TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vergleichender Einsatz von High-Level Synthese auf Datenanalyse-Algorithmen für Leistungsmessgeräte

Masterarbeit von
Johannes Schaub

Januar 2018

Erstgutachter:
Prof. Dr. Andreas Koch

Zweitgutachter:
Dr.-Ing. Andreas Engel
M.Sc. Jens Korinth

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

Vergleichender Einsatz von High-Level Synthese auf Datenanalyse-Algorithmen für Leistungsmessgeräte

Masterarbeit von Johannes Schaub
Eingereicht am 30.01.2018
Erstgutachter: Prof. Dr. Andreas Koch
Zweitgutachter: Dr.-Ing. Andreas Engel, M.Sc. Jens Korinth

Eigenständigkeitserklärung

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

Darmstadt, 30. Januar 2018

(Johannes Schaub)

Danksagungen

Die vorliegende Masterarbeit hätte nicht ohne die Unterstützung durch zahlreiche Personen fertiggestellt werden können.

Danken möchte ich zunächst meinen beiden Betreuern an der Technischen Universität Darmstadt, Dr.-Ing. Andreas Engel und M.Sc. Jens Korinth, die mehrere Zwischenversionen der Masterarbeit gegengelesen und wertvolle Verbesserungsvorschläge gegeben hatten. Danken möchte ich auch dem Leiter des Fachgebiets Prof. Dr. Andreas Koch für die Vergabe des Themas sowie der guten Ausbildung durch seine Vorlesung „Fortgeschrittener Compilerbau“. Denen bei der Zwischenpräsentation anwesenden Mitarbeitern des Fachgebiets gebührt mein Dank für die anregende Diskussion und aufschlussreichen Fragen.

Des Weiteren möchte ich mich bei dem Geschäftsführer des Leistungsmessgeräteherstellers, Herrn Dr. Conrad Zimmer, meinem Betreuer im Unternehmen, Herrn M.Sc. Tobias Vollberg, sowie allen anderen Mitarbeitern der Entwicklungsabteilung bedanken, die diese Arbeit möglich gemacht und mich mit zahlreichen fachlichen Diskussionen und Hinweisen unterstützt haben.

Kurzfassung

Die steigende Komplexität in der Hardwareentwicklung digitaler Systeme stellt Entwickler vor zunehmend schwierigen Herausforderungen. Das Arbeiten auf der niedrigen Abstraktionsebene einer HDL (Hardware Description Language) erschwert das Fokussieren auf funktionale Systemeigenschaften. Mit der zunehmend eingesetzten HLS (High-Level Synthese) können Hardwarebeschreibungen für digitale Schaltkreise mithilfe von Hochsprachen wie C++ erstellt werden. In wissenschaftlichen Arbeiten wurde untersucht, ob mit dem aktuellen Stand dieser Technik vergleichbar leistungsstarke und ressourceneffiziente Schaltkreise entworfen werden können [27][14][49]. Da einige HLS-Compiler auf existierenden Softwarecompiler-Bibliotheken basieren [9][8][28][34], wurden Verbesserungen an bestehenden Compiler-Optimierungen vorgeschlagen [15], die üblicherweise heuristisch für Softwareprogramme optimiert sind und auf die HLS teilweise negative Auswirkungen haben.

In dieser Masterarbeit wird der Frage nachgegangen, ob und wie gut mit dem Vivado HLS[43] Werkzeug von Xilinx C++-Quelltext verarbeiten kann, der weiterhin mit klassischen Prozessoren verwendet und ggf. auch im geringen Umfang weiterentwickelt bzw. geändert wird. Zu diesem Zweck werden gemeinsam genutzte Implementierungen von Algorithmen aus Bereichen der Leistungsmessung verwendet. Für eine Bewertung der Ergebnisqualitäten werden handgeschriebene VHDL (Very High Speed Integrated Circuit Hardware Description Language) und CPU (Central Processing Unit)-spezifische C++-Lösungen vorgestellt und deren Performanz sowie Ressourcenbedarf mit denen der gemeinsam genutzten C++-Implementierungen verglichen. Zusätzlich werden die Auswirkungen von geringfügigen Änderungen am Quelltext auf die Performanz und den Ressourcenbedarf untersucht.

