INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Performanceanalyse von Stencil-Kernels auf FPGAs

Sergej-Alexander Breiter

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Performanceanalyse von Stencil-Kernels auf FPGAs

Sergej-Alexander Breiter

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Pascal Jungblut
Abgabetermin:	29. Januar 2020

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 29. Januar 2020

. . . . . .
(Unterschrift des Kandidaten)

Abstract

In dieser Arbeit wird die Performance von OpenCL Stencil-Kernels auf Field-programmable Gate Arrays (FPGAs) untersucht. Ausgehend von einer naiven Implementierung werden inkrementell algorithmische und FPGA-spezifische OpenCL Compiler-Optimierungen angewandt und deren Auswirkung auf die Performance analysiert. Zur gezielten Optimierung und Analyse der Performance wird die Hardware-Architektur von FPGAs studiert.

Es wird gezeigt, dass eine optimierte Implementierung, welche FPGA-spezifische Architektureigenschaften ausnutzt, eine Beschleunigung um einen Faktor von über 150 gegenüber einer naiven Implementierung eines Stencil-Algorithmus erreichen kann. Die erwartete Energieeffizienz des optimierten Kernels liegt in der selben Größenordnung, wie die der besten Supercomputer aus der Green500-Liste im November 2019.

In this study the performance of OpenCL stencil kernels on Field-programmable Gate Arrays (FPGAs) is investigated. Starting from a naive implementation, incremental algorithmic and FPGA-specific OpenCL compiler optimizations are applied and their impact on performance is analyzed. The hardware architecture of FPGAs is studied to optimize and analyze the performance in a targeted manner.

It is shown that an optimized implementation, which takes advantage of FPGA-specific architecture properties, can achieve an acceleration by a factor of over 150 compared to a naive implementation of a stencil algorithm. The expected energy efficiency of the optimized kernel is of the same order of magnitude as that of the best supercomputers from the Green500 list in November 2019.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Nachdem es über Jahrzehnte gelungen war, die Größe von Transistoren durch Miniaturisierung zu verkleinern und somit die Transistordichte auf Computerchips zu erhöhen wird, aufgrund der Verlangsamung von Moores Law, in der High-Performance Computing (HPC)-Community nach neuen Möglichkeiten gesucht, Berechnungen schneller und energieeffizienter durchzuführen.

Dadurch rücken neben GPUs, welche heutzutage in Supercomputern eingesetzt werden, um Berechnungen zu beschleunigen, mittlerweile auch exotische Computerkonzepte in den Fokus. Technische Verbesserungen bei Field-programmable Gate Arrays (FPGAs) und deren Potential als energieeffizienter Softwarebeschleuniger machen FPGAs immer attraktiver für den High-Performance-Bereich [VB16]. Die Performance eines FPGA kann mit über 10TFLOP/s die Performance einer CPU bei weitem übertreffen [WHU17].

Die Programmierung von FPGAs unterscheidet sich deutlich von der herkömmlicher Komponenten. Es muss bei FPGAs nicht nur Quellcode in Maschinensprache übersetzt werden, sondern eine Hardwarebeschreibung erstellt werden, welche das logische Verhalten des FPGA definiert und der gewünschten Anwendung entspricht.

Mit OpenCL für FPGAs wurde von [CNK+13] ein High-Level Synthesis (HLS)-Tool eingeführt, welches es dem durchschnittlichen Softwareentwickler erleichtert, die Programmierung von FPGAs auch ohne detaillierte Kenntnisse von traditionellen Hardware Description Languages (HDLs) wie VHDL oder Verilog vorzunehmen. Bei der HLS wird eine Hardwarebeschreibung aus einer C-basierten Anwendung erzeugt. Zusätzlich eignet sich das Programmiermodell von OpenCL für FPGA-Anwendungen. Im OpenCL-Modell ruft ein Host ein Device auf, auf dem ein Teil einer Anwendung ausgeführt wird. Die Kommunikation zwischen Host und Device findet über einen gemeinsamen geteilten Speicherbereich statt.

1.2 Problemstellung

Ziel dieser Arbeit ist die Analyse der Performance einer Anwendung auf einem FPGA. Als Vertreter einer häufig verwendeten Klasse von Anwendungen wird die Performance eines Stencil-Kernels auf einem FPGA untersucht.

Zur Performanceanalyse sollen Metriken definiert und gemessen werden, welche eine systematische Untersuchung erlauben. Um Einflussfaktoren auf die Performance zu bestimmen, werden vor Beginn der Messungen mögliche Faktoren identifiziert und anschließend analysiert.

Kernels werden mit der Open Computing Language (OpenCL) implementiert und aus den Messungen erzielte Ergebnisse werden in geeigneter Form visualisiert und interpretiert.

1.3 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die Grundlagen des Themas behandelt. Ausgehend von einer Einführung in die FPGA-Technologie wird der OpenCL-Standard mit Bezug auf FPGAs behandelt. Anschließend werden die grundlegenden Begriffe zu Stencil Codes eingeführt, so wie deren Funktionsweise und Bedeutung im HPC-Kontext erläutert. Am Ende dieses Kapitels werden aktuelle themenverwandte Arbeiten diskutiert.

Kapitel 3 stellt Optimierungsmethoden vor, welche zur Stencil-Berechnung auf FPGAs anwendbar sind. Dieses Kapitel ist aufgeteilt in zwei Abschnitte, von denen der erste Abschnitt algorithmische Konzepte behandelt. Der zweite Abschnitt geht auf FPGA-spezifische OpenCL Compiler-Optimierungen ein.

Nach der Behandlung der theoretischen und technischen Grundlagen wird in Kapitel 4 die Vorgehensweise erläutert, worauf in Kapitel 5 die zur Evaluation verwendete Hard- und Software und die Implementierung einzelner Kernels behandelt wird und das Ergebnis der Messungen wird in tabellarischer Form dargestellt.

Eine Interpretation und graphische Aufarbeitung der erhaltenen Resultate bildet den Abschluss des Hauptteils.

Zuletzt werden die erhaltenen Ergebnisse zusammengefasst und mögliche weitere und tiefer gehende Untersuchungen vorgeschlagen, welche an diese Arbeit anknüpfen können.

2 Grundlagen

2.1 FPGAs

FPGAs sind integrierte Schaltkreise, deren logisches Design auch nach der Herstellung für verschiedene Algorithmen konfiguriert werden kann. Dadurch unterscheiden sich FPGAs von den fest verschalteten Application-specific Integrated Circuits (ASICs). Sie werden unter anderem bei Embedded Systems und zur Entwicklung von Prototypen verwendet, haben aber eine geringere Flächen- und Energieeffizienz als ASICs.

Im Gegensatz zu CPUs und GPUs, bei denen die Architektur der Arithmetic Logical Units (ALUs) fest ist, können FPGAs anwendungsspezifische ALUs implementieren und können dadurch energieeffizienter als CPUs und GPUs sein. Die durch technische Verbesserungen weiter steigende Performance und Energieeffizienz machen FPGAs auch attraktiv für den Einsatz im HPC-Bereich [WHU17]. Die beiden größten Hersteller von FPGAs sind Xilinx und, seit der Übernahme von Altera im Dezember 2015, Intel.

2.1.1 FPGA-Architektur

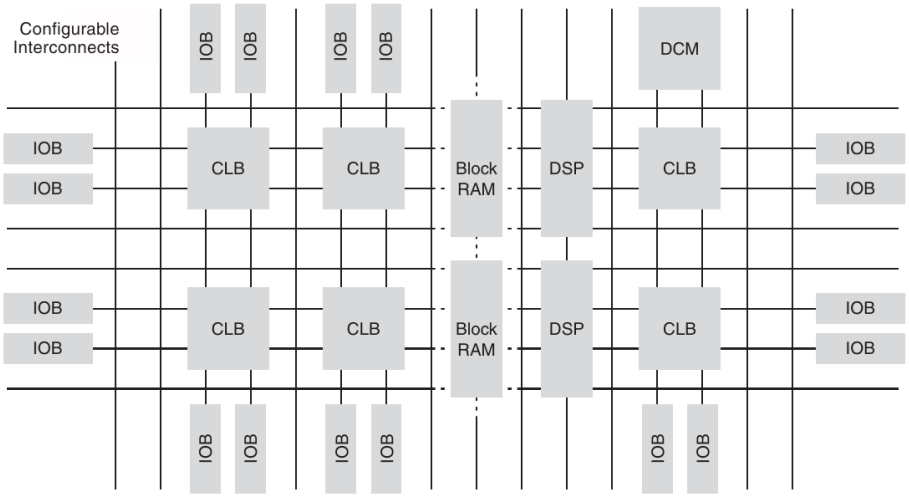


Abbildung 2.1 – High-Level FPGA Blockdiagramm [PS10] Fig. 2

FPGAs bestehen im Wesentlichen aus einer Anordnung von Configurable Logic Blocks (CLBs), Digital Signal Processors (DSPs), Block RAM (BRAM) und Input/Output Blöcken (IOB) und einer konfigurierbaren Schaltmatrix, welche die einzelnen Elemente verbindet (vgl. Abbildung 2.1). CLBs und DSPs können verwendet werden, um arithmetische und logische Operationen wie z.B. Additionen, Multiplikationen und Vergleiche auszuführen. Der Output

einer Operation kann direkt als Input der nächsten Operation genutzt werden, wodurch eine massive Parallelität auf Anweisungs- und Datenebene ermöglicht wird [PS10].

CLB

Ein CLB enthält mehrere Look-up Tables (LUTs) und durch Flip-Flops (FFs) implementierte Speichereinheiten [Xil18d].

LUT

LUTs sind die Grundbausteine eines FPGA und können eine beliebige boolsche Funktion von N boolschen Variablen implementieren. N bezeichnet dabei die Anzahl an Inputs für einen LUT. Im Wesentlichen ist ein LUT eine konfigurierbare Wahrheitstabelle der Größe N, wobei $N = 6$ ein typischer Wert für aktuelle FPGAs ist. Das Ergebnis der logischen Operationen wird in Registern in Form von FFs gespeichert [Xil18a]. Die Wahrheitstabellen der LUTs werden vor der Ausführung eines FPGA festgelegt und definieren die Logik der Berechnung auf dem FPGA.

LUTs können zudem als kleiner 64-Bit Speicher konfiguriert werden, was der schnellste Speichertyp auf einem FPGA ist, da er auf einem beliebigen Teil des FPGA instanziiert werden kann. Außerdem können durch LUTs adressierbare Schieberegister implementiert werden (vgl. Abschnitt 3.1.1) [Xil18c].

DSP

Ein DSP ist eine eingebettete ALU, die Multiplikationen, Additionen und Aufsummierungen vornehmen kann. Diese häufig auftretenden Operationen können somit auf effizientere dedizierte ALUs ausgelagert werden und müssen nicht durch LUTs implementiert werden. Ein typischer DSP-Block, welcher in modernen FPGAs eingesetzt wird, ist der DSP48 Block. Dieser kann Funktionen der Form $P = B \cdot (A + D) + C$ oder $P \mathrel{+}= B \cdot (A + D)$ implementieren [Xil18c].

BRAM

BRAM ist ein Dual-Port-SRAM-Modul um einen Speicher auf einem FPGA für größere Datenmengen bereitzustellen. Die Dual-Port-Eigenschaft von BRAM erlaubt zwei parallele Zugriffe auf verschiedene Bereiche des Speichers während des selben Takts. Ein BRAM hat typischerweise eine Kapazität von 18 kbit oder 36 kbit [Xil18c].

2.1.2 FPGA-Programmierung

Traditionell werden FPGAs durch HDLs wie VHDL oder Verilog programmiert. Dieser Prozess ist zeitaufwändig und der Entwickler muss die Architektur der verwendeten Hardware genau kennen [EWH17]. Um Softwareentwicklern den Zugang zur FPGA-Programmierung zu erleichtern, wurde von [CNK+13] die HLS mit OpenCL für FPGAs eingeführt. Bei der HLS wird aus einer C-basierten Anwendung eine Hardwarebeschreibung der Anwendung in einer HDL generiert.

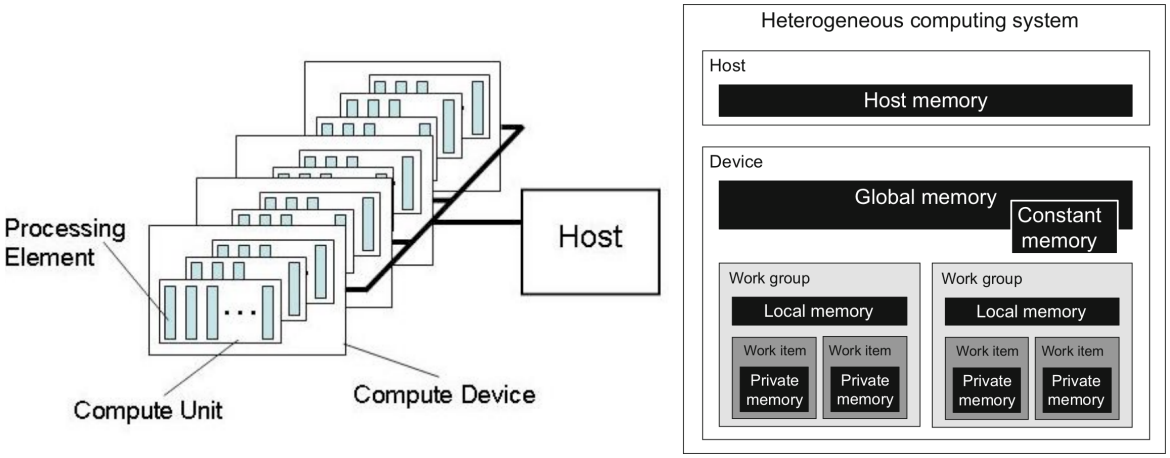
Im Gegensatz zur herstellerspezifischen HLS kann der selbe OpenCL Code für unterschiedliche FPGAs kompiliert und ausgeführt werden. Aus der Hardwarebeschreibung einer Anwendung kann der *Bitstream* für eine spezifische FPGA-Plattform generiert werden. Der Bitstream kodiert die Konfiguration und Platzierung der Komponenten des FPGA und definiert dadurch die Logik einer FPGA-Anwendung [WHU17].

2.1.3 HPC mit FPGAs

Derzeit werden im HPC-Bereich neben CPUs auch GPUs als Beschleuniger eingesetzt. FPGAs können durch ihre Rekonfigurierbarkeit für beliebige Anwendungen verwendet werden und haben das Potential als energieeffizienter und performanter Beschleuniger im HPC-Bereich eingesetzt zu werden [WHU17]. Hochleistungsrechenzentren können besonders von energieeffizienten Beschleunigern profitieren, da die Verringerung der Betriebskosten mit der Größe des Systems skaliert.

2.2 OpenCL

OpenCL ist ein offener, lizenzkostenfreier Standard für die plattformunabhängige Programmierung paralleler Software und Hardwarebeschleuniger. Das Programmiermodell von OpenCL ermöglicht es, Teile einer Anwendung auf dafür performantere Hardware auszulagern. Das Besondere an OpenCL ist, dass der selbe OpenCL Code für verschiedenartige Prozessoren kompiliert und ausgeführt werden kann. Die wichtigsten Konzepte von OpenCL werden im Folgenden erklärt und die zentralen Begriffe eingeführt.



(a) OpenCL Platform Model [Khr19] Fig. 3.1

(b) OpenCL Memory Model [WHU17] Fig. 1.4

Abbildung 2.2

2.2.1 Platform Model

Die OpenCL Platform besteht aus einem Host und einer Sammlung an Geräten (OpenCL Devices), die von der OpenCL-Umgebung verwaltet werden (vgl. Abbildung 2.2a). Ein Host ist mit einem oder mehreren OpenCL Devices verbunden und eine Anwendung wird auf dem

Host ausgeführt. Die OpenCL-Umgebung erlaubt es einer Anwendung Hardwareressourcen zu teilen und Kernels (vgl. Abschnitt 2.2.2) auf OpenCL Devices innerhalb der OpenCL Platform auszuführen.

OpenCL Devices
Eine Menge aus einer oder mehreren vom Host verwalteten Compute Units (CUs).

Compute Unit
Eine CU erlaubt die Ausführung einer Work-Group (vgl. Abschnitt 2.2.2) und besteht aus einem oder mehreren Processing Elements (PEs) und Local Memory.

Processing Element
Ein PE ist ein virtueller skalarer Prozessor. Ein Work-Item (vgl. Abschnitt 2.2.2) kann auf einem oder mehreren PEs ausgeführt werden.

[Khr19]

2.2.2 Execution Model

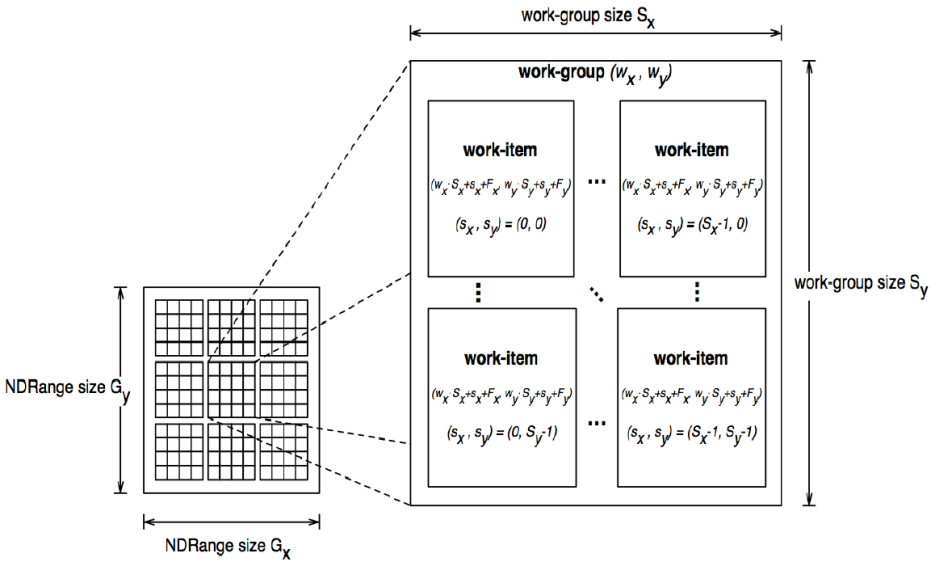


Abbildung 2.3 – OpenCL Execution Model: Beispiel eines NDRange mit Work-Items innerhalb ihrer zugehörigen Work-Group und den Work-Item- und Work-Group IDs [Khr19] Fig. 3.2

Eine OpenCL-basierte Anwendung kann in zwei getrennten Teilen betrachtet werden. Ein Teil ist der *Host Code*, welcher auf der Host CPU ausgeführt wird, der andere Teil ist der *Device Code* oder *Kernel Code*, welcher auf einem OpenCL Device ausgeführt wird.

Host Code

Der Host Code ist für den Speichertransfer, die Speicherverwaltung und die Einreihung von Kernels in eine Ausführungswarteschlage zur Ausführung auf einem OpenCL Device verantwortlich [Khr19].

Kernel

Ein OpenCL Kernel ist eine Funktion, die auf einem OpenCL Device ausgeführt wird. Aufgerufen werden kann ein Kernel vom Host oder einem anderen Kernel. Es gibt im OpenCL-Modell zwei grundlegende Arten von Kernels: *NDRange Kernels* und *Single-Task Kernels*.

NDRange Kernel

Die Idee von NDRange Kernels ist, ein N-dimensionales Problem in mehrere Teilprobleme (Work-Groups) zu zerlegen, welche wiederum aus kleineren Teilproblemen (Work-Items) bestehen.

Work-Groups können auf verschiedene Hardwareressourcen verteilt werden, wobei auf diesen innerhalb einer Work-Group mehrere Work-Items auf der gleichen Hardware parallel ausgeführt werden können (vgl. Abbildung 2.3). Die Work-Items eines NDRange Kernel werden implizit als Schleife über den NDRange ausgeführt [Khr19].

NDRange

Der NDRange ist ein N-dimensionaler Indexraum ($N \in \{1, 2, 3\}$) und wird auf mehrere Work-Groups aufgeteilt, welche jeweils einen Teil des Indexraums abdecken. Ein NDRange wird durch drei Integer Arrays der Länge N definiert:

Global Size
: Die Größe G_i des gesamten Indexraums in jeder Dimension.

Offset Index
: Zeigt den initialen Wert F_i der Indizes in jeder Dimension an.

Local Size
: Die Größe S_i der Work-Groups in jeder Dimension.

Work-Groups werden ähnlich zugeordet (vgl. Abbildung 2.3) [Khr19].

Work-Groups

Eine Work-Group ist eine Menge an verwandten Work-Items, die auf einer einzelnen CU ausgeführt werden. Die Work-Items einer Work-Group führen den selben Kernel aus und haben einen gemeinsamen geteilten Speicherbereich - den Local Memory (vgl. Abschnitt 2.2.3).

