreichte Slack der Koordinate gezeigt. 36

A.1. CGRA mit einem Status Signal, PE und cBox in einem PBlock, aus 5.3.1 I
A.2. CGRA mit einem Status Signal, cBox in einem PBlock, aus 5.3.1 . . II

A.3. CGRA mit allen Status Singalen, PE und cBox in einem PBlock, aus 5.3.3 III
A.4. Homogener CGRA, cBox in einem PBlock, aus 5.3.4 IV

Tabellenverzeichnis

5.1. Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position eines PBlock verändert. In dem PBlock ist die cBox enthalten. 28
5.2. Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE5 enthalten. 29
5.3. Besten Ergebnisse der Teiloptimierung eines CGRAs mit einem Status Signal. Es wurde die Position eines PBlock verändert. In dem PBlock sind die cBox und PE5 enthalten. 29
5.4. Besten Ergebnisse der Teiloptimierung eines CGRAs mit drei Status Signalen. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE1 enthalten. 30
5.5. Besten Ergebnisse der Teiloptimierung eines CGRAs mit drei Status Signalen. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und die CCU, und im Anderen das PE1 enthalten. 31
5.6. Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von einem PBlock verändert. In dem PBlock ist die cBox und das PE0 enthalten. 32
5.7. Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE0 enthalten. 32
5.8. Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und die CCU, und im Anderen das PE0 enthalten.. 33

5.9. Besten Ergebnisse der Teiloptimierung eines CGRAs, in der alle PEs mit einem Status Signal mit der cBox verbunden sind. Es wurde die Position von zwei PBlöcken verändert. In einem PBlock ist die cBox und im Anderen das PE0 enthalten. 33
5.10. Besten Ergebnisse der Teiloptimierung eines homogenen CGRAs. Es wurde die Position von einem PBlock verändert. In dem PBlock ist die cBox enthalten. 34

Abkürzungsverzeichnis

FPGA Field-Programmable Gate Array

DSP Digital Signal Processing

CGRA Coarse-Grained Reconfigurable Array

RAM Random-Access Memory

PE Processing Element

HDL Hardware Desciption Language

CCU Context Control Unit

ASIC Application-Specific Circuit

TCL Tool Command Language

XDC Xilinx Design Constraint

ALU Arithmetic Logic Unit

cBox Control-Box

IC Integrated Circuit

Literatur

[1] L. Cheng und M. D. F. Wong. “Floorplan Design for Multimillion Gate FPGAs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.12 (2006), S. 2795–2805.

[2] Y. Feng und D. P. Mehta. “Heterogeneous floorplanning for FPGAs”. In: *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID’06)*. 2006, 6 pp.–.

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3. Aufl. Site 508-513. New York, NY, USA: Cambridge University Press, 2007.

[4] T. Ruschke, L. J. Jung und C. Hochberger. “A Near Optimal Integrated Solution for Resource Constrained Scheduling, Binding and Routing on CGRAs”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, S. 213–218.

[5] T. Ruschke, L. J. Jung, D. Wolf und C. Hochberger. “Scheduler for Inhomogeneous and Irregular CGRAs with Support for Complex Control Flow”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, S. 198–207.

[6] S. S. Skiena. *The Algorithm Design Manual*. 2008. URL: `http://dx.doi.org/10.1007/978-1-84800-070-4`.

[7] *Vivado Design Suite Tcl Command Reference Guide*. Xilinx. Dez. 2013.

[8] *Vivado Design Suite User Guide*. Xilinx. Okt. 2017.

[9] *Vivado Design Suite User Guide Implemenation*. Xilinx. Apr. 2016.

[10] *Vivado Design Suite User Guide Using Constraints*. Xilinx. März 2013.

[11] E. F. Y. Young, C. C. N. Chu und M. L. Ho. “Placement constraints in floorplan design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.7 (2004), S. 735–745.

[12] *Zynq-7000 SoC Data Sheet: Overview*. v1.11.1. Xilinx. Juli 2018.