Die Ergebnisse dieser Masterarbeit zeigen, dass minimale und lokale Änderungen am C++-Quelltext oft aufgrund zu viel beanspruchter Hardwareressourcen zu nicht implementierbaren Synthesen führen. Außerdem werden prinzipielle Probleme bei der direkten Verwendung von Vivado HLS auf einem sich ändernden Quelltext für die Integrationsfähigkeit in ein FPGA (Field Programmable Gate Array)-System konstatiert, da die stabilen Schnittstellen zwischen verschiedenen digitalen Schaltkreisen effektive Optimierungen verhindern, da diese sonst zu inkompatiblen Schnittstellen führen könnten.

Inhaltsverzeichnis

1. Motivation, Ziele und Aufbau

1.1. Motivation

Durch die steigende Komplexität von SoCs (System on Chips) zeichnet sich zurzeit ein Problem ab: Die Komplexität dieser Systeme wächst schneller als die Produktivität ihrer Entwickler. Dieser Umstand wird auch als *Design Productivity Gap* [27] bezeichnet. Der Grund für die stagnierende Produktivität ist die niedrige Abstraktion der eingesetzten HDL (Hardware Description Language). HLS (High-Level Synthese) hebt die Abstraktionsebene auf die funktional/algorithmische Ebene und ermöglicht den Entwicklern ein direktes Programmieren des gewünschten Systemverhaltens. Die HLS erhält als Eingabe ein Programm in einer Hochsprache (wie C++, im Folgenden exemplarisch verwendet) und erzeugt als Ausgabe eine Hardwarebeschreibung (wie VHDL (Very High Speed Integrated Circuit Hardware Description Language)), welche sich für die Synthese für FPGAs (Field Programmable Gate Arrays) oder ASICs (Application-Specific Integrated Circuits) eignet. Forschungen zur HLS gehen bis in die 1980er Jahre zurück [26]. Produktiv eingesetzt wird die HLS aber erst seit einigen Jahren [5].

Unabhängig von dieser Entwicklung gibt es ein starkes Interesse aus der Industrie, die Ausführung einzelner Abschnitte eines C++ Programms mit Hilfe spezieller Hardware zu beschleunigen. Hierfür eignen sich insbesondere allgemein programmierbare GPGPUs (General-Purpose Graphics Processing Units) und FPGAs. GPGPUs haben Vorteile bei Fließkommaberechnungen [33][21], haben aber eine deutlich höhere Leistungsaufnahme und Wärmeentwicklung [6][25]. Die Anschaffungskosten von Grafikkarten sind niedriger als die von FPGAs[6]. Für Systeme, die aber ohnehin bereits Berechnungen auf einem FPGA vornehmen, kann es sich lohnen, die bisher auf dem Hauptprozessor ausgeführten Berechnungen teilweise auf dieses FPGA zu verlagern. Die automatische Synthese von HDL-Code für FPGAs eröffnet diesen Bereich auch für Software-Entwickler.

1.2. Ziele der Arbeit

In dieser Masterarbeit wird daher untersucht, inwieweit existierende C++-Quelltexte mit Hilfe der HLS automatisch nach VHDL konvertiert werden können. Damit separat vorhandene Quelltexte für Software- und HLS-Implementierungen vermieden werden, soll der zugrundeliegende C++-Quelltext nach wie vor auch von Software-Compilern verarbeitet werden können. Dabei können sich Änderungen am Quelltext auf das Ergebnis für CPUs (Central Processing Units) positiv und auf das Ergebnis der HLS unverhältnismäßig negativ auswirken, und umgekehrt. Das kann sich nachteilig auf die Produktivität von Softwareentwickler niederschlagen, die dann stets darauf achten müssen, diese ambivalenten Änderungen am Quelltext zu vermeiden. Ein Ziel dieser Arbeit ist es daher auch, diese Auswirkungen zu analysieren.

Die Untersuchung wird an Implementierungen von Algorithmen vorgenommen, die in Leistungsmessgeräten ausgeführt werden. Die Ausführung passiert in gewöhnlichen Prozessoren, was in Abschnitt 2.1 zusammen mit der Architektur der Messgeräte näher erklärt wird. Für eine zukünftige Variante dieser Geräte ist angedacht, dass die Ausführung einiger Aufgaben durch ein bereits vorhandenes FPGA beschleunigt und dadurch der Prozessor entlastet wird. Dieser kann dann leistungsschwächer ausgestattet sein.