Die Position einer Work-Group innerhalb des NDRange wird durch ein N-dimensionales Tupel aus N Indizes w_i bestimmt und die Anzahl der Work-Groups wird aus dem NDRange und der Work-Group Größe abgeleitet [Khr19].

Work-Items

Ein Work-Item ist ein Teil einer Work-Group und wird parallel von einem oder mehreren PEs ausgeführt. Jedes Work-Item wird einer Work-Group zugeordnet und besitzt eine local ID, welche die Position des Work-Item innerhalb der Work-Group repräsentiert.

Ein Work-Item unterscheidet sich von anderen durch seine global ID, oder innerhalb einer Work-Group durch seine local ID. Work-Items müssen global unabhängig voneinander sein, können aber innerhalb ihrer Work-Group durch Barrieren synchronisiert werden [Khr19].

Single-Task Kernels

Ein Single-Task Kernel ist als ein sequenzielles Programm implementiert und äquivalent zu einem NDRange Kernel mit nur einem Work-Item (Work-Group mit $S_x = 1, S_y = 1, S_z = 1$). Single-Task Kernels werden auch als Single-Work-Item Kernels bezeichnet.

2.2.3 Memory Model

Das OpenCL Memory Model unterscheidet zwischen verschiedenen Speicherbereichen bzw. Adressräumen (vgl. Abbildung 2.2b). Jeder Speicherbereich hat dabei verschiedene Allozierungs- und Zugriffseigenschaften. Im Folgenden wird auf diese Speicherbereiche und ihre Zuordnung im FPGA-Kontext eingegangen.

Host Memory

Host Memory ist nur vom Host zugänglich. Von Kernels benötigte Daten müssen vom Host Memory in den Global Memory transferiert werden.

Global Memory

Global Memory ist der Bereich des Speichers, welcher sowohl vom OpenCL Host, als auch vom OpenCL Device zugänglich ist und wird verwendet, um Daten zwischen Host und Device auszutauschen.

Ein Speicherobjekt im globalen Adressraum wird durch den Host alloziert und der Kernel hat Lese- und Schreibrechte. Während der Ausführung eines Kernels verliert der Host seine Zugriffsrechte.

Constant Memory

Constant Memory liegt im globalen Adressraum, jedoch hat nur der OpenCL Host Schreibrechte. Ein typischer Anwendungsfall ist der Transfer von konstanten Daten vom Host zum Kernel, welche vom Kernel für die Berechnung benötigt werden.

Local Memory

Local Memory ist der Speicherbereich, welcher einer einzelnen CU zugeordnet ist. Der Host hat weder Zugriff auf, noch Kontrolle über den Local Memory. Dieser Speicherbereich erlaubt Lese- und Schreiboperationen von allen PEs innerhalb einer CU.

Alle Work-Items innerhalb einer Work-Group teilen sich den selben lokalen Adressraum. Local Memory wird üblicherweise verwendet, um Daten zu speichern, welche zwischen mehreren Work-Items geteilt werden müssen. Auf den Local Memory kann nur von Work-Items innerhalb der selben Work-Group zugegriffen werden.

Falls es Datenabhängigkeiten auf dem Local Memory zwischen den Work-Items gibt, müssen diese mittels einer Barriere synchronisiert werden. Erst wenn alle Work-Items diesen Synchronisationspunkt erreicht haben, können die Work-Items mit der Ausführung fortfahren.

Private Memory

Wie beim Local Memory hat der Host keinen Zugang zum Private Memory. Private Memory ist allerdings nur innerhalb eines individuellen Work-Item zugänglich.

[Khr19]

OpenCL Memory Model auf FPGAs

Das OpenCL Memory Model definiert das Verhalten und die Hierarchie des Speichers, welcher von OpenCL-Andwendungen verwendet werden kann. Diese hierarchische Repräsentation des Speichers ist allen OpenCL-Implementierungen gemein, aber es ist dem Hersteller überlassen, wie das OpenCL Memory Model auf eine spezifische Hardware abgebildet wird.

Auf Xilinx-FPGAs gilt für die physikalische Abbildung des OpenCL Memory Model, welche von *SDAccel*[1] verwendet wird:

Global Memory, Constant Memory
: Global Memory und Constant Memory ist physikalisch mit dem FPGA verbunden und wird üblicherweise durch DRAM implementiert.

Private Memory, Local Memory
: Private Memory und Local Memory liegt innerhalb des FPGA und wird durch Register oder BRAM implementiert.

[Xil16a]

2.3 Stencil Codes

Stencils sind eine der wichtigsten Berechnungsmuster im HPC-Bereich. Sie werden zum Beispiel bei der Lösung diskretisierter Differenzialgleichungen mithilfe der Finite-Differenzen-Methode, bei physikalischen Simulationen, bei der Bildverarbeitung und bei Convolutional Neural Networks (CNNs) genutzt [JZ16].

2.3.1 Stencil Terminologie

Bei der Stencilberechnung wird ein mehrdimensionales Gitter iterativ traversiert, wobei für jede Zelle a des Gitters ein aktualisierter Wert aus der gewichteten Summe der Werte seiner Nachbarn und der Zelle selbst berechnet wird (vgl. Formel 2.1).

$$a(\vec{x},\, t+1) = \sum_i c_i \cdot a(\vec{x} + \Delta\vec{x}_i,\, t) \tag{2.1}$$

[1] SDAccel ist die OpenCL-Umgebung von Xilinx.

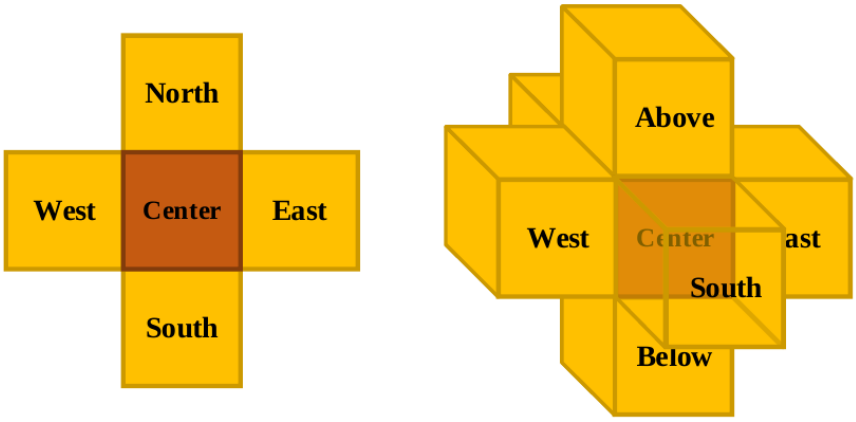


Abbildung 2.4 – Stencilmuster **links**: 2D Stencil, **rechts**: 3D Stencil jeweils erster Ordnung (Radius 1) [Zoh18] Fig. 5-1

Das Muster des Stencils gibt an, welche Nachbarn bei der Berechnung involviert sind. Der maximale Abstand zwischen den Nachbarn und der Zelle wird als Radius bzw. Ordnung des Stencils bezeichnet (vgl. Abbildung 2.4).

Die Berechnung erfolgt typischerweise iterativ über mehrere Zeitschritte, wobei das Gitter zum Zeitpunkt t zur Berechnung des nächsten Zeitpunktes $t+1$ verwendet wird [WHU17].

2.3.2 2D 5-Punkt Stencil

Im zweidimensionalen Beispiel aus Abbildung 2.4 gilt für die Berechnungsvorschrift 2.1:

$$\begin{aligned} a(\vec{x},\, t+1) = c_0 \cdot a(\vec{x},\, t) + c_1 \cdot a(\vec{x} + \Delta\vec{x}_1,\, t) + c_2 \cdot a(\vec{x} + \Delta\vec{x}_2,\, t) \\ + c_3 \cdot a(\vec{x} + \Delta\vec{x}_3,\, t) + c_4 \cdot a(\vec{x} + \Delta\vec{x}_4,\, t) \end{aligned} \tag{2.2}$$

mit:

$$\Delta\vec{x}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \Delta\vec{x}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \Delta\vec{x}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \Delta\vec{x}_3 = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \Delta\vec{x}_4 = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{2.3}$$

Dieses Stencilmuster wird auch als 2D 5-Punkt Stencil bezeichnet [WHU17].

In dieser Arbeit wird der 2D 5-Punkt Stencil **einer** Iteration auf beliebigen Inputgrößen untersucht mit:

$$Radius = 1 \qquad , \qquad c_i = \frac{1}{N} = 0,2 \qquad \forall i \in \{0, \ldots, 4\} \tag{2.4}$$

2D Jacobi Stencil aus der Polybench Benchmark Suite [P+12]

2.3.3 Stencil Codes auf FPGAs

Es existiert zur Stencilberechnung mit OpenCL auf FPGAs eine Vielzahl an verwandten Arbeiten. Sie unterscheiden sich in der verwendeten Hardware, dem untersuchten Stencilmuster und der Implementierung des Designs.

[JZ16] ist eine der ersten Arbeiten zu OpenCL Stencil-Kernels auf FPGAs. Es werden Single-Task- und NDRange Stencil-Kernels unterschiedlicher Dimensionalität und Muster auf Intel FPGAs untersucht. [JZ16] ist auch eine der ersten Arbeiten in der *Schieberegister* (vgl. Abschnitt 3.1.1) zur Stencilberechnung eingesetzt werden, wodurch eine deutliche Performancesteigerung gegenüber den Implementierungsbeispielen des Herstellers ohne Schieberegister gemessen wurde. Die Implementierung beschränkt sich auf eine feste Inputgröße.

[Zoh18] ist eine umfassende Doktorarbeit zum Thema, deren Inhalt teilweise auch in [ZPM18] und [ZMSM16] veröffentlicht wurde. Es werden 2D- und 3D Stencil-Kernels mehrerer Iterationen beliebiger Inputgrößen auf Intel FPGAs analysiert. Datenwiederverwendung wird durch *Spatial Blocking* (vgl. Abschnitt 3.1.1) mit Schieberegistern erreicht und Iterationen werden mit *Temporal Blocking* (vgl. Abschnitt 3.1.2) parallelisiert, wodurch auf Intels Arria 10 (Peak Performance 1.450 GFLOP/s) eine Performance von 745 GFLOP/s mit einem 2D Stencil 1. Ordnung ($9FLOP/Stencilupdate$) erreicht wurde.

Um redundante Berechnungen bei iterativen Stencil-Algorithmen mit Temporal Blocking zu reduzieren, wird in [WL17] die Verwendung von OpenCL-Pipes untersucht. Die Verwendung von Pipes ermöglicht die Kommunikation zwischen Kernels, ohne Daten in einen externen Speicher zu schreiben. Die Implementierung wurde auf dem Virtex-7-FPGA von Xilinx evaluiert und ergab eine Performancesteigerung zur verwendeten Baseline – Details zur erreichten Performance wurden nicht veröffentlicht.

Für GPUs optimierte OpenCL Stencil-Kernels wurden in [VHKF16] auf FPGAs gemessen und durch Compiler-Optimierungen auf FPGAs adaptiert. Die Performance der GPU-Kernels konnte durch FPGA-spezifische Optimierungen deutlich gesteigert werden und es wird ein Überblick über verwendbare Optimierungsmethoden gegeben, von denen einige im anschließenden Kapitel behandelt werden.

3 Optimierung von Stencil-Kernels

3.1 Algorithmische Optimierung des Speicherzugriffs

Die Performance einer Memory-Bound-Anwendung ist durch die verfügbare Datenübertragungsrate[1] zu einem entfernten Speicher limitiert. Bei Stencil-Kernels kann die Lokalität der Berechnung ausgenutzt werden, um eine signifikante Reduktion der benötigten Bandbreite zum Global Memory zu erreichen [Zoh18].
Im Folgenden werden algorithmische Optimierungen vorgestellt, die zur Verbesserung von Speicherzugriffen führen können.

3.1.1 Spatial Blocking

Verläuft der Berechnungspfad einer Anwendung entlang wiederverwendbarer bereits verfügbarer Daten, können diese innerhalb eines lokalen Speichers behalten werden, um ein redundantes Laden der Daten zu vermeiden. Da physikalisch naher Speicher nicht unbegrenzt verfügbar ist, wird ein Problem in kleinere Probleme aufgeteilt, um wiederverwendbare Daten behalten zu können.

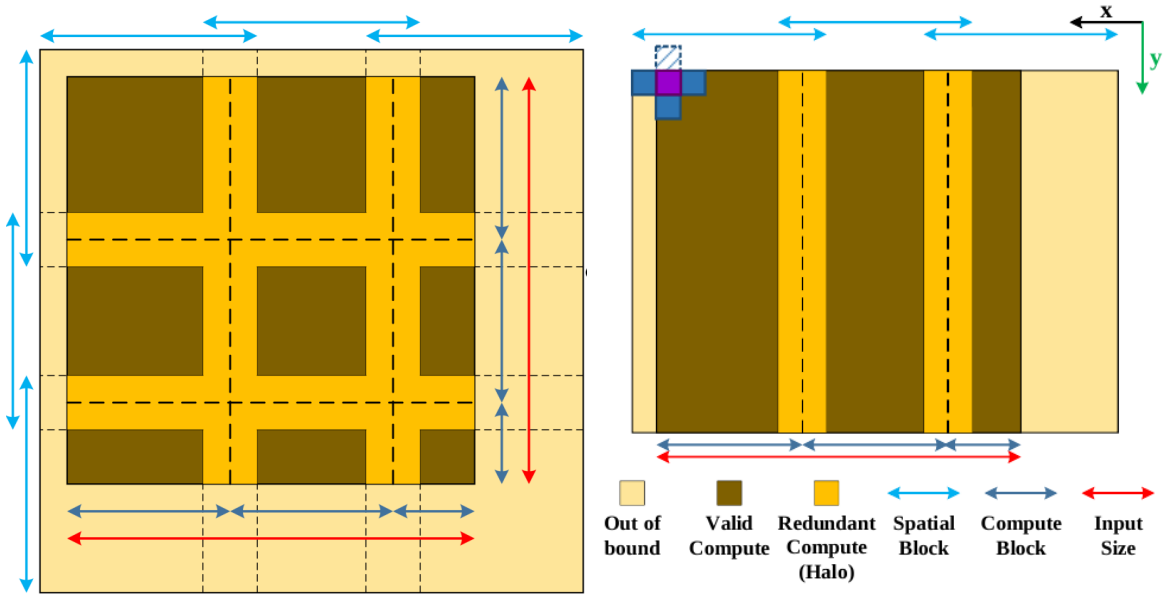
Ein zu berechnendes Gitter wird beim Spatial Blocking in Blöcke unterteilt und einzelne Blöcke des Gitters werden parallel oder sequenziell abgearbeitet. Bei Stencilberechnungen mit Spatial Blocking erfolgen, in Abhängigkeit des Stencilradius, redundante Speicherzugriffe an den Rändern jedes Blocks, da diese Daten für mehrere Blöcke benötigt und geladen werden (vgl. Abbildung 3.1) [Zoh18]. Die Performance einer Memory-Bound-Anwendung kann durch Spatial Blocking profitieren indem die Anzahl an redundanten Speicherzugriffen reduziert wird.

Auf CPUs wird Spatial Blocking üblicherweise durch Loop Tiling implementiert, um die Anzahl an Cache-Misses zu reduzieren. Die Größe eines Blocks wird dabei an die Speicherkapazität des CPU Cache angepasst und sollte klein genug sein, um wiederverwendbare Daten behalten zu können. Diese Bedingung wird bei Stencil-Codes Layer-Condition genannt [STHW14].

Spatial Blocking auf FPGAs mit OpenCL

Auf FPGAs wird der Cache direkt auf dem FPGA durch BRAM oder CLBs implementiert. Der Anwender muss den Cache zur Compile-Zeit festlegen und die notwendige Kapazität wird durch die Blockgröße bestimmt. Die maximale Kapazität, und damit die Größe eines Blocks, ist durch die Hardwareressourcen des FPGA begrenzt. Die Größe des Input ist für Stencil-Kernels mit Datenwiederverwendung ohne Spatial Blocking limitiert [ZPM18].

[1] Im Folgenden auch Bandbreite.



(a) Spatial Blocking in x- und y-Richtung aus [ZPM18] Fig. 4

(b) Spatial Blocking in x-Richtung. y-Richtung wird gestreamed aus [Zoh18] Fig. 5-3

Abbildung 3.1 – 2D Spatial Blocking

Bei **NDRange Kernels** muss sich der Cache im Local Memory befinden, damit alle Work-Items Zugriff auf die Daten erhalten (vgl. Abschnitt 2.2.3). Außerdem muss der gesamte Block vor der Berechnung geladen werden, um eine notwendige Synchronisierung einzelner Work-Items während der Berechnung zu vermeiden.
Mit **Single-Task Kernels** besteht die Möglichkeit einen zyklischen Puffer zu verwenden, welcher es erlaubt die Berechnung zu beginnen, ohne vorher den gesamten Block zu laden.

Der zyklische Puffer kann auf FPGAs als Schieberegister (vgl. Abschnitt 3.1.1) realisiert werden, was einer der FPGA-spezifischen Vorteile ist. Zusammen mit der Notwendigkeit von Barrieren zur Synchronisation bei NDRange Kernels ist die Verwendung eines zyklischen Puffers der Hauptgrund, warum Stencil-Kernels als Single-Task Kernels effektiver als NDRange Kernels auf FPGAs implementiert werden können [ZPM18].

Bei einem zyklischen Puffer werden nur diejenigen Zellen gespeichert, welche für die Berechnung weiterhin benötigt werden, wodurch die benötigte Speichergröße unabhängig von einer Dimension des Input ist und die Berechnung in dieser Dimension gestreamed werden kann (vgl. Abbildung 3.1b).

Schieberegister

Ein Schieberegister ist eine Reihe miteinander verbundener Register und kann als effektiver Speichertyp auf FPGAs eingesetzt werden, um Datenwiederverwendung entlang eines Berechnungspfades zu ermöglichen [Xil18a].

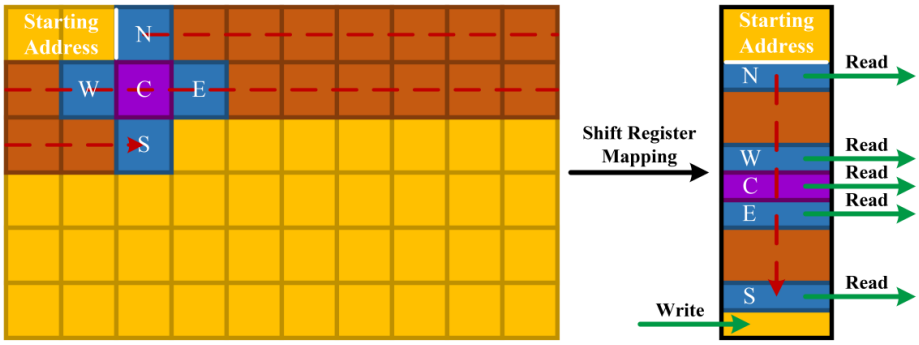
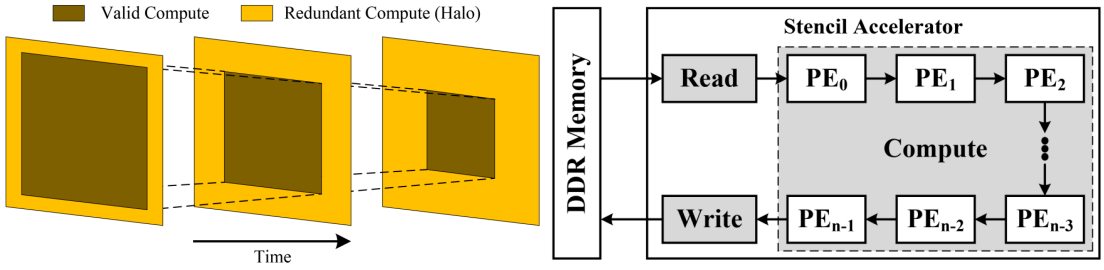


Abbildung 3.2 – Abbildung der zur Datenwiederverwendung weiterhin benötigten Zellen (braun) auf den Speicherinhalt eines Schieberegisters bei der 5-Punkt 2D Stencil Berechnung, aus [ZPM18] Fig. 3



(a) Abnehmender gültiger Bereich bei Temporal Blocking [ZPM18] Fig. 5

(b) Pipeline mehrerer PEs bei Temporal Blocking [ZPM18] Fig. 2

Abbildung 3.3 – Temporal Blocking

Abbildung 3.2 zeigt die Stencilberechnung mithilfe von Schieberegistern. Die Berechnung startet von oben links und bewegt sich vorwärts in x-Richtung. Wenn der Rand erreicht ist, geht die Berechnung am Anfang der nächsten Zeile weiter. Dabei wird jedes Mal der nächste geladene Datenpunkt am Ende des Schieberegisters eingefügt und der Inhalt des Schieberegisters in Richtung Anfang des Schieberegisters verschoben, wodurch sich die zur Berechnung benötigten Daten bei jeder Iteration an der selben Stelle im Schieberegister befinden, was zusätzlich eine statische Adressierung des Puffers erlaubt.

Größere Schieberegister können auf FPGAs durch BRAM, kleinere Schieberegister durch CLBs implementiert werden [VHKF16].

3.1.2 Temporal Blocking

Selbst mit Spatial Blocking werden viele Stencilmuster Memory Bound sein. Stencilberechnungen erfolgen üblicherweise iterativ über mehrere Zeitschritte. Die benötigte Bandbreite kann dann durch Temporal Blocking weiter reduziert werden, indem mehrere Iterationen eines Blocks berechnet werden, bevor das Ergebnis in einen externen Speicher geschrieben wird [MM15].

Außer bei der letzten Iteration müssen vorher entsprechende redundante Berechnungen er-

folgen, um das gültige Gesamtergebnis der letzten Iteration zu erhalten, denn bei jeder Iteration entsteht ein zunehmender ungültiger Bereich, da an den Rändern eines Blocks Daten benötigt werden, welche nicht verfügbar sind (vgl. Abbildung 3.3a) [ZPM18].

Auf FPGAs können mehrere Iterationen parallel berechnet werden. Anstatt in einem PE mehrere Iterationen sequenziell zu berechnen, kann eine Pipeline mehrerer PEs implementiert werden, welche jeweils eine der aufeinanderfolgenden Iterationen berechnen. Der Output eines PE dient als Input des nächsten PE (vgl. Abbildung 3.3b) [ZPM18].

Diese Architektur kann mit Schieberegistern implementiert werden, da mit einem zyklischen Puffer nicht alle Elemente der vorherigen Iteration benötigt werden, bevor die Berechnung beginnen kann. Die Berechnung der Iteration bei $t + 1$ beginnt verzögert zur Berechnung bei t, da erst die benötigten Daten der vorherigen Iteration im Schieberegister des nächsten PE verfügbar seien müssen [Zoh18].

3.1.3 Burst Memory Transfer (BMT)

Bei BMTs werden zusammenhängende Speicherblöcke aufsteigender Adressreihenfolge (row-major Order) vom Global Memory in den FIFO-Speicher des FPGA geladen oder vom FIFO in den Global Memory geschrieben, wodurch der Kernel Daten lesen kann, bevor die Daten zur Berechnung benötigt werden [Xil16a].

Daten im BMT-Modus zu transferieren versteckt die Verzögerung von Speicherzugriffen und verbessert die Bandbreitennutzung und Effizienz des Memory Controller. Die Anzahl an Speicherzugriffen wird durch Zusammenführung verringert, wodurch zusätzlich der auftretende Overhead von DDR-DRAM beim Umschalten zwischen Lesen und Schreiben reduziert wird [Xil19b] [Dre07].

BMTs können durch die Verwendung der OpenCL-Funktion *async_work_group_copy* oder durch Speicherzugriffe in einer Schleife synthetisiert werden. Damit der HLS-Compiler von Xilinx BMTs in einer Schleife synthetisiert, müssen folgende Bedingungen erfüllt sein [Xil16b]:

1. Die Schleife muss eine Loop Pipeline sein (vgl. Abschnitt 3.2.1).
2. Speicherzugriffe müssen kontinuierlich und in aufsteigender Adressreihenfolge erfolgen.
3. Speicherzugriffe dürfen nicht innerhalb einer bedingten Anweisung erfolgen.
4. Verschachtelte Schleifen dürfen nicht abgeflacht[2] sein.

Falls dadurch BMTs synthetisiert werden, kann es performanter sein, mehr Daten als nötig zu laden oder zu speichern [Xil18a].

Das Zusammenführen von Speicherzugriffen kann auch durch Loop Unrolling (vgl. Abschnitt 3.2.3) oder Vektorisierung (vgl. Abschnitt 3.2.4) erreicht werden [WHU17].

[2] abgeflachte Schleife: for($y < rows$) { for($x < cols$) } $\Rightarrow$ for($i < rows * cols$)

3.2 FPGA-spezifische OpenCL Compiler-Optimierung

Bei OpenCL können durch Compiler-Direktiven Hardware-Optimierungen implementiert werden, von denen einige herstellerspezifisch sind. Da in dieser Arbeit ein FPGA der Firma Xilinx untersucht wird, werden im Folgenden auch spezifische Optimierungen des OpenCL-Compilers *sdscc* von Xilinx vorgestellt. Das Prinzip der jeweiligen Optimierung ist jedoch allgemein.

3.2.1 Pipelining

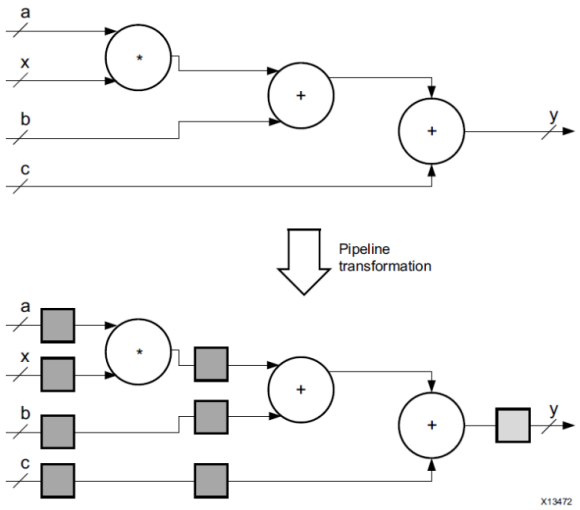


Abbildung 3.4 – Transformation eines Befehls in eine Pipeline.
Die grauen Boxen repräsentieren Register [Xil16a] Fig. 2-7

Beim Pipelining wird ein Befehl durch Zerlegung in mehrere unabhängige Instruktionen in eine Pipeline transformiert. Die Instruktionen einer Pipeline werden auch Pipeline-Stages genannt und sind durch Register getrennt, welche das Ergebnis der Berechnung einer Pipeline-Stage speichern und als Input der nächsten Pipeline-Stage dienen (vgl. Abbildung 3.4).

Alle Instruktionen der Pipeline können parallel innerhalb eines Taktzyklus ausgeführt werden und es können sich mehrere Befehle gleichzeitig in der Pipeline befinden, was Parallelität auf Instruktionsebene erreicht. Die Anzahl der Pipeline-Stages wird als Tiefe einer Pipeline bezeichnet und bestimmt den Faktor der möglichen Beschleunigung durch Pipelining [Xil16a].

Loop Pipelining

Beim Loop Pipelining werden die Befehle für die Ausführung einer Iteration einer Schleife in eine Pipeline transformiert. Die konfigurierbare Architektur von FPGAs erlaubt es, die benötigte Hardware einer Iteration einer beliebigen Schleife zu instanziieren. Pipeline-Stages können auf FPGAs durch LUTs, Register durch FFs implementiert werden [Xil16a]. Die Berechnung mehrerer Iterationen der Schleife kann sich gleichzeitig in der Pipeline befinden, was eine Beschleunigung der Schleife erreicht.

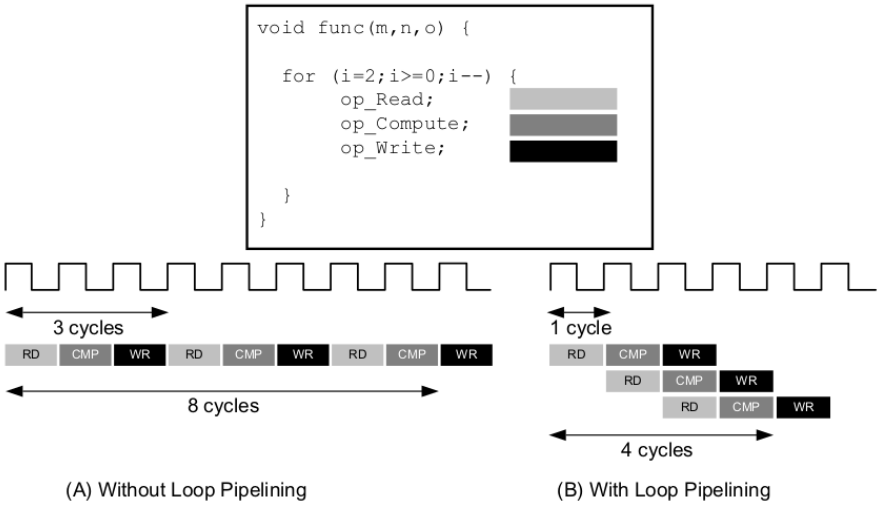


Abbildung 3.5 – Loop Pipelining [Xil16b] Fig. 1-50

Abbildung 3.5 zeigt an einem Beispiel den Unterschied zwischen einer Schleife mit Pipelining (Loop Pipeline) und einer Schleife ohne Pipelining. Die Gesamtverzögerung einer Loop Pipeline kann anhand folgender Performancemetriken berechnen werden:

Initiation Interval
: Das Initiation Interval (II) gibt die Anzahl an Zyklen an, bevor die nächste Iteration einer Loop Pipeline anfängt Daten zu verarbeiten.

Iteration Latency
: Die Iteration Latency gibt die Anzahl an Zyklen für eine Iteration an.

Trip Count
: Der Trip Count gibt die Anzahl an Iterationen in einer Schleife an.

Loop Latency
: Die Loop Latency (LL) ist die Gesamtverzögerung einer Loop Pipeline und gibt die Anzahl an Zyklen für die Berechnung der gesamten Schleife an.

[Xil19a]

Ein II von 1 ist optimal, jedoch abhängig von der FPGA-Frequenz und den Operationen oder Speicherzugriffen innerhalb der Schleife nicht immer erreichbar [ZPM18].

Zu beachten ist, dass nur perfekte oder halb perfekte Schleifen als Pipeline implementiert werden können. Außerdem führt eine Synchronisationsbarriere innerhalb einer Loop Pipeline zur Aufteilung der Loop Pipeline in zwei Schleifen, da alle Iterationen die Barriere erreichen müssen, bevor die erste Instruktion der ersten Iteration nach der Barriere beginnt [Xil17a]. Da Parallelität in einer CU auf FPGAs durch Pipelining erreicht wird, ist der Performanceverlust durch eine Barriere höher, als bei GPUs [WHU17].

Task-Level Pipelining

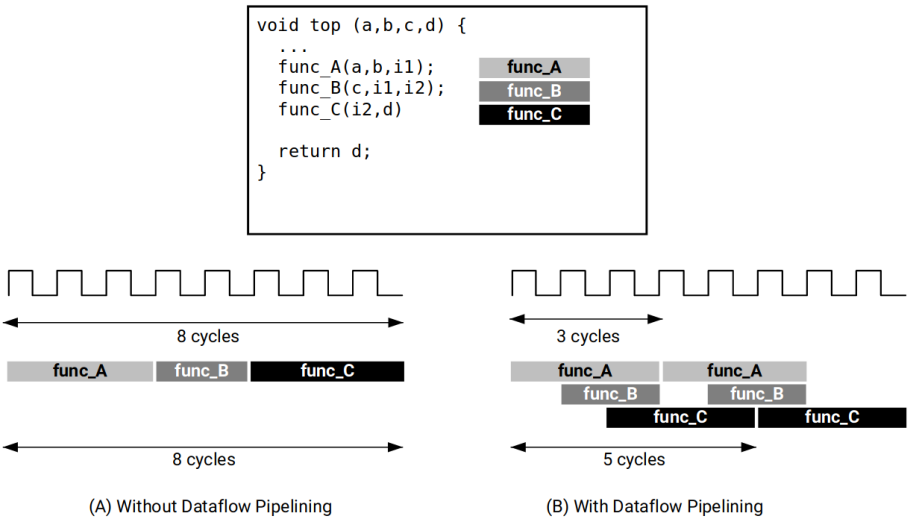


Abbildung 3.6 – Task-Level Pipelining [Xil19b] Fig. 8

Normalerweise werden Funktionen oder Schleifen innerhalb eines Kernels sequenziell ausgeführt und die gesamte Ausführungsdauer berechnet sich aus der Summe der Ausführungszeiten der einzelnen Aufgaben. Task-Level Pipelining erlaubt es, Funktionen oder Schleifen innerhalb eines Kernels parallel auszuführen (vgl. Abbildung 3.6). Diese Art der parallelen Ausführung wird auch als Task-Level Parallelism bezeichnet und von Xilinx Dataflow genannt[Xil17a].

Damit der OpenCL-Compiler Task-Level Parallelism synthetisieren kann, müssen bestimmte Coding Styles eingehalten werden. Um die erfolgreiche Optimierung zu garantieren wird es empfohlen, den Kernel Code nach dem *Load-Compute-Store Pattern* zu strukturieren (vgl. Abbildung 3.7) [Xil19b].

Load-Compute-Store Pattern

Beim Load-Compute-Store Pattern wird der Kernel Code in drei Funktionen oder Schleifen aufgeteilt:

Die *Load-Funktion* ist ausschliesslich dafür zuständig Daten effizient vom Global Memory zum Kernel zu bewegen. Die *Compute-Funktion* ist für die Datenverarbeitung zuständig und kann in mehrere Aufgaben aufgeteilt werden. Die *Store-Funktion* ist das Gegenstück zur Load Funktion indem sie die Daten vom Kernel in den externen Speicher bewegt (vgl. Abbildung 3.7) [Xil19a].

3.2.2 Array Partitioning

Arrays werden auf FPGAs typischerweise in BRAMs gespeichert, welche über maximal zwei Daten-Ports verfügen. Dadurch wird der interne Durchsatz bei Lese- oder Schreibintensiven Algorithmen limitiert und ein II von 1 ist möglicherweise nicht zu erreichen, falls das Array

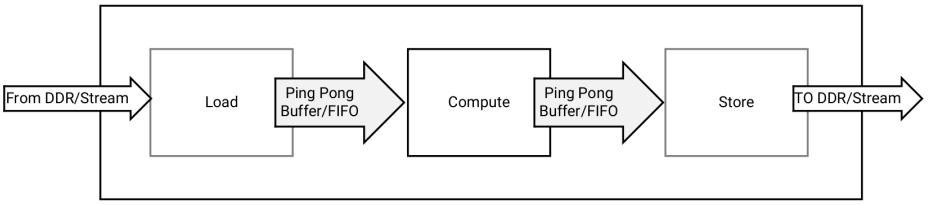


Abbildung 3.7 – Load-Compute-Store Pattern [Xil19a] Fig. 12

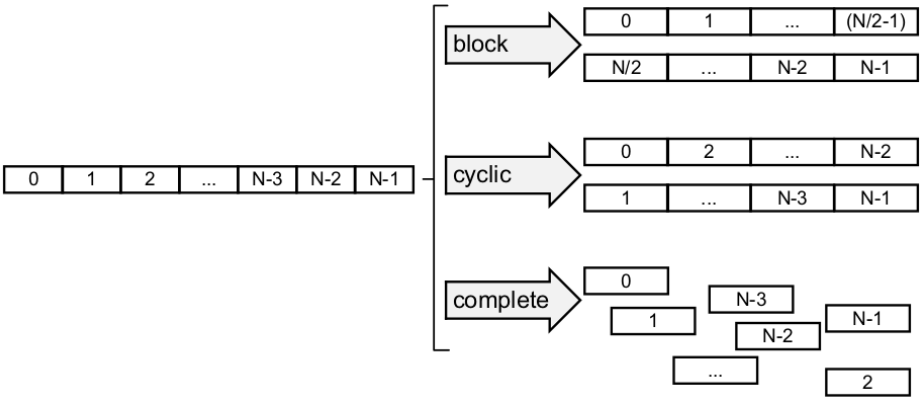


Abbildung 3.8 – Array Partitioning in einer Dimension jeweils mit Faktor 2 bei block und cyclic [Xil16b] Fig. 1-54

auf **einem** BRAM gespeichert ist und mehr als zwei Zugriffe parallel erfolgen.

Um die interne Bandbreite zu erhöhen, kann der Inhalt des Array auf mehrere BRAMs verteilt werden, um so die effektive Anzahl an Daten-Ports zu erhöhen und ein II von 1 zu erreichen. Abhängig vom Zugriffsmuster auf das Array können unterschiedliche Partitionierungstypen besser geeignet sein (vgl. Abbildung 3.8):

block
: Das Array wird in gleich große Blöcke aufeinanderfolger Elemente des ursprünglichen Array aufgeteilt.

cyclic
: Das Array wird in gleich große Blöcke aufgeteilt, wobei die Elemente des ursprünglichen Array abwechselnd auf die Blöcke verteilt werden.

complete
: Das Array wird komplett aufgeteilt, was der Auflösung des Speichers in Registern entspricht. Diese Partitionierung wird zur Implementierung eines Array als Schieberegister verwendet.

Array Partitioning erzeugt zusätzliche Objekte, welche auf dem FPGA platziert und verwaltet werden müssen, wodurch die maximale Frequenz des FPGA sinken und somit die Ausführungsdauer des Kernels steigen kann [Xil16b].

3.2.3 Loop Unrolling

Loop Unrolling fasst mehrere Iterationen einer Schleife zusammen und reduziert damit den Trip Count der Schleife, wodurch die Ausführung der Schleife beschleunigt werden kann. Der *Unroll-Faktor* gibt an, wie viele Iterationen zusammengefasst werden.

Auf FPGAs führt Loop Unrolling zu einer Erhöhung der benötigten Hardwareressourcen und der Bandbreitenanforderung für die Schleife um den Unroll-Faktor, da mehrere Iterationen parallel berechnet werden und somit zusätzliche ALUs instanziiert werden müssen und mehr Zugriffe parallel auf einen Speicher erfolgen.

Übersteigen die Anforderungen durch Loop Unrolling die vorhandenen Ressourcen, hat Loop Unrolling keinen weiteren positiven Effekt auf die Performance [Xil19a]. Loop Unrolling kann auch verwendet werden, um das Pipelining einer verschachtelten Schleife zu ermöglichen [WHU17].

3.2.4 Vektorisierung

OpenCL ermöglicht die Verwendung vorderfinierter Vektortypen, wie $float4$ oder $double8$, welche aus primiven Datentypen zusammengesetzt sind [Khr19]. Die Verwendung von Vektortypen hat einen ähnlichen Effekt wie Loop Unrolling.

Kernel Vectorization

Bei OpenCL kann ein Compiler-Hinweis über die Berechnungsbreite eines Kernels gegeben werden, wodurch der Compiler mehrere Work-Items zusammenführen oder ein Work-Item in mehrere Threads aufteilen kann, um die Hardware besser auszunutzen [Khr19].

3.2.5 Compute Unit Replication

Die Hardware für die Ausführung eines Kernels kann auf FPGAs repliziert werden, wodurch die parallele Ausführung mehrerer Kernels ermöglicht wird. Diese Art der Optimierung wird Compute Unit Replication (CUR) genannt und kann nur angewandt werden, falls die Hardwareressourcen ausreichen. CUR hat keinen positiven Effekt auf die Performance, falls die Performance schon bei einem Kernel durch die Bandbreite limitert ist [WHU17].

4 Methodik

Untersucht werden in dieser Arbeit Einflussfaktoren auf die Performance mittels OpenCL-implementierter 2D 5-Punkt Stencil-Kernels einer Iteration auf FPGAs (vgl. Abschnitt 2.3.2). Die Implementierung soll auf unterschiedlichen Inputgrößen ausführbar sein.

4.1 Metriken

Die verwendeten Metriken werden zur Quantifizierung von Aussagen über die Performance eines Kernels gemessen oder errechnet.

4.1.1 Gemessene Metriken

Ausführungszeit

Die zwischen Beginn und Ende der Ausführung eines Kernels vergangene Ausführungszeit t wird zur Bestimmung der Performance benötigt. Die Messung der Ausführungszeit wird automatisiert und mithilfe eines zuverlässigen Chronometer durchgeführt.

Hardwareressourcen

Die benötigten Hardwareressourcen zur Instanziierung eines Kernel werden nach der endgültigen Platzierung und dem Routing der FPGA-Komponenten gespeichert.

Maximale Frequenz

Die Gültigkeit einer Berechnung kann durch Überschreitung der für ein Kernel-Design maximal zulässigen FPGA-Frequenz f_{max} während der Ausführung beeinträchtigt werden. Die Berechnung wird nach jeder Ausführung auf Gültigkeit überprüft. Gültige Ergebnisse werden nur bis f_{max} erzielt.

4.1.2 Parameter

FPGA-Frequenz

Durch die Ausführung des FPGA mit unterschiedlichen Frequenzen kann zum einen der Einfluss der Frequenz auf die Ausführungszeit und zum anderen die maximale Frequenz des Kernels auf der verwendeten Hardware bestimmt werden.

Es gilt unter Vernachlässigung anderer Faktoren für die Abhängigkeit der Ausführungszeit von der FPGA-Frequenz:

$$t \propto f^{-1}$$

Verwendeter Datentyp

Es wird der in wissenschaftlichen Anwendungen typischerweise verwendete Datentyp *double* bei der Stencilberechnung verwendet. Die Größe des verwendeten Datentyp beträgt:

$$sizeof(double) = 8bytes$$

Problemgröße

Der Stencil-Algorithmus wird auf quadratischen Matrizen ausgeführt. Die Größe der Matrizen ist daher in beiden Dimension gleich und kann durch **eine** Variable definiert werden:

$$N := N_x = N_y$$

N wird im Folgenden als Problemgröße bezeichnet, da der Rechenaufwand von N abhängt.