Die Algorithmen bestehen zum einen aus der Berechnung von verschiedenen akkumulierten Messwerten (in Teilen der Arbeit auch als UIP- oder Statistikberechnung bezeichnet). Diese Werte spielen in der Elektrotechnik bei der Berechnung von Größen wie Effektivwert, Mittelwert, Spitzenwert sowie verschiedener Leistungskennwerten eine wichtige Rolle. Zum anderen wird ein Algorithmus zur Abtastratenkonvertierung behandelt (in Teilen der Arbeit auch als Polynominterpolation bezeichnet). Die Aufgabe ist hier, N Abtastwerte in 2^q äquidistante Werte zu konvertieren, wobei 2^q die nächstgrößere Zweierpotenz nach N ist ($q = \lceil \log_2 N \rceil$). Diese Konvertierung dient als Vorstufe für eine folgende FFT (Fast Fourier Transform).

1.3. Aufbau

In Kapitel 2 werden technische Grundlagen vermittelt, die für das Verständnis der Arbeit hilfreich sind. Außerdem werden kurz die Zielplattformen und Leistungsmessgeräte beschrieben.

In Kapitel 3 werden ähnliche wissenschaftliche Arbeiten zum Masterarbeitsthema aufgeführt und kurz beschrieben.

In Kapitel 4 wird die Implementierung verschiedener Signalverarbeitungsalgorithmen beschrieben. Die Implementierungen sind dergestalt, dass sie sich sowohl als Eingabe für eine HLS, als auch als Eingabe für Software-Compiler eignen.

In Kapitel 5 werden zielplattformoptimierte Implementierungen der Algorithmen entwickelt.

In Kapitel 6 wird danach ein Vergleich zwischen den portablen und optimierten Implementierungen vorgenommen, um zu ermitteln, wie viel Performanz zugunsten einer portablen Quelltextbasis aufgegeben wird.

Um herauszufinden, welche Quelltextkonstrukte für Performanzverluste verantwortlich sind, werden in Kapitel 7 die Auswirkungen von Quelltextmodifikationen auf das Ergebnis der HLS und Software-Kompilierung untersucht. Diese Erkenntnisse werden auf die portablen Implementierungen übertragen, damit nachteilige Quelltextkonstrukte vermieden werden.

In Kapitel 8 wird die Arbeit zusammengefasst und das Ergebnis aus der Sicht eines Leistungsmessgeräte-Herstellers bewertet. Eine der Entwicklungsmethodiken wird anhand dieser Bewertung auch für weitere Entwicklungen ausgewählt.

2. Technische Grundlagen und eingesetzte Systeme

In diesem Kapitel werden zunächst einige messtechnische Eckdaten der Leistungsmessgeräte genannt und grob deren Aufbau geschildert, damit Anforderungen an die zu implementierenden Funktionalitäten verständlicher werden. Dann werden die verwendeten Entwicklungswerkzeuge, Software- und Hardwareplattformen vorgestellt. Schlussendlich folgen einige technische Grundlagenerklärungen, die für das Verständnis dieser Masterarbeit wichtig sind.

2.1. Messtechnische Eckdaten

Die in dieser Arbeit zugrundegelegten messtechnischen Daten, wie Signalabtastrate und Abtastwertauflösung, entstammen dem LMG610, welches ein kompaktes Einkanalgerät aus der LMG600 Leistungsmessgeräte-Familie[48] repräsentiert. Das Gerät ist technisch gesehen ein generischer Computer und besteht neben den üblichen Komponenten aus einem austauschbaren Leistungsmesskanal, der in den drei Varianten A, B und C angeboten wird, die sich in Abtastrate, Auflösung, Filtern und zahlreichen anderen Details unterscheiden [48]. Verwendet wird in dieser Arbeit der A-Kanal, auf dem sich auch sämtliche nachfolgende Angaben beziehen. Die Kanäle bestehen jeweils aus einem Spannungs- und einem Strommesskanal (siehe Abbildung 2.1). Ein Spannungs- oder Strommesskanal besteht wiederum aus einem optional aktivierbaren, analogen Antialias-Tiefpassfilter, gefolgt von einem 18 Bit ADC (Analog Digital Converter) (siehe Tabelle 2.1 für weitere Details). Die Abtastwerte speisen ein Xilinx Spartan-6[41] FPGA (XC6SLX9-TQG144c) mit zwei unabhängig voneinander arbeitenden Datenpfaden. Hier werden zum einen die Nulldurchgänge für die Ermittlung einer *Syncfrequenz* erkannt, und zum anderen das Signal digital gefiltert. Die Ermittlung der Syncfrequenz kann von separat vorhandenen Syncsignalfiltern, einer DC-Addition, Hysterese und speziellen Demodulationsoptionen beeinflusst werden, die vor der Nulldurchgangserkennung angewendet werden. Die verarbeiteten Signale aller Kanäle werden von einem weiteren Xilinx Spartan-3[35] *Base*-FPGA (XC3S2000-4FGG676C) gesammelt, welches sich nicht auf dem Kanal,