Kernelgröße und Blockgröße

Die Kernelgröße k_{size} entspricht der von einem Kernel *berechneten* Größe eines Blocks und unterscheidet sich von der Blockgröße b_{size} darin, dass k_{size} ohne den Rand angegeben ist. Bei dem verwendeten 2D 5-Punkt Stencil gilt:

$$b_{size} = k_{size} + 2$$

Blockanzahl

Die Blockanzahl n_{blocks} gibt an, wie oft ein Kernel ausgeführt werden muss, um ein Problem zu berechnen.

Werden quadratische Blöcke verwendet gilt:

$$n_{blocks} = (N/k_{size})^2$$

Wird das Problem nur in x-Richtung in Blöcke unterteilt gilt:

$$n_{blocks} = N/k_{size}$$

N wird als ganzzahliges Vielfaches von k_{size} gewählt.

4.1.3 Abgeleitete Metriken

Performance

Die Anzahl an Fließkommaoperationen pro Sekunde wird in [FLOP/s] oder [FLOPs] angegeben und ist ein Maß für die Performance eines Kernels. Durch die Performance lassen sich, unabhängig von der **absoluten** Ausführungszeit einer Berechnung, Aussagen über die Geschwindigkeit einer Berechnung und dadurch über die Qualität eines Kernels treffen.
Für die Performance F eines Kernels gilt unter Vernachlässigung anderer Faktoren:

$$F \propto t^{-1} \Rightarrow F \propto f$$

Eine Abweichung der Performance vom linearen Frequenzverlauf bedeutet den Einfluss vernachlässigter Faktoren und kann Hinweise auf den Einfluss von Effekten des Datentransfers zu einem entfernten Speicher geben.

Bei einer Memory-Bound-Anwendung ist das Erreichen einer maximalen Performance ab einer Grenzfrequenz f_{opt} zu erwarten. Die so bestimmbare Grenzfrequenz kann zur Analyse und der Bestimmung der optimalen Frequenz des Kernels verwendet werden.

Bei dem untersuchten Stencil erfordert die Berechnung einer Zelle 5 Fließkommaoperationen – vier Additionen und eine Multiplikation (vgl. Abschnitt 2.3.2). Um die Performance eines Kernels auf einem Problem zu berechnen, wird die Summe aller Fließkommaoperationen durch die Ausführungszeit geteilt:

$$F = 5 \cdot N^2/t$$

Bandbreite

Die Bandbreite T gibt die Rate an, in welcher Daten verarbeitet werden. Um die Bandbreite zu berechnen wird das benötigte Datenvolumen V für ein Problem durch die Ausführungszeit geteilt [Xil19a]:

$$T = max(V_{in}, V_{out})/t$$

Für die in dieser Arbeit implementierte Stencilberechnung gilt:

$$V_{in} \geq V_{out} \Rightarrow T = V_{in}/t$$

Das aus dem Input transferierte Datenvolumen V_{in} wird aus der Anzahl an transferierten Daten und der Datentypgröße berechnet.

Für eine Implementierung ohne Spatial Blocking werden zur Berechnung eines Stencils jeweils 5 Daten geladen. Bei dem verwendeten Datentyp gilt für die Bandbreite:

$$T = 5 \cdot N^2 \cdot sizeof(double)/t$$

Wird Spatial Blocking mit quadratischen Blöcken verwendet gilt:

$$T = n_{blocks} \cdot b_{size}^2 \cdot sizeof(double)/t$$

Wird das Problem nur in x-Richtung in Blöcke unterteilt gilt[1]:

$$T = n_{blocks} \cdot (N+2) \cdot b_{size} \cdot sizeof(double)/t$$

[1] $N + 2$ aufgrund des, bei der verwendeten Implementierung geladenen, oberen und unteren Randes des Input.

Rechenintensität

Die Rechenintensität eines Algorithmus ist $q = \mathrm{f}/m$, wobei f die Anzahl der Grundoperationen (z.B. Fließkommazahl Additionen und Multiplikationen) und m die Anzahl der Wörter ist, die zwischen schnellem und langsamem Speicher verschoben werden.

4.1.4 Geschätzte Metriken

Energieeffizienz

Die während der Ausführung aufgenommene Leistung P wird gemessen, oder durch ein geeignetes Modell geschätzt. Die Leistung ist abhängig von der FPGA-Frequenz:

$$P_{total} = P_{switching} + P_{static}$$
$$P_{switching} = \alpha \cdot C_L \cdot V_{DD}^2 \cdot f \Rightarrow P_{switching} \propto f$$

Die erwartete Energieeffizienz F/P wird in [GFLOPs / W] angegeben.

Performancemetriken

Die zur Aufstellung des Performancemodells benötigten Performancemetriken einer Loop Pipeline werden nach der Kompilierung des Kernel-Code in eine HDL durch einen HLS-Compiler geschätzt und gespeichert.

4.2 Prozessablauf

4.2.1 Benchmarks

Jeder Kernel wird mit einer Reihe von FPGA-Frequenzen und auf verschiedenen Problemgrößen ausgeführt (vgl. Listing 4.1). Es werden bei jeder Ausführung Metriken gemessen und die Berechnung wird auf Gültigkeit überprüft.

```
function benchmark()
{
    for each (problem_size, frequency) do:
    {
        init(problem, fpga)
        start = get_time()
        kernel_exec(fpga)
        end = get_time()
        validate(problem)
    }
}
```

Listing 4.1 – Benchmark Pseudocode

Validierung der Berechnung

Die Berechnung des FPGA wird nach jeder Messung auf Gültigkeit überprüft, um die Zulässigkeit der Berechnung zu garantieren.

4.2.2 Performancenalyse und Optimierung

Eine Analyse der Performance erfolgt sowohl durch geeignete Visualisierung, als auch durch Sichtung der Messdaten nach jeder Messung. Die Erkenntnisse dieser Zwischenanalysen werden notiert und haben die Verbesserung des Verständnis der Einflussfaktoren auf die Performance zum Ziel. Zur quantifizierbaren Performanceanalyse zusätzlich dient ein geeignetes Performancemodell.

Optimierung

Es werden nach jeder Messung Modifikationen am Kernel vorgenommen. Die Modifikationen basieren auf den, aus der Analyse gesammelten, Erfahrungen und dienen zur Erforschung der Auswirkung einer Modifikation auf die Performance.

Kernel-Modifikationen werden durch Parametervariation, Änderung der Art der Implementierung und Anwendung FPGA-spezifischer Optimierungsmethoden vorgenommen. Basierend auf den Erkenntnissen vorausgehender Performanceanalysen können Kernels gezielt optimiert werden.

Performance-Referenz

Ausgangspunkt der Analysen ist eine naive Implementierung – die Baseline. Die Baseline hat die Referenzierbarkeit der folgenden Messungen zum Ziel.

Um eine obere Performanceschranke einer Memory-Bound-Anwendung zu bestimmen, wird vor Beginn der eigentlichen Messungen ein Memory Benchmark durchgeführt. Die gemessene Schranke dient als zusätzliche Referenz der erreichten Performance und kann zur Definition des Endes eines Optimierungsprozess verwendet werden [Xil16a].

Simulation

Betrifft eine Kernel-Modifikation den untersuchten Stencil-Algorithmus, wird der Algorithmus in einer Simulation geprüft, um die Korrektheit zu garantieren, bevor der Bitstream eines Kernels kompiliert wird. Ziel der Simulation ist eine Beschleunigung des Debugging.

Optimierungsmethoden

Es werden folgende Optimierungsmethoden (vgl. Kapitel 3) untersucht:

1. Spatial Blocking
2. Burst Memory Transfer
3. Loop Pipelining
4. Task-Level Pipelining
5. Array Partitioning
6. Loop Unrolling
7. Vektorisierung
8. CUR

4.2.3 Struktur

Die Performanceanalyse erfordert eine wohlgeordnete Struktur der Messdaten. Jeder Kernel erhält eine Kernel-ID. Kernel-Informationen werden in separaten Ordnern verwaltet und verschiedenartige Implementierungen werden zur besseren Referenzierbarkeit systematisch benannt. Alle Informationen werden in einem einfachen maschinenlesbaren Format gespeichert.

Die Struktur hat das Ziel der Fehlervermeidung und Erleichterung einer methodischen und automatisierten Auswertung.

Automatisierung

Es werden möglichst viele Prozesse automatisiert, um sowohl eine Reproduzierbarkeit zu garantieren, als auch das Auftreten von Fehlern zu minimieren.

Nach Möglichkeit werden folgende Prozesse, oder Teile der Prozesse automatisiert:

1. Ausführung der Messung
2. Nummerierung der Messung
3. Übertragen und Speichern von Messergebnissen
4. Visualisierung der Messung
5. Auswertung der Messung
6. Kompilierung
7. Kopieren von Log-Dateien

Reproduzierbarkeit

Um die Reproduzierbarkeit der Ergebnisse zu garantieren, wird die Konfiguration der verwendeten Hardware- und Softwarekomponenten nach Möglichkeit von der ersten bis zur letzten Durchführung einer Messung unverändert beibehalten. Eine Änderung der Konfiguration kann die zwingende Wiederholung aller vorausgehenden Messungen zufolge haben. Eine gute Planung kann die Notwendigkeit der nachträglichen Anpassung der Messbedingungen vermeiden. Ist eine Änderung der Konfiguration unvermeidlich, müssen alle davon betroffenen Messungen konsequent wiederholt werden.

4.3 Performancemodell

Ein Performancemodell bestimmt die erwartete Performance eines Kernels und kann zum Vergleich mit der gemessenen Performance und zur zielgerichteten Optimierung nützen. Eine deutliche Abweichung einer Messung vom Modell deutet auf den Einfluss eines Effekts hin, welcher im Performancemodell nicht berücksichtigt wird.

4.3.1 Verwendetes Performancemodell

Das verwendete Performancemodell basiert auf der Anzahl an, zur Ausführung einer Schleife benötigten, Taktzyklen und vernachlässigt absichtlich Effekte, welche typischerweise durch

Schnittstellen zu einem Arbeitsspeicher auftreten können. Bei der Analyse können mithilfe eines Performancemodells Aussagen über die Güte des Modells und das Verhalten der gemessenen Kernel-Performance in Bezug auf die erwartete Performance getroffen werden.

4.3.2 Erwartete Performance einer Schleife

Die erwartete Ausführungszeit für den kompletten Durchlauf **einer** (verschachtelten) Schleife eines Kernels kann anhand der LL der Schleife bestimmt werden. Die LL kann durch die Performancemetriken berechnet werden.

Eine Schleife ohne Loop Pipelining wird sequenziell ausgeführt und es gilt:

$$LL = IterationLatency \cdot TripCount$$

Mit Loop Pipelining gilt:

$$LL = IterationLatency + II \cdot (TripCount - 1)$$

Anhand der LL kann die erwartete gesamte Ausführungszeit $\hat{t}$ bei gegebener Problemgröße und FPGA-Frequenz berechnet werden:

$$\hat{t} = n_{blocks} \cdot LL \cdot f^{-1}$$

Mit $\hat{t}$ kann die erwartete Performance $\hat{F}$ und Bandbreite $\hat{T}$ berechnet werden (vgl. Abschnitt 4.1.3).

4.3.3 Performancevergleich

Die LL ist ungeeignet für den Vergleich mit anderen Kernels, da die Ausführungszeit auf einem Problem von k_{size} und N abhängt. Um den Vergleich zu erleichtern, wird die Latency per Update (LPU) definiert, welche das Verhältnis der LL einer Schleife zu Stencil-Updates angibt:

$$LPU := \begin{cases} LL/k_{size}^2 & , \quad quadratische\ Blöcke \\ LL/(k_{size} \cdot N) & , \quad y - streaming \end{cases}$$

4.3.4 Folgerung aus dem Performancemodell

Ein hoher Trip Count versteckt die Iteration Latency bei der Ausführungszeit einer Loop Pipeline:

$$LL = IterationLatency + II \cdot (TripCount - 1)$$
$$\Rightarrow LL \approx II \cdot TripCount \quad \text{für:} \quad TripCount \gg IterationLatency$$

Zusätzlich kann gefolgert werden, dass die LL und somit die Performance einer Loop Pipeline bei hohem Trip Count vom II der Schleife dominiert wird. Eine weitere Folgerung

ist, dass die Reduzierung des Trip Count durch Loop Unrolling die Performance nur dann verbessert, wenn das Verhältnis von II zu Unroll Factor sinkt.

4.3.5 Performancemodell unter Berücksichtigung einer Schnittstelle

Wird die Limitierung der Performance durch die maximale Bandbreite T_{max} der verwendeten Schnittstellen berücksichtigt, lässt sich das zuvor aufgestellte Performancemodell erweitern:

$$\hat{F}' = \begin{cases} \hat{F} & , \quad \hat{T} \leq T_{max} \\ F_{max} & , \quad \hat{T} > T_{max} \end{cases}$$

Dieses Performancemodell sagt einen linearen Anstieg der Performance mit der Frequenz bis zu einem Grenzwert F_{max} voraus. Erreicht die Performance F_{max}, ist ein Knick im Verlauf zu erwarten. Die Rechenintensität eines Algorithmus bestimmt den Zusammenhang von T_{max} und F_{max}.

Maximale Performance des untersuchten Stencils

Wird der Rand vernachlässigt[2], kann F_{max} des untersuchten 2D 5-Punkt Stencils bei optimaler Datenwiederverwendung bestimmt werden:

$$F_{max} = q \cdot T_{max} = 5FLOP/sizeof(double) \cdot T_{max}$$

F_{max} dient neben der Baseline als weitere Performance-Referenz oder als Input eines Performancemodells.

[2]Daten am Rand werden weniger oft verwendet. Daten in der Mitte werden bei 5 Berechnungen verwendet.

5 Evaluation

In diesem Kapitel wird zunächst die zur Evaluation verwendete Hard- und Software vorgestellt. Darauf folgt eine Beschreibung der untersuchten Implementierungen und die erzielten Messergebnisse.

5.1 Hardware

Zur Ausführung und Messung wurde das Zybo Z7-10 (ZYBO) Entwicklerboard der Firma Digilent verwendet, auf dem ein FPGA der Firma Xilinx verbaut ist. Im Folgenden werden die Architektur des ZYBO und wichtige technische Details des ZYBO erläutert.

5.1.1 ZYBO

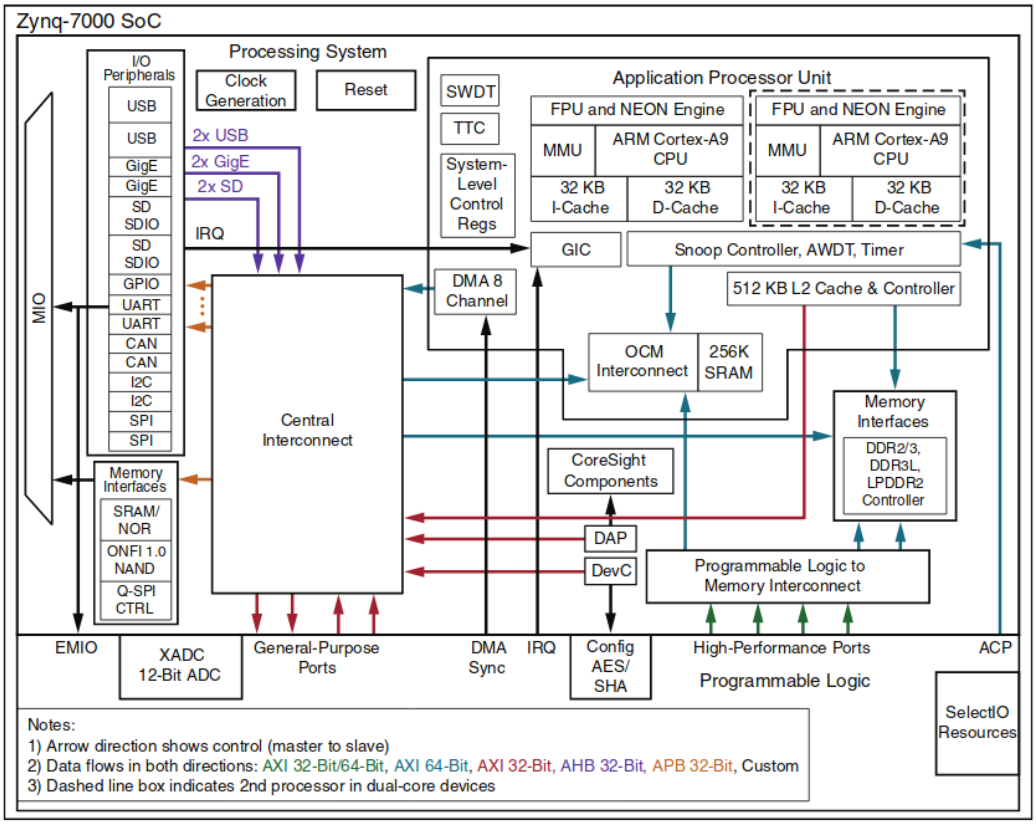


Abbildung 5.1 – ZYBO Board Layout [Xil18d] Fig. 1

Das ZYBO ist ein relativ kleines und kostengünstiges Entwicklerboard im Vergleich zu teuren FPGA-Boards, wie Intels Arria 10 oder dem Zynq UltraScale+ von Xilinx (vgl. Tabelle 5.1).

Das Board wird in zwei getrennten Teilen betrachtet:

Processing System
: Das Processing System (PS) umfasst im Wesentlichen die Input/Output-Devices, einen Dual-Core ARM-Prozessor, einen DDR-Arbeitsspeicher und den Memory Controller (vgl. Abbildung 5.1 oben).

Programmable Logic
: Die Programmable Logic (PL) besteht aus einem FPGA der Firma Xilinx und den Schnittstellen zum PS (vgl. Abbildung 5.1 unten).

Der ARM-Prozessor des ZYBO übernimmt die Rolle des OpenCL Host und agiert als Controller des FPGA. Programmiert wird das PS und die PL über eine USB-Schnittstelle, welche zusätzlich zur Datenübertragung und zur Stromversorgung genutzt wird.

Die PL ist über vier High-Performance-Schnittstellen mit dem selben Memory Controller verbunden, welcher von der Host CPU verwendet wird (vgl. Abbildung 5.2). Andere übliche FPGA-Board-Architekturen beinhalten einen dedizierten DDR-Arbeitsspeicher für den FPGA.

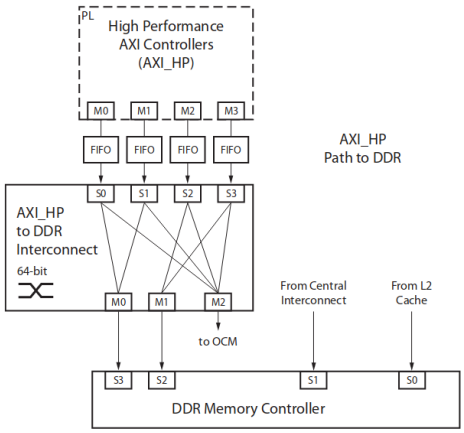


Abbildung 5.2 – ZYBO AXI interconnect [Xil18d] Fig. 10-2

Board	Digilent Zybo Z7-10
CPU	667MHz dual-core Cortex-A9
RAM	1 GB DDR3L with 32-bit bus @ 1066 MHz
FPGA Part Number	XC7Z010-1CLG400C
Logic slices	4400
6-input LUTs	17600
Flip-Flops	35200
Block RAM	2.1Mbit (60 * 36 kbit Blocks)
DSP Slices	80

Tabelle 5.1 – ZYBO Hardware-Spezifikation [Xil18e]

5.1.2 Schnittstellen

Das ZYBO verfügt über mehrere verschiedene konfigurierbare Schnittstellen vom PS zur PL, von denen die High-Performance- und General-Purpose-Schnittstellen verwendet wurden.

Eine General-Purpose-Schnittstelle wird als Schnittstelle zu den Kontrollregistern eines Kernels auf dem FPGA eingesetzt. Die High-Performance-Schnittstellen dienen zur Übertragung großer Datenmengen vom und zum FPGA.

Eine High-Performance-Schnittstelle verfügt über die doppelte Band- und Busbreite im Vergleich zur General-Purpose-Schnittstelle (vgl. Tabelle 5.2).

Interface	Bus [bit]	IF Takt [MHz]	Read [MB/s]	Write [MB/s]	R+W [MB/s]	#Interfaces
General-Purpose AXI	32	150	600	600	1200	2
High-Performance AXI_HP	64	150	1200	1200	2400	4
DDR	32	1066	4264	4264	4264	1

Tabelle 5.2 – ZYBO Schnittstellen-Spezifikation [Xil18d]

5.2 Software

Von der Beschreibung eines Kernels in OpenCL bis zur Ausführung auf einem FPGA sind mehrere Schritte notwendig. Mit der *Vivado Tool Suite* bietet Xilinx ein Softwarepaket für die Entwicklung von Xilinx-kompatiblen Hardwarebeschleunigern. Sie besteht aus drei verschiedenen Softwaretools, welche für die Schritte bis zur Ausführung verwendet werden können.

5.2.1 Vivado HLS

Zunächst muss eine Hardwarebeschreibung der OpenCL Kernels in einer HDL für den verwendeten FPGA generiert werden. Als Zielsprache wurde VHDL gewählt und zur HLS die Software *Vivado HLS 2019.1* von Xilinx verwendet. Neben einem HLS-Compiler bietet diese Software verschiedene Profiling- und Analysewerkzeuge.

Nach der HLS können die benötigten Hardwareressourcen und die Performancemetriken eines Kernels abgelesen werden. Diese Werte sind Schätzungen und können nach der endgültigen Platzierung des Kernels und der I/O-Funktionalität auf dem FPGA von den tatsächlichen Werten abweichen.

Im Analysetool von *Vivado HLS 2019.1* können Faktoren ermittelt werden, welche die Frequenz des FPGA limitieren, oder das II erhöhen, um den OpenCL Code gezielt anzupassen und die erwartete Performance zu verbessern. Jeder Kernel wurde in Vivado HLS mit der für das Design maximalen Frequenz kompiliert. Da von Vivado HLS nur eine Hardwarebeschreibung generiert wird und die endgültige Platzierung der Komponenten später erfolgt, kann die tatsächliche maximale Frequenz abweichen.

Schnittstellensynthese und OpenCL in Vivado HLS

Während der HLS gruppiert Vivado HLS alle Schnittstellen eines OpenCL Kernels folgendermaßen (vgl. Listing 5.1):

Skalare Schnittstellen
Alle skalaren Schnittstellen werden in eine einzige AXI4-Lite Schnittstelle gruppiert.

Array- und Pointer-Schnittstellen
Alle Array- und Pointer-Schnittstellen werden in eine einzige AXI4 Schnittstelle gruppiert.

Keine weiteren Schnittstellenspezifikationen sind bisher in Vivado HLS bei **OpenCL** Kernels erlaubt [Xil16b]. Es ist beispielsweise nicht möglich, den Input und Output eines Kernels in separate Schnittstellen zu gruppieren und anschliessend mit unterschiedlichen High-Performance-Schnittstellen zu verbinden. Die C-Synthese in Vivado HLS und die OpenCL-Synthese von *SDAccell* bietet mehr Konfigurationsmöglichkeiten zur Schnittstellensynthese [Xil19a].

Benötigt eine Anwendung eine hohe Bandbreite zwischen Kernel und Global Memory, sollte der verwendete Datentyp der Pointer-Schnittstellen an die Busbreite der Schnittstellen zum Global Memory angepasst werden. Die Breite der synthetisierten Pointer-Schnittstelle definiert die Breite des Datenpfades zwischen Kernel und Memory Controller (vgl. Listing 5.1) [Xil16a].

```
1  __kernel void example (
2
3    __global float  *gmem0,  //  32-bit Pointer-Schnittstelle
4    __global double *gmem1,  //  64-bit Pointer-Schnittstelle
5    __global float4 *gmem2,  // 128-bit Pointer-Schnittstelle
6             int     size    //          skalare Schnittstelle
7                           )
8  {
9      /* some code */
10 }
```

Listing 5.1 – Schnittstellensynthese

5.2.2 Vivado

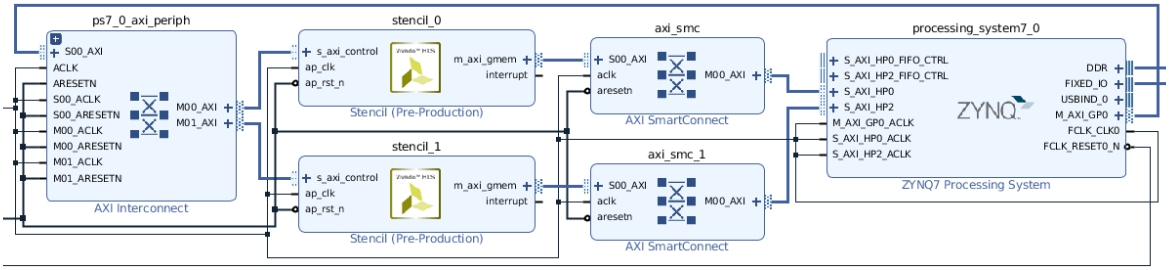


Abbildung 5.3 – Ausschnitt aus einem Vivado Blockdiagramm mit zwei Stencil-Kernels. Die Kernels sind über separate Schnittstellen zum Global Memory (gmem) mit dem PS verbunden

Mit der Software *Vivado 2019.1* wird die Hardwarebeschreibung eines Kernels in ein Design für ein spezifisches FPGA-Board integriert. Dazu wird ein Blockdiagramm erstellt, in dem die Komponenten eines Designs für den ZYBO konfiguriert und verbunden werden können (vgl. Abbildung 5.3).

Sollen mehrere Kernels auf der PL instanziiert werden, können die gruppierten Pointer-Schnittstellen jedes Kernels im Blockdiagramm über eine separate High-Performance-Schnittstelle mit dem PS verbunden werden.

Nachdem das Design des Blockdiagramms abgeschlossen und verifiziert ist, kann die endgültige Platzierung und Verschaltung der Komponenten der PL in einem weiteren Kompiliervorgang erfolgen. Als Ergebnis wird der FPGA-Bitstream der Anwendung und ein Board Support Package (BSP) generiert. Das BSP beinhaltet die Initialisierung beim Starten des Geräts und die Gerätetreiber für eine gegebene Hardware-Plattform [Xil17b].

Hardwareressourcen und Leistung

Nach der Platzierung und dem Routing werden von Vivado die verwendeten Hardwareressourcen und die zu erwartende aufgenommene Leistung bei der Ausführung eines Designs ausgeben.

Die von Vivado berechnete Leistung eines Kernels bezieht auf $f = 250\text{MHz}$, da die Kompilierung mit einer Board-Konfiguration dieser Frequenz ausgeführt wurde. Zur Berechnung der aufgenommenen Leistung eines Kernels wurde $P_{switching}$ der verwendeten FPGA-Komponenten aufsummiert. $P_{switching}$ ist abhängig von der FPGA-Frequenz und der tatsächliche Wert wird bei anderen Frequenzen als 250MHz stärker abweichen.

5.2.3 Xilinx Software Development Kit (SDK)

Nachdem der Bitstream und das BSP zusammengestellt wurden, kann der Kernel auf dem ZYBO ausgeführt werden. Mit der Software *Xilinx SDK* kann die PL des ZYBO mit dem Bitstream programmiert und das PS mit dem BSP initialisiert und gestartet werden. Zur Messung wird der Host Code auf der ARM CPU des ZYBO ausgeführt. Der Host Code enthält den Programmablauf der Benchmarks und den Kontrollcode zur Ausführung der OpenCL Kernels auf dem FPGA.

Xilinx SDK bietet auch Tools zur Performanceanalyse der verwendeten Schnittstellen von PS zu PL. Eine Messung und Analyse der Bandbreite und Verzögerung der Schnittstellen wird allerdings erst durch Integration eines *AXI Performance Monitor* in das Design für den ZYBO ermöglicht [Xil14].

Kernel Ausführung

Da die OpenCL-Host-Code-Umgebung *SDAccell* von Xilinx das ZYBO nicht unterstützt, wird die OpenCL-Funktionalität für das ZYBO manuell implementiert. Zur Ausführung von Kernels muss die Funktionalität der OpenCL-Funktionen *clEnqueueNDRangeKernel*, *clEnqueueTask* und *clSetKernelArg* implementiert werden.

Die Implementierung der OpenCL-Funktionalität und der gemessenen Kernels ist auf GitHub verfügbar [Bre20].

Zur Messung der Ausführungszeit wurde die *Xilinx Standalone Library* Funktion *XTime_GetTime* verwendet, welche die Anzahl an vergangenen Taktzyklen der ARM CPU auf den ZYBO misst [Xil18b].

5.3 Implementierungen

In diesem Abschnitt werden Implementierungen auf dem ZYBO evaluiert. Zunächst werden die Implementierungen beschrieben und deren Performancemetriken angegeben. Die Performancemetriken sind von Vivado HLS nach der Synthese entnommen.

Variationen der Implementierungen wie z.B. Kernelgröße oder Optimierungen werden als Subskript zur nummerierten Implementierung angegeben. Die Kernelgröße beträgt 32, wenn nicht anders angegeben.

In Tabelle 5.4 werden die Messresultate zusammengefasst. Darauf folgt eine graphische Aufarbeitung der Implementierungen und Optimierungen.

Verwendete Problemgrößen und Frequenzen in [MHz]:

$$f \in \{50, 66, 100, 133, 153, 200, 250, 285, 333\}$$

$$N \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$$

5.3.1 Memory Benchmark

Um die Bandbreite der verwendeten High-Performance-Schnittstelle zu messen, wurde ein Kernel (MEMCPY) implementiert, welcher ausschliesslich Daten im BMT-Modus vom Global Memory zum FPGA und zurück bewegt. In einer weiteren Messung wurde der selbe Kernel dahingehend modifiziert, dass er Daten atomar bewegt (MEMCPY$^{\neg_{BMT}}$), um den Effekt von BMTs zu isolieren.

Beide Kernels bewegen zeilenweise 128 Daten aus einer Inputmatrix und das Zugriffsmuster ähnelt dem der Kernels mit Spatial Blocking (vgl. Listing 8.1).

5.3.2 IMPL0

Die Baseline (IMPL0) ist eine naive Implementierung ohne Optimierungen. Daten werden nicht wiederverwendet und das Zugriffsmuster ist willkürlich (vgl. Listing 8.2).

Als erste Optimierung wurde Loop Pipelining auf die Baseline angewandt und gemessen. Das II der Loop Pipeline beträgt 5 bei IMPL0, da in jeder Iteration 5 Zellen geladen werden.

5.3.3 IMPL1

IMPL1 ist ein NDRange Kernel und eine Optimierung von IMPL0 durch Spatial Blocking. Die Work-Group-Größe ist bei IMPL1 gleich k_{size} – jede Work-Group deckt einen zu berechnenden Teil des NDRange ab und einzelne Work-Groups überlappen nicht (vgl. Listing 8.3).

Durch die notwendige Barriere zur Synchronisation der Work-Items wird die Loop Pipeline in zwei Schleifen aufgeteilt – eine Schleife (*Loop*1) vor der Barriere für das Laden der Daten und eine Schleife (*Loop*2) nach der Barriere für die Berechnung und das Speichern der Daten.

Bei Spatial Blocking werden an den Rändern jedes Blocks Daten benachbarter Blöcke zur Berechnung benötigt. Falls die Work-Groups nicht überlappen ist eine 1:1 Zuordnung von Work-Item zu Inputzelle nicht möglich. IMPL1 verwendet eine 1:4 Zuordnung, bei der $(b_{size}/2)^2$ Work-Items vier Zellen aus dem Global Memory in den Local Memory laden. Das II von *Loop*1 beträgt dadurch $II_1 = 4$.

In *Loop*2 werden in einer Iteration 5 Zellen aus dem Local Memory parallel zur Berechnung des Stencils benötigt. Der Local Memory ist im Dual-Port BRAM implementiert. Das II von *Loop*2 beträgt dadurch $II_2 = \lceil 5/2 \rceil = 3$.

Um den Einfluss von Spatial Blocking und Loop Pipelining zu messen, wurde IMPL1 sowohl mit, als auch ohne Loop Pipelining gemessen. Der Vergleich von IMPL1 ohne Loop Pipelining (IMPL1$\neg_{LP}$) zu IMPL0 ermöglicht Aussagen über die Auswirkung von Spatial Blocking auf die Performance. Der Vergleich von IMPL1 zu IMPL1$\neg_{LP}$ ermöglicht Aussagen über Loop Pipelining.

5.3.4 IMPL2

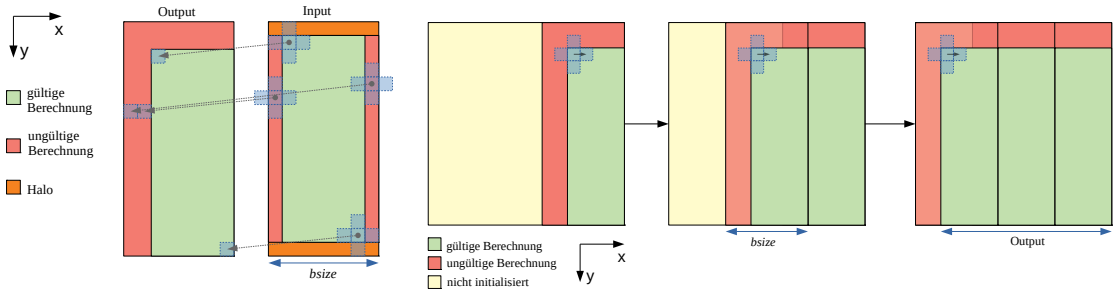
Als algorithmische Optimierung von IMPL1 wurde in IMPL2 die Work-Group-Größe gleich b_{size} gesetzt. Die Work-Groups überlappen an den Rändern jedes Blocks und eine 1:1 Zuordnung der Work-Items zu den benötigten Zellen eines Blocks ist möglich. Das II von *Loop*1 reduziert sich auf $II_1 = 1$ und das Laden wird im BMT-Modus synthetisiert.

Als zweite Optimierung wurde auf IMPL2 Array Partitioning angewandt, wodurch sich das II von *Loop*2 auf $II_2 = 1$ reduziert.

Es wurde ein Kernel ohne Array Partitioning (IMPL2$\neg_{AP}$) gemessen, um den Unterschied zu IMPL1 und den Einfluss von Array Partitioning zu messen. Um den Einfluss von k_{size} bei IMPL2 zu messen, wurde IMPL2 mit $k_{size} = 16$, $k_{size} = 32$ und $k_{size} = 64$ gemessen.
Der Kernel IMPL2$_{64}$ wurde bei IMPL2$_{CUR}$ repliziert – die Performancemetriken sind daher gleich (vgl. Tabelle 5.3).

5.3.5 IMPL3

IMPL3 ist ein Single-Task Kernel, besteht aus einer Schleife und implementiert Spatial Blocking mit einem Schieberegister als zyklischen Puffer (vgl. Listing 8.5).



(a) Berechnung eines Blocks. Die Berechnung innerhalb eines Blocks erfolgt in x-Richtung.

(b) Berechnung des gesamtem Output. Blöcke werden entgegen der x-Richtung abgearbeitet.

Abbildung 5.4 – Visualisierung der Schieberegister Implementierung

Um eine gesonderte Behandlung ungültiger Berechnungen am Rand jedes Blocks zu vermeiden, wurde der Output der Berechnung bei IMPL3 transformiert:

$$\tilde{out}(x, y) = \begin{cases} out(x-1, y) & , \quad x > 0 \\ out(bsize-1, y-1) & , \quad x = 0 \end{cases}$$

Ungültige Berechnungen, welche am Rand eines Blocks entstehen, werden dadurch auf die selbe Seite des Blocks – links von gültigen Berechnungen – im Output geschrieben. Werden die Blöcke in der richtigen Reihenfolge bearbeitet, können ungültige Berechnungen in der nächsten Blockiteration überschrieben werden und es entsteht ein zusammenhängender gültiger Output (vgl. Abbildung 5.4).

Ziel der Transformation ist das Ausführen aller Speicheroperationen im BMT-Modus durch Eliminierung bedingter Anweisungen. Erreicht wird die Transformation durch Erweiterung des Schieberegisters um ein Element.

Ein Nachteil von IMPL3 ist, dass die Blockreihenfolge entscheidend für die Korrektheit ist, wodurch eine Parallelisierung durch CUR erschwert wird.

Gemessen wurde IMPL3 mit $k_{size} = 32$, $k_{size} = 64$ und $k_{size} = 128$. Bei IMPL3$^{\neg_{BMT}}$ werden ausschliesslich Berechnungen und Speicheroperationen an gültigen Stellen vorgenommen, wodurch der HLS-Compiler die Speicheroperationen nicht im BMT-Modus synthetisiert.

Die y-Richtung wird bei IMPL3 gestreamed, wodurch der Trip Count und somit die LL von der Größe des Input in y-Richtung abhängt, welche erst zur Laufzeit bekannt ist. In Tabelle 5.3 sind die Performancemetriken bei $N = 2048$ angegeben.

5.3.6 IMPL4

IMPL4 ist nach dem Load-Compute-Store Pattern implementiert und besteht aus drei Schleifen. Es werden auf dem FPGA zwei Puffer im BRAM instanziiert, welche den Input und Output eines Blocks zwischenspeichern können, was sich in den erhöhten BRAM-Ressourcen

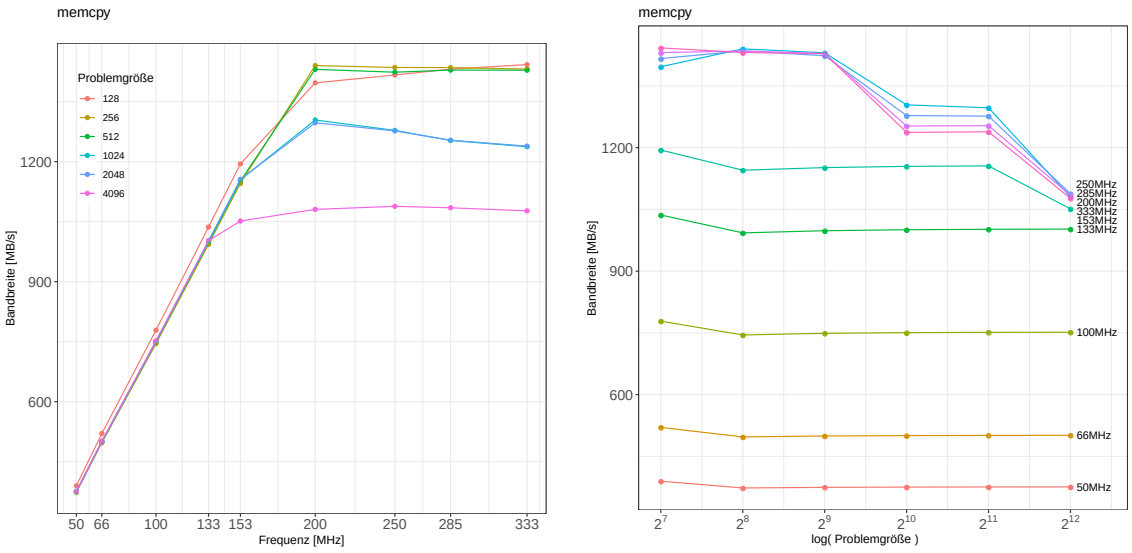
widerspiegelt. Das Lesen und Schreiben der Daten wird im BMT-Modus ausgeführt.

Um den Effekt von Task-Level Pipelining zu untersuchen wurde IMPL4 mit und ohne Dataflow-Optimierung (IMPL4$\neg_{DF}$) gemessen (vgl. Listing 8.6).