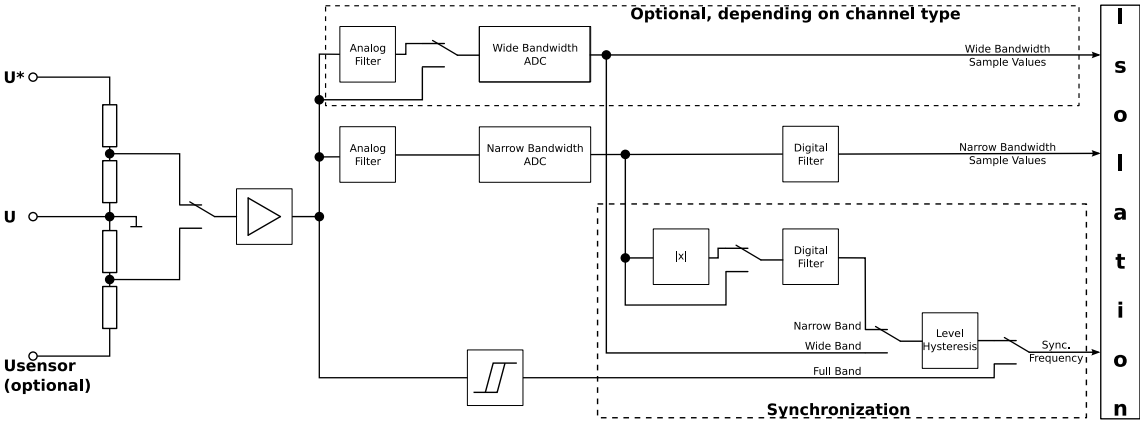

Abbildung 2.1.: Blockdiagramm eines Messkanals der LMG600 Familie. Dargestellt ist der Spannungsmessteil. Der Strommessteil ist, mit Ausnahme des Spannungsteilers im linken Teil des Diagramms, identisch aufgebaut. Grafik aus dem Messgerätehandbuch [48]

sondern auf dem eigentlichen LMG610 befindet. Hierbei wird eine galvanische Trennung der Kanäle vom Base-FPGA mittels Übertragern realisiert. In regelmäßigen Abständen zu 120 Werten pro Spannungs- und Stromkanal werden diese Werte an den Intel Atom E3845[18] Hauptprozessor übermittelt. Die Zahl 120 ergibt sich aus der maximalen ADC Abtastrate von etwa 1,2 MS/s und einer Puffer-Zeitlänge von ca. 100 µs für die Kommunikation zwischen FPGA und CPU. Während der Verarbeitung im FPGA liegen die Werte als 18 Bit Festkommazahlen im Intervall von $(-1.0 \ldots +1.0)$ vor, relativ bezogen auf den momentan aktiven Messbereich. Vor der Übertragung an den Prozessor werden die Werte in das *single-precision* IEEE754 Fließkommaformat [16] konvertiert. Auf dem Prozessor erfolgen unter anderem die Berechnung von statistischen Größen (Effektivwert, elektrische Wirkleistung [22]) und eine FFT zur Frequenzanalyse.