5.4 Ergebnisse

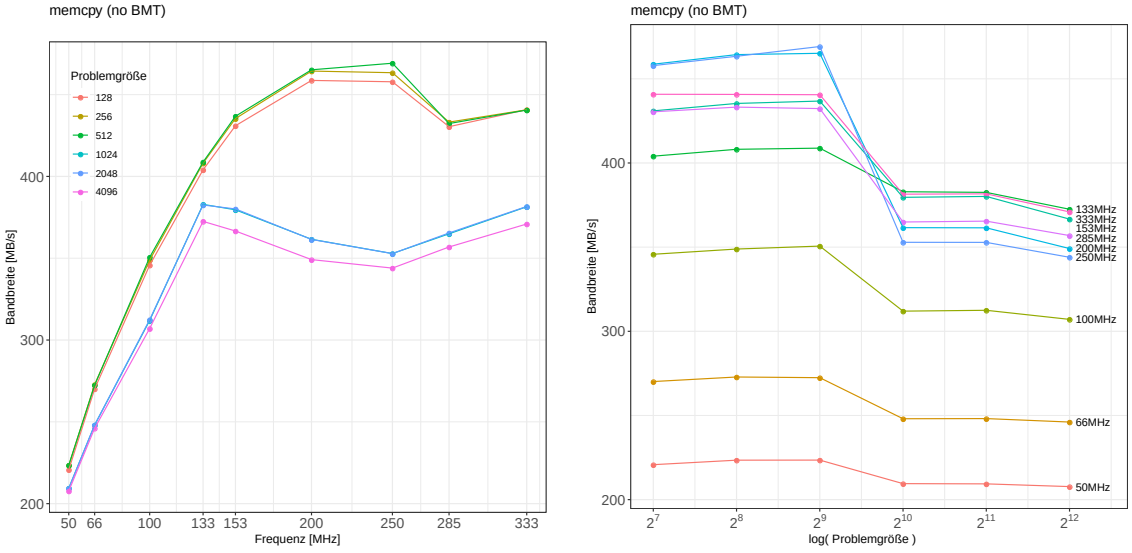
			Loop1			Loop2			Loop3		
Implementierung	**LPU**	LL	II_1	IL_1	LL_1	II_2	IL_2	LL_2	II_3	IL_3	LL_3
IMPL0	334	342017	-	334	-						
IMPL0$_{LP}$	5.32	5450	**5**	334	5448						
IMPL1$\neg_{LP}$	354	362498	-	148	-	-	206	-			
IMPL1	7.34	7517	**4**	148	4234	**3**	206	3274			
IMPL2$\neg_{AP}$	4.86	4981	1	152	1306	**3**	207	3671			
IMPL2$_{16}$	3.93	1007	1	150	472	1	209	531			
IMPL2$_{32}$	2.61	2673	1	152	1306	1	209	1363			
IMPL2$_{64}$	2.22	9075	1	154	4508	1	209	4563			
IMPL2$_{CUR}$	2.22	9075	1	154	4508	1	209	4563			
IMPL3$\neg_{BMT}$	1.069	70040	1	341	-						
IMPL3$_{32}$	1.069	70076	1	376	-						
IMPL3$_{64}$	1.035	135676	1	377	-						
IMPL3$_{128}$	1.018	266877	1	378	-						
IMPL4$\neg_{DF}$	3.48	3571	1	152	1306	1	72	1044	1	143	1165
IMPL4$_{32}$	2.48	2540	1	-	1308	1	-	1096	1	-	1167
IMPL4$_{64}$	2.15	8814	1	-	4510	1	-	4168	1	-	4239

Tabelle 5.3 – Performancemetriken. IL: *IterationLatency*. LL hängt bei IMPL3 von N ab und ist für $N = 2048$ angegeben. Die IL einzelner Schleifen wird nach der Dataflow-Optimierung von IMPL4 von Vivado HLS nicht angegeben.



(a) MEMCPY Frequenzabhängigkeit **(b)** MEMCPY Problemgrößenabhängigkeit

Abbildung 5.5 – MEMCPY

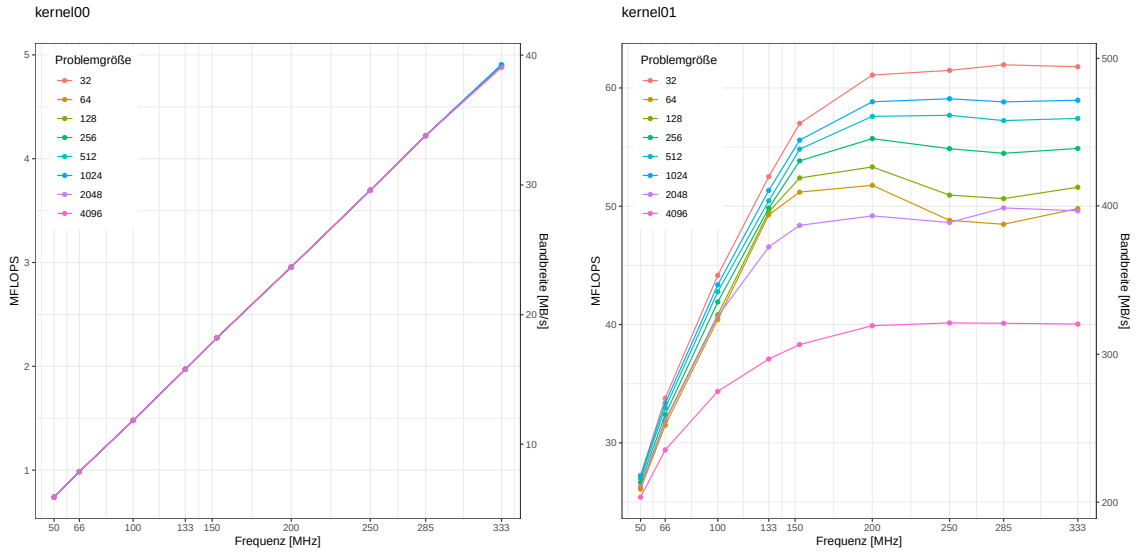


(a) MEMCPY$\neg_{BMT}$ Frequenzabhängigkeit **(b)** MEMCPY$\neg_{BMT}$ Problemgrößenabhängigkeit

Abbildung 5.6 – MEMCPY$\neg_{BMT}$

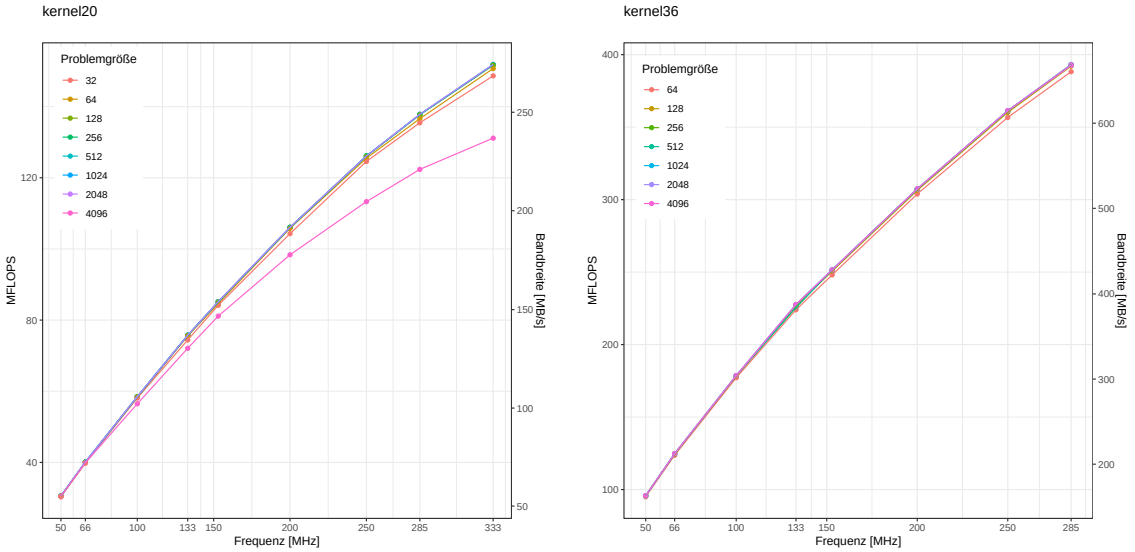
	K. ID	P [W]	LUT [%]	FF [%]	DSP [%]	BRAM [%]	CLB [%]	f_{max} [MHz]	MFLOPS	T [MB/s]	k_{size}	GFLOPS / W	Spat. Bick.	Loop Pipel.	Array Part.	Task Pipel.	Burst R/W	Implementierung
NDRange	0	0.23	27.39	21.08	32.50	3.33	48.55	333	4.88	39.06	32	0.02						IMPL0
	1	0.26	29.22	20.47	32.50	3.33	46.32	333	49.62	396.94	32	0.19						IMPL0$_{LP}$
	19	0.20	28.61	20.81	28.75	15.00	49.02	250	5.57	10.06	32	0.03	✓					IMPL1$^{\neg}{}_{LP}$
	20	0.30	34.34	24.20	32.50	15.00	57.34	333	151.85	274.28	32	0.50	✓	✓				IMPL1
	34	0.26	33.13	23.25	28.75	11.67	54.00	333	325.49	587.91	32	1.24	✓	✓			R	IMPL2$^{\neg}{}_{AP}$
	35	0.21	42.38	28.03	36.25	18.33	65.57	250	239.36	484.70	16	1.13	✓	✓	✓		R	IMPL216
	30	0.24	43.08	28.24	36.25	18.33	66.91	285	352.88	637.38	32	1.45	✓	✓	✓		R	IMPL232
	36	0.26	43.44	28.45	36.25	36.67	66.84	285	393.15	668.96	64	1.52	✓	✓	✓		R	IMPL264
	37	0.50	85.14	56.06	72.50	73.33	99.02	250	647.29	1101.41	64	1.30	✓	✓	✓		R	IMPL2$_{CUR}$
Single-Task	47	0.25	39.06	28.04	32.50	3.33	64.73	285	223.30	380.00	32	0.88	✓	✓	✓			IMPL3$^{\neg}{}_{BMT}$
	46	0.47	55.85	32.81	32.50	3.33	81.98	285	693.55	1180.00	32	1.49	✓	✓	✓		R+W	IMPL332
	48	0.37	57.37	33.03	32.50	3.33	85.50	285	697.58	1186.97	64	1.90	✓	✓	✓		R+W	IMPL364
	49	0.36	59.73	33.27	32.50	3.33	85.89	285	856.64	1393.00	128	2.39	✓	✓	✓		R+W	IMPL3128
	68	0.46	39.85	28.70	36.25	20.00	64.23	333	441.30	797.11	32	0.96	✓	✓	✓		R+W	IMPL4$^{\neg}{}_{DF}$
	69	0.35	48.65	32.68	36.25	33.33	75.11	333	607.30	1096.94	32	1.72	✓	✓	✓	✓	R+W	IMPL432
	70	0.39	49.44	32.89	36.25	69.17	77.98	333	724.94	1233.54	64	1.87	✓	✓	✓	✓	R+W	IMPL464
	-	-	20.54	13.13	3.75	3.33	33.41	333	-	1237.47	128	-		✓			R+W	MEMCPY
	-	-	19.74	13.11	3.75	3.33	33.91	333	-	381.28	128	-		✓				MEMCPY$^{\neg}{}_{BMT}$

Tabelle 5.4 – Messergebnisse bei Problemgröße 2048



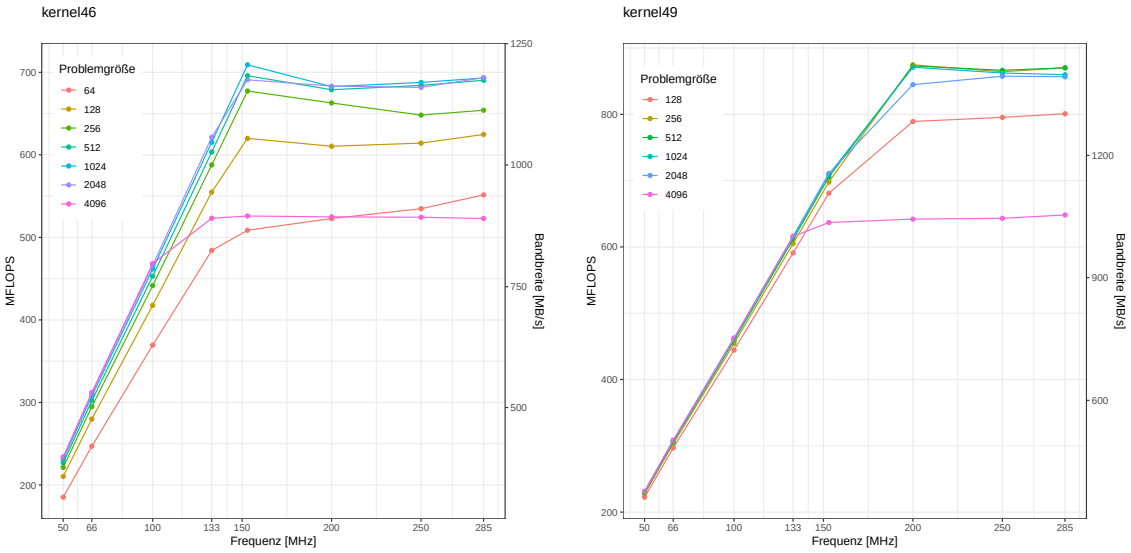
(a) IMPL0 **(b)** IMPL0$_{LP}$

Abbildung 5.7 – IMPL0 Frequenzabhängigkeit



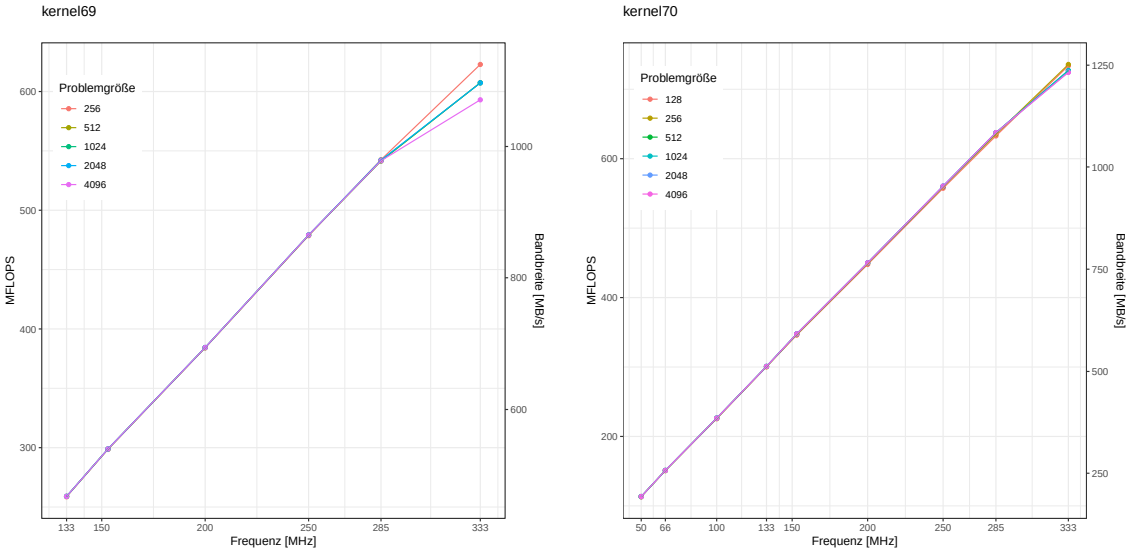
(a) IMPL1 $k_{size} = 32$ **(b)** IMPL2 $k_{size} = 64$

Abbildung 5.8 – IMPL1 / IMPL2 Frequenzabhängigkeit



(a) IMPL3 $k_{size} = 32$ **(b)** IMPL3 $k_{size} = 128$

Abbildung 5.9 – IMPL3 Frequenzabhängigkeit



(a) IMPL4 $k_{size} = 32$ **(b)** IMPL4 $k_{size} = 64$

Abbildung 5.10 – IMPL4 Frequenzabhängigkeit

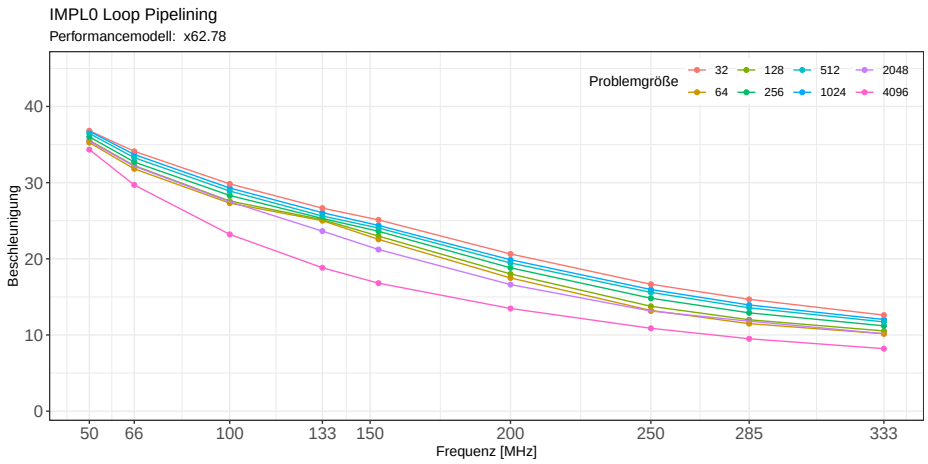


Abbildung 5.11 – Beschleunigung von IMPL0$_{LP}$ im Verhältnis zu IMPL0

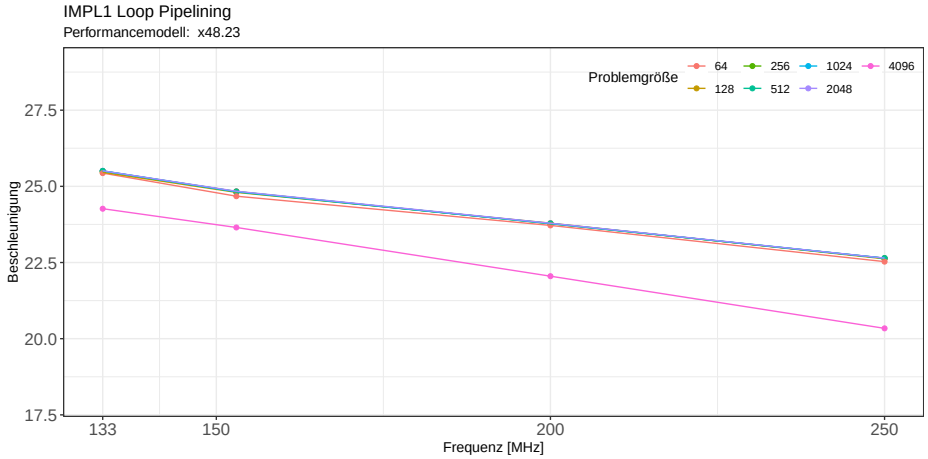


Abbildung 5.12 – Beschleunigung von IMPL1 im Verhältnis zu IMPL1$\neg_{LP}$

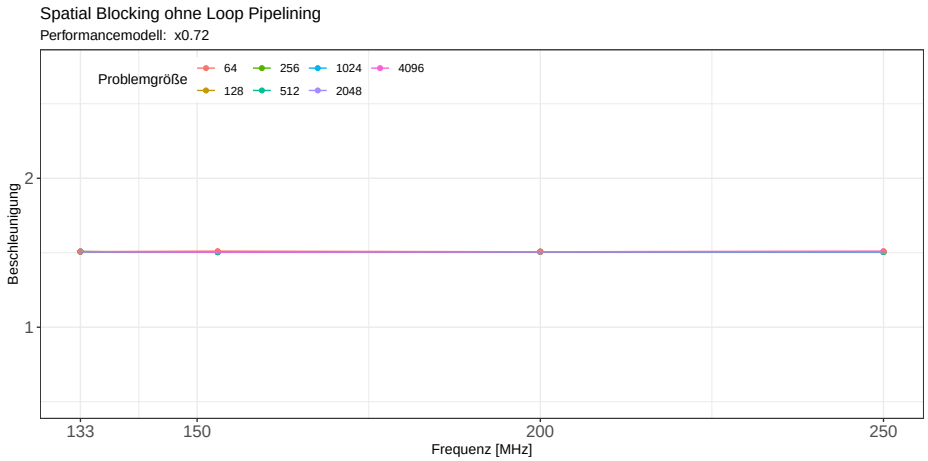


Abbildung 5.13 – Beschleunigung von IMPL1$\neg_{LP}$ im Verhältnis zu IMPL0

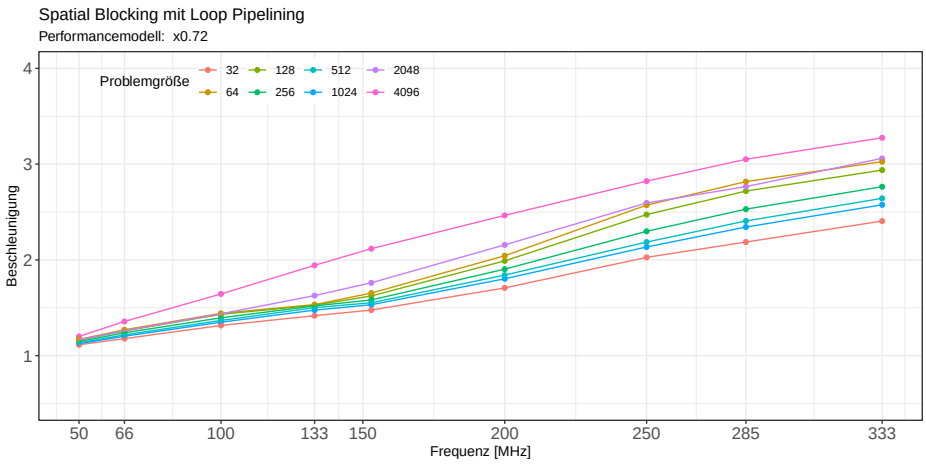


Abbildung 5.14 – Beschleunigung von IMPL1 im Verhältnis zu IMPL0$_{LP}$

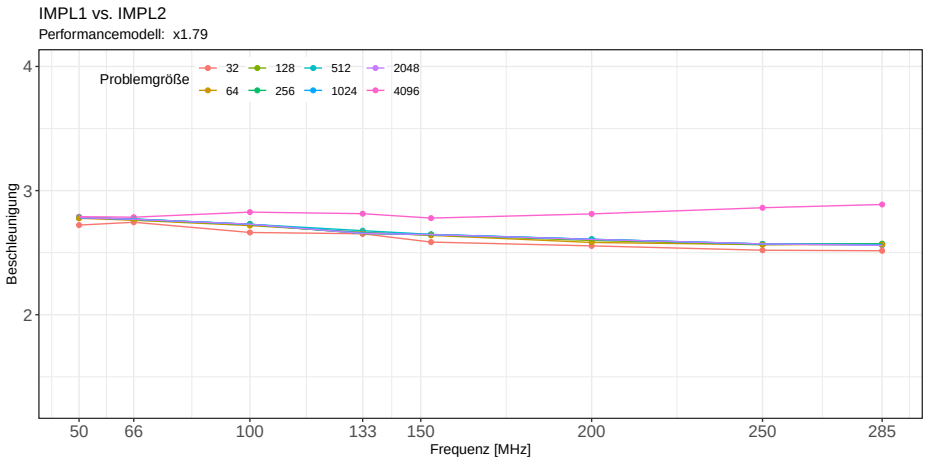


Abbildung 5.15 – Beschleunigung von IMPL2$_{32}$ im Verhältnis zu IMPL2$\neg_{AP}$

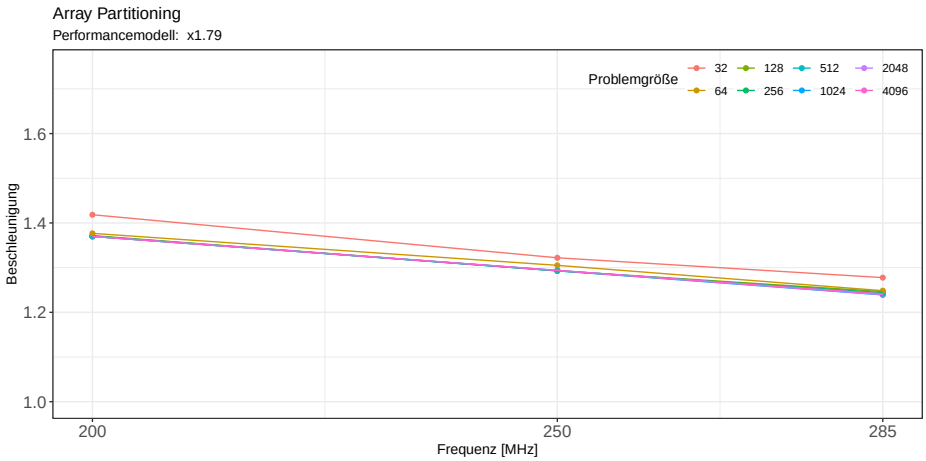


Abbildung 5.16 – Beschleunigung von IMPL2$_{32}$ im Verhältnis zu IMPL2$\neg_{AP}$

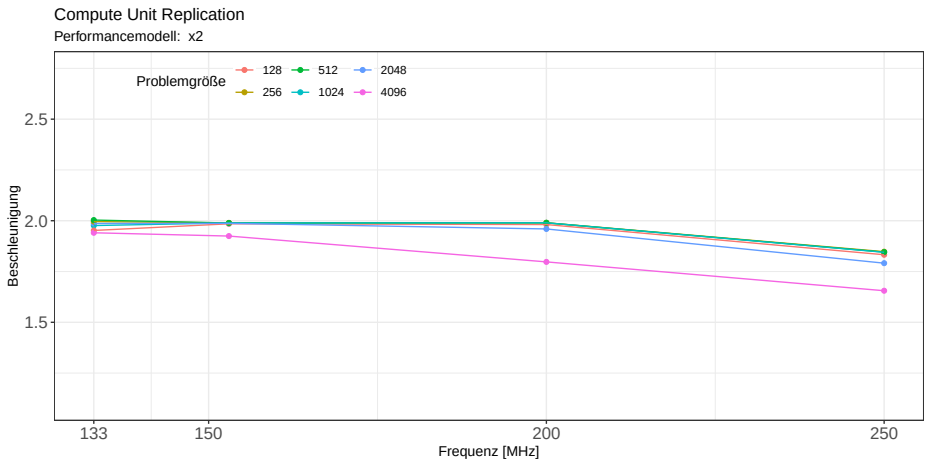


Abbildung 5.17 – Beschleunigung von IMPL2$_{CUR}$ im Verhältnis zu IMPL2$_{64}$

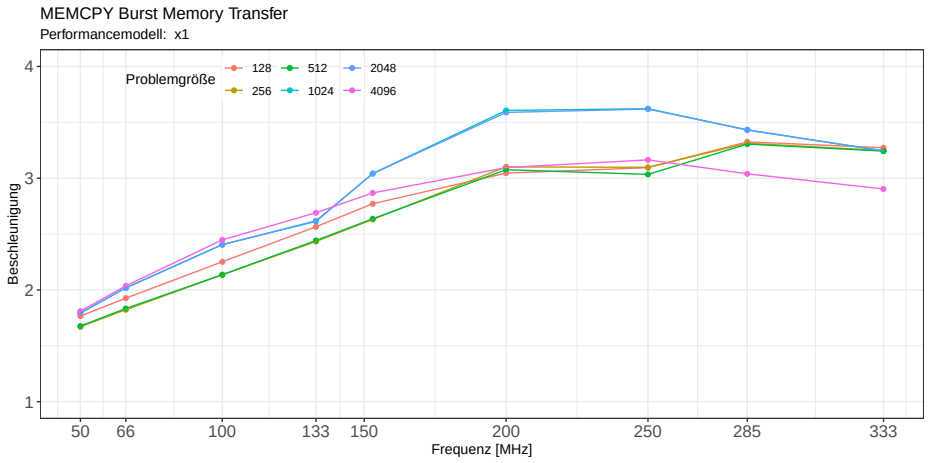


Abbildung 5.18 – Beschleunigung von MEMCPY im Verhältnis zu MEMCPY$^{\neg}{}_{BMT}$

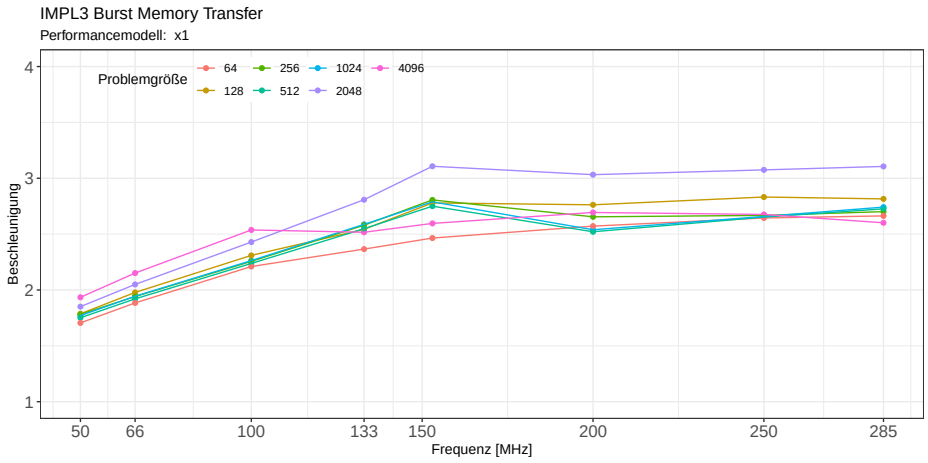


Abbildung 5.19 – Beschleunigung von IMPL3 im Verhältnis zu IMPL3$^{\neg}{}_{BMT}$

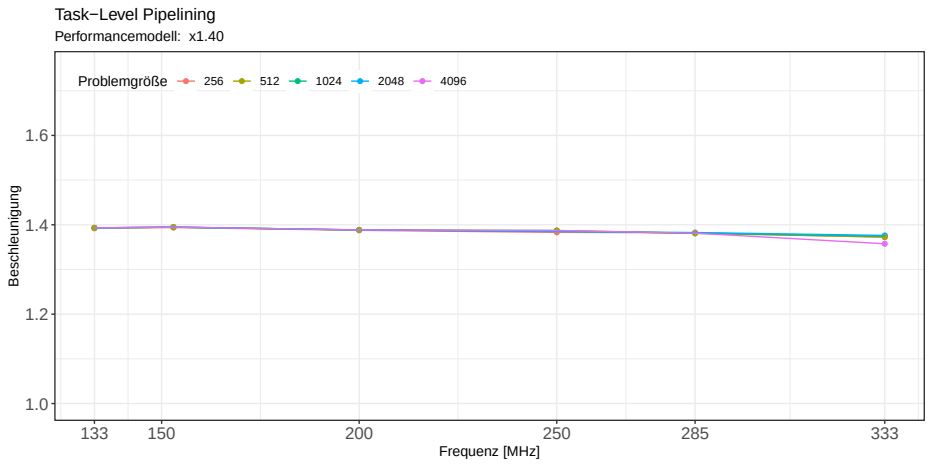


Abbildung 5.20 – Beschleunigung von IMPL4_{32} im Verhältnis zu $\text{IMPL4}_{\neg DF}$

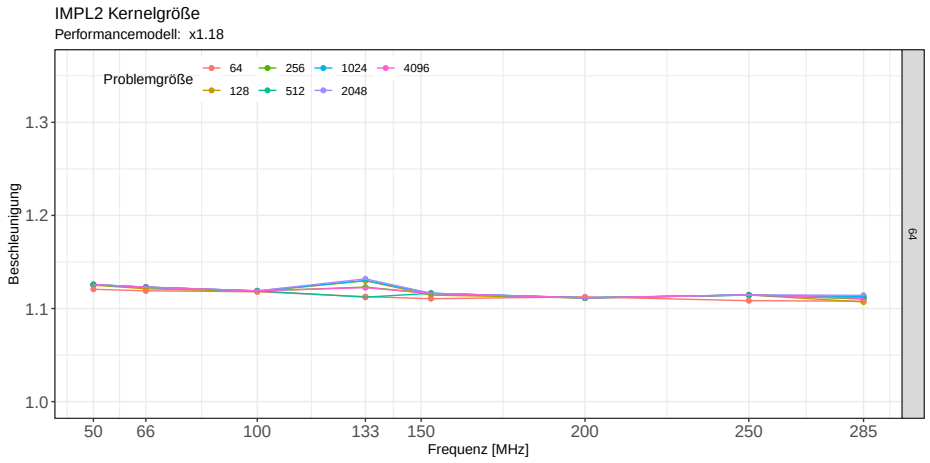


Abbildung 5.21 – Beschleunigung von IMPL2_{64} im Verhältnis zu IMPL2_{32}

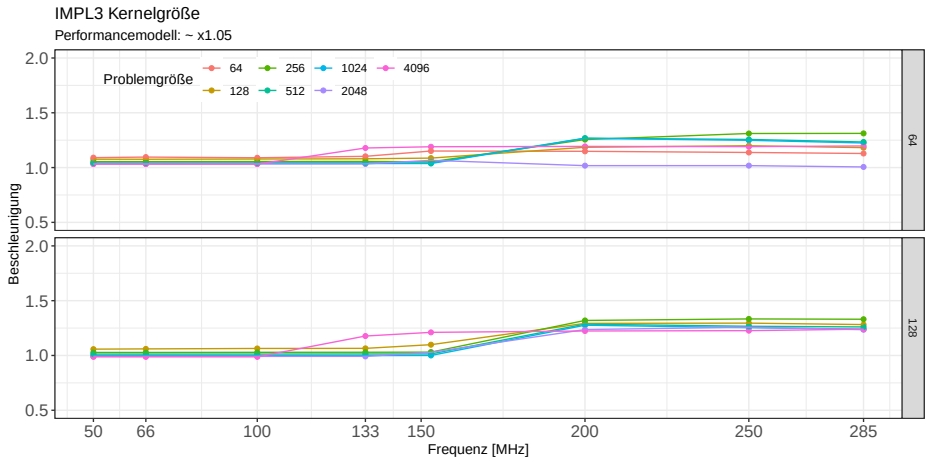


Abbildung 5.22 – Beschleunigung von IMPL3_{64} und IMPL3_{128} im Verhältnis zu IMPL3_{32}

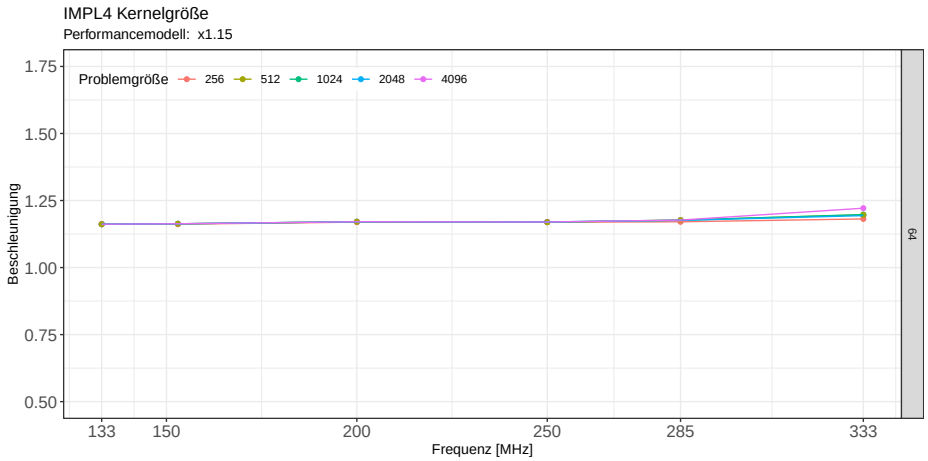


Abbildung 5.23 – Beschleunigung von IMPL4$_{64}$ im Verhältnis zu IMPL4$_{32}$

6 Interpretation

6.1 Gemessene Performance

6.1.1 Memory Benchmark

Eine bemerkenswerte Beobachtung ist, dass alle Messungen mit Limitierung durch den Datentransfer eine deutlich niedrigere Performance bei $N = 4096$ ab $f > 150\,\text{MHz}$ gemein haben. Der DRAM auf dem ZYBO verfügt über 1024 MB Arbeitsspeicher und ist in 8 Bänken organisiert – eine Bank pro 128 MB – und ab $N = 4096$ werden über 128 MB auf dem Global Memory allokiert. 150 MHz ist ausserdem die Frequenz der verwendeten Schnittstelle [Xil18e].

Eine genaue Analyse des Effekts erfordert eine Auseinandersetzung mit der Architektur und Konfiguration des verwendeten DRAM [Dre07]. Dies würde jedoch den Rahmen dieser Arbeit übersteigen.

MEMCPY mit BMT

Beim MEMCPY Kernel war ein linearer Anstieg der Bandbreite mit der FPGA-Frequenz bis zu der vom Hersteller angegebenen maximalen Bandbreite der verwendeten High-Performance-Schnittstelle $T_{max} = 1200\,\text{MB/s}$ zu erwarten.

Mit BMTs ist ein abruptes Ende des linearen Anstiegs zu erkennen. Das Maximum wurde zu $T_{max} = 1429\,\text{MB/s}$ bei $f = 200\,\text{MHz}$ und $N = 256$ gemessen. Daraus lässt sich eine obere Schranke der Performance der Anwendung aufstellen: $F_{max} = 893\,\text{MFLOP/s}$.

Die Limitierung durch die Bandbreite zeigte sich ab $f = 200\,\text{MHz}$ für kleinere Problemgrößen und für $N = 4096$ bereits ab $f = 153\,\text{MHz}$ (vgl. Abbildung 5.5).

Ein ähnliches Verhalten ist bei Memory-Bound-Kernels zu erwarten, welche BMTs zum Lesen und Schreiben verwenden.

MEMCPY ohne BMT

Die gemessene Bandbreite ohne BMTs liegt deutlich unter der Limitierung der High-Performance-Schnittstellen, was auf eine andere Ursache der Limitierung hinweist (vgl. Abbildung 5.6).

Eine Limitierung zwischen 350 MB/s und 450 MB/s ist bei Kernels zu erwarten, welche keine BMTs verwenden.

6.1.2 Loop Pipelining

Die Optimierung durch Loop Pipelining führt zu einer Steigerung der Performance und Erhöhung der verwendeten LUTs und FFs aufgrund der Pipeline-Transformation (vgl. Tabelle 5.4).

Die Beschleunigung durch Loop Pipelining nimmt mit zunehmender Frequenz ab (vgl. Abbildung 5.11, 5.12). Loop Pipelining führt zur Beschleunigung einer Schleife, wodurch mehr Speicheroperationen pro Zeit ausgeführt werden und Limitierungen durch Effekte des Datentransfers bei hohen Frequenzen auftreten.

6.1.3 Spatial Blocking

Spatial Blocking ohne Loop Pipelining führt zu einer Steigerung der Performance und Reduktion des Datenvolumens der Anwendung (vgl. Tabelle 5.4). Es zeigt sich keine Frequenzabhängigkeit der Beschleunigung, da der Durchsatz geringer ist, als die Limitierung durch den Speichertransfer (vgl. Abbildung 5.13).

Der Faktor der Beschleunigung durch Spatial Blocking mit Loop Pipelining nimmt mit steigender Frequenz zu (vgl. Abbildung 5.14). Spatial Blocking reduziert die Anzahl an Speicheroperationen pro Zeit, wodurch bei höheren Frequenzen die Limitierung durch den Speichertransfer geringer ausfällt, als ohne Spatial Blocking.

6.1.4 IMPL1 vs. IMPL2

IMPL2$^{\neg}{}_{AP}$ wird gegenüber IMPL1 durch zwei Faktoren beschleunigt. Zum einen wird das II_1 reduziert, wodurch die Beschleunigung mit der Frequenz abnehmen sollte. Zum anderen wird das Lesen der Daten durch BMTs ausgeführt, was zu einer Zunahme der Beschleunigung mit der Frequenz führen müsste. Die Beschleunigung nimmt mit der Frequenz für $N < 4096$ ab und für $N = 4096$ zu (vgl. Abbildung 5.15).

6.1.5 Array Partitioning

Der Faktor der Beschleunigung durch Array Partitioning nimmt mit der Frequenz ab (vgl. Abbildung 5.16). Array Partitioning führt zu einer Beschleunigung von $Loop1$ durch Reduktion von II_1.

6.1.6 Compute Unit Replication

Bei IMPL2$_{CUR}$ ist eine Verdopplung der Performance gegenüber IMPL2$_{64}$ zu erwarten, da die summierte Bandbreite der beiden verwendeten High-Performance-Schnittstellen noch unter der maximalen Performance des Arbeitsspeichers liegt (vgl. Tabelle 5.2).

Die Performance wird durch CUR bei $N < 4096$ und $f < 250\,\text{MHz}$ wie erwartet um einen Faktor 2 gesteigert. Ab $f = 200\,\text{MHz}$ reduziert sich der Beschleunigungsfaktor (vgl. Abbildung 5.17).

Mit CUR werden bereits über 99% der FPGA-Fläche benötigt. f_{max} beträgt bei IMPL2$_{64}$

285MHz und bei IMPL2$_{CUR}$ nur noch 250MHz, obwohl jeweils der gleiche Kernel instanziiert wurde.

6.1.7 Burst Memory Transfer

Durch BMTs wurde eine Beschleunigung um ca. einen Faktor 3 gemessen, welcher bei niedrigeren Frequenzen geringer ausfällt. Der Effekt durch atomare Speicheroperationen wird im Performancemodell nicht berücksichtigt (vgl. Abbildung 5.19, 5.18).

Bemerkenswert ist, dass bei IMPL3$\neg_{BMT}$ sowohl weniger Daten berechnet, als auch gespeichert werden. Trotzdem ist die Performance aufgrund des atomaren Speichertransfers geringer, als bei IMPL3. Auch bei IMPL4 führt die Optimierung durch BMTs zu einer höheren Performance im Vergleich zu IMPL2, obwohl die LPU bei IMPL2 geringer ist.

6.1.8 Loop Unrolling und Vektorisierung

Loop Unrolling und Vektorisierung führen zu einer Verbreiterung des Datenpfades einer Anwendung und wurden aufgrund der 64-bit-Schnittstelle zum Global Memory und dem 64-bit-Datentyp vor den Messungen als ungeeignet auf dem ZYBO identifiziert. Das II würde sich um den Unroll-Factor bzw. der Dimension des Vektors erhöhen.

Es ist davon auszugehen, dass beide Optimierungen auf dem ZYBO zu keiner Performancesteigerung führen. Auf einer Hardware mit höherer Busbreite zum Global Memory ist davon auszugehen, dass die Performance profitiert.

6.1.9 Kernelgröße

Eine höhere Kernelgröße verbessert die Performance von Kernels mit Spatial Blocking durch zwei Faktoren: zum einen erhöht sich der Trip Count, was die Iteration Latency versteckt und zum anderen verringert sich das Verhältnis von redundanten Speicherzugriffen zu notwendigen Speicherzugriffen, wodurch die Bandbreite zwischen Kernel und Global Memory besser genutzt werden kann. Beides spiegelt sich sowohl in der gemessenen Performance, als auch in der LPU der untersuchten Kernels wieder (vgl. Abbildung 5.21 5.22 5.23).

6.1.10 Zusammenfassung

Die gemessene Beschleunigung durch Optimierungen des Speichertransfers steigt mit der Frequenz. Führt eine Optimierung zur Beschleunigung der Schleife, sinkt die gemesse Beschleunigung mit der Frequenz.