Abhängig von der Messanwendung werden unterschiedliche Anforderungen an das Ausgabesignal des ADC gestellt. Muss die Leistung auch aus hochfrequenten Signalanteilen erfasst werden, wird eine hohe Abtastrate benötigt und eine statistische Abtastung des Signals (*random sampling*) ohne aktivierten Antialias-Filter vorgenommen [22], da das Filter interessante hochfrequente Verzerrungen des Signals dämpfen würde. Soll aber die Leistungsverteilung in niedrigen Frequenzbereichen des Signals ermittelt werden, ist ein Antialias-Filter für die Einhaltung des Nyquist-Shannon-Abtasttheorems [31] erforderlich. Deshalb kommt in den Messkanälen ein *Breitband*-Wandler für die statistische Abtastung und ein *Schmalband*-Wandler für das alias-freie Messen zur Anwendung. In Tabelle 2.1 sind die in dieser Masterarbeit angenommenen Eckdaten angegeben, die dem Handbuch der LMG600 Familie [48] entnommen wurden.

Tabelle 2.1.: Eckdaten der ADCs. Die ADCs arbeiten nach dem SAR-Verfahren. Es werden zwei verschiedene ADCs verwendet, die jeweils für unterschiedliche Messaufgaben gebraucht werden.

	Katalognummer	Abtastrate	Auflösung
Breitband	AD7643[1]	$1,\overline{21}$ MS/s	18 Bit
Schmalband	AD7691[2]	$151,\overline{51}$ kS/s	18 Bit

Für die plattformunabhängigen Implementierungen aus Kapitel 4 wird für die statistische Kennwertberechnung der Breitbandwandler als Datenquelle angenommen, während bei der Abtastratenkonvertierung die Daten des Schmalbandwandlers zugrunde gelegt werden. Die Implementierungen erhalten Werte im Intervall $(-1.0 \ldots + 1.0)$.

2.2. Zielplattformen

Die Implementierungen der Algorithmen werden auf einem Intel Atom E3845[18] und ARM Cortex-A9 Prozessor[4] softwareseitig, sowie auf einem Xilinx SoC-FPGA hardwareseitig evaluiert und für diese in Kapitel 5 optimiert. Die programmierbare Logik und der ARM Prozessor sind Bestandteile eines Zynq-7010 SoC[46][1]. In den nachfolgenden Abschnitten werden der Intel Prozessor und das SoC detaillierter beschrieben. Dabei wird bei den Prozessoren besonders viel Wert auf die Verfügbarkeit sogenannter *Vektorinstruktionen* gelegt. Vektorinstruktionen können auf mehreren Werten (Vektorelementen) mit nur einer Instruktion parallel operieren, deshalb werden diese Instruktionen auch als SIMD (Single Instruction, Multiple Data) Instruktionen bezeichnet. In Kapitel 5 wird auf die Verwendung dieser Technik näher eingegangen, inklusive einige der dabei auftretenden Probleme.

Der Grund für die Auswahl der Intel- und ARM-Prozessoren liegt in der Verwendung der Intel CPU in einem bereits existierenden Leistungsmessgerät, dem LMG610 (siehe Abschnitt 2.1). Dieses Messgerät enthält neben der Intel CPU auch ein separates FPGA. Für die Entwicklung eines kompakteren Gerätes ist geplant, die Intel CPU durch einen leistungsschwächeren ARM Prozessor zu ersetzen. Erste Versuche ergaben, dass nicht alle anfallenden Aufgaben auf dieser schwächeren CPU ausgeführt werden können, weshalb für diese Aufgaben ein FPGA als Beschleuniger verwendet werden soll. Da das Zynq-7010 SoC sowohl eine ARM Cortex-A9 CPU als auch einen Bereich für programmierbare Logik enthält, werden die Imple-

[1] Xilinx bezeichnet den programmierbaren Logikteil nicht als FPGA.

mentierungen in dieser Masterarbeit auf beiden Ausführungseinheiten dieses SoC evaluiert. Tabelle 2.2 enthält einige Kennwerte der eingesetzten CPUs und ist ein Versuch der Performanzeinordnung beider Prozessoren. Die Angaben beziehen sich nur auf einen CPU-Kern und sind Abschätzungen der Idealperformanz, ohne Beachtung von real existierenden, aber nicht pauschal vorhersehbaren Pipelinekonflikten (engl. *hazards*) und ähnlichem. Die Intel Atom CPU kann Instruktionen innerhalb seiner beiden Fließkomma-Pipelines nicht OoO (Out of Order) ausführen. Jedoch können Instruktionen vor der Zuweisung an die Pipelines umsortiert werden. Der Transfer von Ergebnissen zwischen den Intel Atom Pipelines passiert ohne Taktversatz[12] (engl. *data bypass*), sodass es möglich scheint, dass eine entsprechend ausgeführte *add*-Operation auf die Ergebnisse einer zuvor ausgeführten *mul*-Operation der anderen Pipeline ohne Performanzverlust zugreifen kann. Es wurde das Durchsatzverhalten der Vektorinstruktionen `mulps` und `addps` vom Intel-Prozessor bzw. `vmla` vom ARM-Prozessor zugrunde gelegt (aus [11] und [3]), welche die *multiply-accumulate*-Operation durchführen. Tabelle 2.3 enthält schließlich einige Kennwerte zur programmierbaren Logik des Zynq-7010.