Spatial Blocking und BMTs optimieren den Speichertransfer. Dadurch steigt die Beschleunigung mit der Frequenz.

Loop Pipelining und Array Partitioning führen zu einer Beschleunigung der Schleifen, wodurch mehr Speicheroperationen pro Zeit ausgeführt werden. Dadurch sinkt die Beschleunigung mit steigender Frequenz.

Die Kernelgröße sollte maximal gewählt werden, falls eine Optimierung durch CUR nicht durch die benötigten Hardwareressourcen verhindert wird.

6.2 Vergleich mit dem Performance-Modell

	$f = 250$MHz			$f = 285$MHz		
Implementierung	%	MFLOPS	T [MB/s]	%	MFLOPS	T [MB/s]
IMPL1	74	125 /170	228 /308	71	138 /194	249 /351
IMPL2$\neg_{AP}$	98	251 /257	453 /464	97	285 /293	514 /529
IMPL2$_{16}$	75	239 /318	485 /643	-	- /362	- /734
IMPL2$_{32}$	68	324 /479	586 /865	65	353 /546	637 /986
IMPL2$_{64}$	64	361 /564	615 /960	61	393 /643	667 /1094
IMPL2$_{CUR}$	63	647 /1028	1101 /1920	-	- /643	- /2188

Tabelle 6.1 – Performancemodell für IMPL1 und IMPL2 bei N=2048, $f = 250$MHz und $f = 285$MHz. gemessen / erwartet

	$f = 150$MHz			$f = 200$MHz		
Implementierung	%	MFLOPS	T [MB/s]	%	MFLOPS	T [MB/s]
IMPL3$_{32}$	99	691 /701	1176 /1124	78	683 /935	1163 /1499
IMPL3$_{64}$	102	737 /725	1217 /1162	74	695 /966	1149 /1549
IMPL3$_{128}$	97	711 /737	1156 /1181	87	845 /982	1374 /1575

Tabelle 6.2 – Performancemodell für IMPL3 bei N=2048 und $f = 150$MHz und $f = 200$MHz. gemessen / erwartet

	$f = 333$MHz		
Implementierung	%	MFLOPS	T [MB/s]
IMPL4$_{32}$	90	607 /671	1097 /1212
IMPL4$_{64}$	93	725 /774	1234 /1317
IMPL4$\neg_{DF}$	92	441 /477	797 /862

Tabelle 6.3 – Performancemodell für IMPL4 bei N=2048 und $f = 333$MHz. gemessen / erwartet

6.2.1 IMPL1 und IMPL2

Das Performancemodell weicht bei IMPL1 und IMPL2 deutlich von den Messungen ab. Auffällig ist, dass das Performancemodell bei IMPL2$\neg_{AP}$ eine gute Übereinstimmung zeigt, obwohl das Speichern nicht durch BMTs ausgeführt wird.

Der gemessene Faktor der Beschleunigung durch BMTs entspricht ungefähr dem II von *Loop*2, wodurch die Genauigkeit begründet werden kann, da sich die Faktoren aufheben.

Wird *Loop*2 durch Array Partitioning beschleunigt, weicht das Performancemodell ab (vgl. Tabelle 6.1).

6.2.2 IMPL3

Bei IMPL3 ist die maximale Performance zwischen $f = 150\text{MHz}$ und $f = 200\text{MHz}$ zu erwarten, da das Performancemodell für Frequenzen größer $f = 200\text{MHz}$ $T > T_{max}$ voraussagt (vgl. Tabelle 6.2). Zwischen $f = 150\text{MHz}$ und $f = 200\text{MHz}$ liegt auf f_{opt} bei IMPL3, da hier die Energieeffizienz bei maximaler Performance am höchsten ist.

6.2.3 IMPL4

Eine Abweichung vom Performancemodell ist bei IMPL4_{32} und IMPL4_{64} ab $f = 333\text{MHz}$ zu erwarten, da das Performancemodell $T > T_{max}$ voraussagt und wurde ab $f = 333\text{MHz}$ wurde gemessen (vgl. Tabelle 6.3, Abbildung 5.10).

6.2.4 Bewertung des Performancemodells

Das Performancemodell zeigt immer dann eine gute Übereinstimmung mit der gemessenen Performance, wenn die vorausgesagte benötigte Bandbreite unter der Performance der High-Performance-Schnittstelle liegt und BMTs für den Datentransfer zum Global Memory synthetisiert wurden. Es kann gefolgert werden, dass besonders bei atomaren Speicheroperationen Effekte auftreten, welche nicht im Performancemodell berücksichtigt werden.

7 Fazit und Ausblick

Im Rahmen dieser Arbeit habe ich viel über die Funktionsweise von Hardware, insbesondere der von FPGAs, digitalem Speicher und Speicherzugriffen, die Programmierung von Hardware, Parallelisierung und Optimierung von Algorithmen und die systematische Vorgehensweise bei der Performanceanalyse und Optimierung gelernt.

Die Lernkurve dieses Themas ist relativ steil – es sind viele Schritte notwendig, bevor die erste Anwendung auf der PL des ZYBO ausgeführt werden kann. Jeder dieser Schritte erfordert das Studium diverser technischer Dokumentationen, deren Zielgruppe zumeist nicht der durchschnittliche Softwareentwickler ist.

Zumindest unter Verwendung von Vivado HLS hat sich kein Vorteil von OpenCL zur HLS gegenüber der herkömmlichen C-Synthese herausgestellt. Besonders die Einschränkung auf die automatisierte Schnittstellensynthese und die reduzierten Optimierungs- und Konfigurierungsmöglichkeiten bei der Verwendung von OpenCL mit Vivado HLS haben sich als klarer Nachteil von OpenCL zur HLS herausgestellt. Mit der Weiterentwicklung von Xilinx *SDAccell* und Intels *Quartus* Software ist jedoch das zukünftige Potential von OpenCL auf anderen FPGAs deutlich erkennbar.

Das OpenCL Execution Model mit Work-Groups und Work-Items stellte sich als ungeeignet für die Anwendung auf den 2D 5-Punkt Stencil auf FPGAs dar. Single-Task Kernels waren – wie auch in anderen Arbeiten erwähnt – deutlich performanter. Der Einstieg in die FPGA-Programmierung wurde durch die Verwendung von OpenCL, im Vergleich zur C-Synthese mit Vivado HLS, erleichtert. Die C-Synthese bietet zwar mehr Möglichkeiten, erfordert aber auch die manuelle Konfiguration mehrerer Komponenten.

Als wichtigste Optimierung der Performance des Stencil-Kernels konnten Datenwiederverwendung, Loop Pipelining und BMTs identifiziert werden, wobei die Optimierung durch Loop Pipelining am einfachsten zu implementieren ist. Das von Xilinx empfohlene Load-Store-Compute Pattern stellte sich als guter Kompromiss zwischen erreichter Performance und Entwicklungsdauer der Anwendung heraus.

Eine Implementierung des Stencil-Algorithmus mehrerer paralleler Iterationen mit Schieberegistern und Temporal Blocking wie beispielsweise in [Zoh18] vorgeschlagen ist aufgrund der begrenzten Hardwareressourcen auf dem ZYBO schwer zu realisieren. Die Erforschung einer ähnlichen Implementierung oder einer anderen Anwendung auf einem FPGA könnte an diese Arbeit anknüpfen.

Es bleibt mit Spannung zu verfolgen, ob sich FPGAs durch weitere technische oder konzeptionelle Verbesserungen im High-Performance-Bereich durchsetzen können.

8 Anhang

8.1 Abkürzungsverzeichnis

ALU Arithmetic Logical Unit

ASIC Application-specific Integrated Circuit

BSP Board Support Package

BRAM Block RAM

BMT Burst Memory Transfer

CLB Configurable Logic Block

CUR Compute Unit Replication

CU Compute Unit

CNN Convolutional Neural Network

DSP Digital Signal Processor

FF Flip-Flop

FPGA Field-programmable Gate Array

HPC High-Performance Computing

HDL Hardware Description Language

HLS High-Level Synthesis

II Initiation Interval

LL Loop Latency

LPU Latency per Update

LUT Look-up Table

OpenCL Open Computing Language

PE Processing Element

PL Programmable Logic

PS Processing System

ZYBO Zybo Z7-10

8.2 Listings

```
#define KSIZE 128

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void stencil(__global double *__restrict out,
             __constant double *__restrict src,
             int out_size)
{
        int block_col = get_group_id(0);

        for (int row = 0; row < out_size; row++) {
            __attribute__((xcl_pipeline_loop(1)))
            for (int col = 0; col < KSIZE; col++) {
                // compute address
                int addr = row * out_size + block_col * KSIZE + col;
                // copy
                // if(col < 0) // this prevents burst mode
                out[addr] = src[addr];
}}}
```

Listing 8.1 – MEMCPY

```
#define HALO 2

__kernel __attribute__((reqd_work_group_size(KSIZE, KSIZE, 1)))
void stencil(__global double *__restrict out,
             __constant double *__restrict src)
{
        int src_size = get_global_size(0) + HALO;
        int out_size = get_global_size(0);

        int global_x = get_global_id(0), global_y = get_global_id(1);

        out[global_y * out_size + global_x] =
                    0.2 * (src[(global_y + 1) * src_size + global_x]
                         + src[(global_y + 1) * src_size + global_x + 1]
                         + src[(global_y + 1) * src_size + global_x + 2]
                         + src[(global_y)     * src_size + global_x + 1]
                         + src[(global_y + 2) * src_size + global_x + 1]);
}
```

Listing 8.2 – IMPL0

```
__attribute__((reqd_work_group_size(KSIZE, KSIZE, 1)))
__kernel
void stencil(__global double *__restrict out,
             __constant double *__restrict src)
{
    __local double buff_in[KSIZE + HALO][KSIZE + HALO];

    __attribute__((xcl_pipeline_workitems)) {
    int src_size = get_global_size(0) + HALO;
    int out_size = get_global_size(0);

    int global_x = get_global_id(0);
    int global_y = get_global_id(1);

    int local_x = get_local_id(0);
    int local_y = get_local_id(1);

    int group_x = get_group_id(0);
    int group_y = get_group_id(1);

    int block_x = KSIZE * group_x;
    int block_y = KSIZE * group_y;

    // load
    if(local_x < BSIZE / 2 && local_y < BSIZE / 2) {
        buff_in[2*local_y][2*local_x]     =
            src[(block_y + 2 * local_y) * src_size + block_x + 2 * local_x];
        buff_in[2*local_y][2*local_x+1]   =
            src[(block_y + 2 * local_y) * src_size + block_x + 2 * local_x + 1];
        buff_in[2*local_y+1][2*local_x]   =
            src[(block_y + 2 * local_y + 1) * src_size + block_x + 2 * local_x];
        buff_in[2*local_y+1][2*local_x+1] =
            src[(block_y + 2 * local_y + 1) * src_size + block_x + 2 * local_x + 1];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    out[global_y * out_size + global_x] =
                    0.2 * (buff_in[local_y + 1][local_x    ]
                         + buff_in[local_y + 1][local_x + 1]
                         + buff_in[local_y + 1][local_x + 2]
                         + buff_in[local_y    ][local_x + 1]
                         + buff_in[local_y + 2][local_x + 1]);
}   }
```

Listing 8.3 – IMPL1

```
__attribute__((reqd_work_group_size(BSIZE, BSIZE, 1)))
__kernel
void stencil(__global double *__restrict out,
             __constant double *__restrict src)
{
    __local double
    buff_in[BSIZE][BSIZE]__attribute__((xcl_array_partition(cyclic, 2, 0)));

    __attribute__((xcl_pipeline_workitems)) {
    int out_size = get_global_size(0);
    int src_size = get_global_size(0) + HALO;

    int global_x = get_global_id(0);
    int global_y = get_global_id(1);

    int local_x = get_local_id(0);
    int local_y = get_local_id(1);

    int block_x = KSIZE * get_group_id(0);
    int block_y = KSIZE * get_group_id(1);

    // load
    int src_addr = (block_y + local_y) * src_size + block_x + local_x;

    buff_in[local_y][local_x] = src[src_addr];

    barrier(CLK_LOCAL_MEM_FENCE);

    // return if on block edge
    if(local_x == 0 || local_y == 0 || local_x == BSIZE - 1 || local_y == BSIZE - 1)
        return;

    // compute and store
    int out_addr = (block_y + local_y - 1) * out_size + block_x + local_x - 1;

    out[out_addr] = 0.2 * (buff_in[local_y    ][local_x    ]
                         + buff_in[local_y    ][local_x + 1]
                         + buff_in[local_y    ][local_x - 1]
                         + buff_in[local_y - 1][local_x    ]
                         + buff_in[local_y + 1][local_x    ]);
}   }
```

Listing 8.4 – IMPL2

```
#define SREG_SIZE (2 * BSIZE + 2)

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void stencil(__global double *__restrict out,
             __constant double *__restrict src,
             int size)
{
  double sreg[SREG_SIZE]__attribute__((xcl_array_partition(complete, 0)));
  int block_col = get_group_id(0);

  for (int row = 0; row < size; row++) {
    for (int col = 0; col < BSIZE; col++) {
      // compute address
      int addr = row * size + block_col * KSIZE + col;

      // load 1
      sreg[SREG_SIZE - 1] = src[addr];

      // compute & store (row=0, row=1, col=0, col=1 will be invalid)
      // if(row > 1 && col > 1) // this prevents burst mode
      out[addr] = 0.2 * (
          sreg[0] +
          sreg[BSIZE - 1] +
          sreg[BSIZE    ] +
          sreg[BSIZE + 1] +
          sreg[2 * BSIZE]);

      // shift registers
      for (int j = 0; j < SREG_SIZE - 1; j++) {
          sreg[j] = sreg[j + 1];
}}}         }
```

Listing 8.5 – IMPL3

```
__kernel
__attribute__ ((xcl_dataflow))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
void stencil(__global double *__restrict out,
             __constant double *__restrict src,
             int out_size)
{
    double buff_in[BSIZE][BSIZE]__attribute__((xcl_array_partition(cyclic, 2, 0)));
    double buff_out[KSIZE * KSIZE];

    const int src_size = out_size + HALO;

    int block_col = get_group_id(0);
    int block_row = get_group_id(1);

    // load
    for (int row = 0; row < BSIZE; row++) {
        __attribute__ ((xcl_pipeline_loop(1)))
        for (int col = 0; col < BSIZE; col++) {
            int src_addr = (block_row * KSIZE + row) * src_size
                           + block_col * KSIZE + col;

            buff_in[row][col] = src[src_addr];
    }}

    // compute
    for (int row = 0; row < KSIZE; row++) {
        __attribute__ ((xcl_pipeline_loop(1)))
        for (int col = 0; col < KSIZE; col++) {
            buff_out[row * KSIZE + col] = 0.2 *
                            (buff_in[row + 1][col]
                           + buff_in[row + 1][col + 1]
                           + buff_in[row + 1][col + 2]
                           + buff_in[row    ][col + 1]
                           + buff_in[row + 2][col + 1]);
    }}

    // store
    for (int row = 0; row < KSIZE; row++) {
        __attribute__ ((xcl_pipeline_loop(1)))
        for (int col = 0; col < KSIZE; col++) {
            int out_addr = (block_row * KSIZE + row) * out_size
                           + block_col * KSIZE + col;

            out[out_addr] = buff_out[row * KSIZE + col];
}   }  }
```

Listing 8.6 – IMPL4

Listings

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Bre20] Breiter, S.A.: *Performanceanalysis of Stencil-Kernels on FPGAs.* https://github.com/h4gb4r7/performanceanalysis_stencil_kernels_zybo, Januar 2020

[CNK+13] Czajkowski, Tomasz S. ; Neto, David ; Kinsner, Michael ; Aydonat, Utku ; Wong, Jason ; Denisenko, Dmitry ; Yiannacouras, Peter ; Freeman, John ; Singh, Deshanand P. ; Brown, Stephen D.: OpenCL for FPGAs: Prototyping a Compiler, 2013

[Dre07] Drepper, Ulrich: What every programmer should know about memory. In: *Red Hat, Inc* 11 (2007), S. 2007

[EWH17] Endo, Tsukasa ; Waidyasooriya, Hasitha M. ; Hariyama, Masanori: Automatic optimization of OpenCL-based stencil codes for FPGAs. In: *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* Springer, 2017, S. 75–89

[JZ16] Jia, Qi ; Zhou, Huiyang: Tuning stencil codes in OpenCL for FPGAs. In: *2016 IEEE 34th International Conference on Computer Design (ICCD)* IEEE, 2016, S. 249–256

[Khr19] Khronos OpenCL Working Group: *The OpenCL Specification.* Version 2.2, Juli 2019. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

[MM15] Muranushi, Takayuki ; Makino, Junichiro: Optimal temporal blocking for stencil computation. In: *ICCS*, 2015, S. 1303–1312

[P+12] Pouchet, Louis-Noël u. a.: Polybench: The polyhedral benchmark suite. In: *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012)

[PS10] Prasanna Sundararajan, Xilinx: *High Performance Computing Using FPGAs.* https://www.xilinx.com/support/documentation/wite_papers/wp375_HPC_Using_FPGAs.pdf. Version: 2010

[STHW14] Stengel, Holger ; Treibig, Jan ; Hager, Georg ; Wellein, Gerhard: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In: *CoRR* abs/1410.5010 (2014). http://arxiv.org/abs/1410.5010

[VB16] Vanderbauwhede, Wim ; Benkrid, Khaled: *High-Performance Computing Using FPGAs.* 1st. Springer Publishing Company, Incorporated, 2016. – ISBN 1493943103

[VHKF16] Verma, Anshuman ; Helal, Ahmed E. ; Krommydas, Konstantinos ; Feng, Wu chun: Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs, 2016

[WHU17] Waidyasooriya, Hasitha ; Hariyama, Masanori ; Uchiyama, Kunio: *Design of FPGA-based computing systems with openCL*. 2017. http://dx.doi.org/10.1007/978-3-319-68161-0. http://dx.doi.org/10.1007/978-3-319-68161-0. – ISBN 3319681605

[WL17] Wang, Shuo ; Liang, Yun: A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In: *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, S. 1–6

[Xil14] Xilinx (Hrsg.): *Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis UG1145*. v2014.4. Xilinx, November 2014. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug1145-sdk-system-performance.pdf

[Xil16a] Xilinx (Hrsg.): *SDAccel Development Environment Methodology Guide Performance Optimization UG1207*. v2.0. Xilinx, August 2016. https://www.xilinx.com/support/documentation/sw_manuals/ug1207-sdaccel-performance-optimization.pdf

[Xil16b] Xilinx (Hrsg.): *Vivado Design Suite User Guide High-Level Synthesis UG902*. v2016.1. Xilinx, April 2016. https://xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf

[Xil17a] Xilinx (Hrsg.): *Vivado HLS Optimization UG1270*. v2017.4. Xilinx, Dezember 2017. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf

[Xil17b] Xilinx (Hrsg.): *Zynq-7000 All Programmable SoC: Embedded Design Tutorial UG1165*. v2017.3. Xilinx, November 2017. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1165-zynq-embedded-design-tutorial.pdf

[Xil18a] Xilinx (Hrsg.): *SDSoC Environment Profiling and Optimization Guide UG1235*. v2017.4. Xilinx, Januar 2018. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1235-sdsoc-optimization-guide.pdf

[Xil18b] Xilinx (Hrsg.): *Xilinx Standalone Library Documentation UG643*. v2018.3. Xilinx, Dezember 2018. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/oslib_rm.pdf

[Xil18c] Xilinx (Hrsg.): *Zynq-7000 SoC Data Sheet: Overview DS190*. v1.11.1. Xilinx, Juli 2018. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[Xil18d] Xilinx (Hrsg.): *Zynq-7000 SoC Technical Reference Manual UG585*. v1.12.2. Xilinx, Juli 2018. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[Xil18e] Xilinx (Hrsg.): *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics Product Specification DS187*. v1.20.1. Xilinx, Juli 2018. https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf

[Xil19a] Xilinx (Hrsg.): *SDAccel Methodology Guide UG1346*. v2019.1. Xilinx, Mai 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1346-sdaccel-methodology-guide.pdf

[Xil19b] Xilinx (Hrsg.): *SDAccel Programmers Guide UG1277*. v2019.1. Xilinx, Mai 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1277-sdaccel-programmers-guide.pdf

[ZMSM16] Zohouri, Hamid R. ; Maruyama, Naoya ; Smith, Aaron ; Matsuoka, Satoshi: Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* Bd. 2016, IEEE, November 2016

[Zoh18] Zohouri, Hamid R.: *High Performance Computing with FPGAs and OpenCL*, Tokyo Institute of Technology, Ph.D. Dissertation, 2018. https://arxiv.org/pdf/1810.09773

[ZPM18] Zohouri, Hamid R. ; Podobas, Artur ; Matsuoka, Satoshi: Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* ACM, 2018, S. 153–162