Tabelle 2.2.: Verwendete CPUs. Taktrate (C), Fließkomma-Ausführungseinheiten (E) und die sich ergebende Anzahl an Fließkommaberechnungen pro Sekunde (F). Der Wert für F wurde durch $C \times E \times 4 \times 0,5$ ermittelt. Der Faktor vier wird wegen der mögliche Verwendung von SIMD-Instruktionen mit vier Vektorelementen, der Faktor $0,5$ wegen dem Durchsatzverhalten von einer *multiply-accumulate* Operation pro zwei Takten verwendet. Für die Intel-CPU wurde ein Wert von E mit $1,5$ angenommen, da eine Parallelität nur zwischen Additionen und restlichen Operationen bestehen kann.

	Taktrate (C)	Daten-Cache	FP-EUs (E)	FLOPS (F)
Atom E3845 [12][11][18]	1,91 GHz	64 KiB 6-way L1 1 MiB 16-way L2	OoO zw. Pipelines FP0: mul, div, conv FP1: add	5,73 $\times 10^9$
Cortex-A9 [46][3]	0,667 GHz	32 kB 4-way L1 512 KiB 8-way L2	2× OoO SIMD	2,67 $\times 10^9$

Tabelle 2.3.: Programmierbare Logik des Zynq-7010

RAM Blöcke	DSPs	LUTs	FFs
60 · 36 kBit	80	17600	35200

2.2.1. Intel Atom E3845

Intel Atom Prozessoren aus der E3800 Familie implementieren die *Intel-64* Architektur, auch als *x86_64* bekannt, die einen 64 Bit Befehlssatz und einen 32 Bit Kompatibilitätsmodus für die *IA-32* Architektur enthält. Die Familie basiert auf der Mikroarchitektur *Silvermont*, die auf eine kleine Energieaufnahme spezialisiert ist [20]. Das Modell, welches hier untersucht wird, ist der Atom E3845. Dieser besitzt vier Prozessorkerne und implementiert die SIMD Instruktionen der *Streaming SIMD Extensions* bis einschließlich Version 4.2 und die *Supplemental Streaming SIMD Extensions 3 (SSSE3)*. Alle unterstützten SSE Versionen sind auf maximal 128 Bit breite Register beschränkt. Erst die nicht implementierten AVX SIMD Erweiterungen der Intel-64 Architektur bieten Registerbreiten von 256 Bit bzw. 515 Bit bei AVX-512. Bereits die erste SSE Variante enthält die Unterstützung von 128 Bit Register, und somit Vektorberechnungen auf vier `float`-Werten pro Instruktion.

2.2.2. Xilinx Zynq 7010

Verwendet wird das Zynq-7010 (XC7Z010CLG400-1) der Xilinx Zynq-7000 Familie [45], nachfolgend verkürzend auch *Zynq* genannt. Abbildung 2.2 zeigt die Architektur dieser SoCs. Das System besteht aus zwei Bereichen, der sogenannten „Programmable Logic“ oder PL, in dem sich das FPGA befindet, und dem „Processing System“-Bereich, auf dem sich der Prozessor und die Verbindungslogik der einzelnen ICs (Integrated Circuits) befinden.

Im nachfolgenden Abschnitt wird der ARM Cortex-A9 Prozessor vorgestellt, der sich in Abbildung 2.2 rechts oben befindet. In Abbildung 2.3 wird dessen Mikroarchitektur detaillierter dargestellt.

2.2.3. ARM Cortex-A9

Im Handbuch [4, S. 19] wird der Prozessor vom Hersteller *ARM Limited* beschrieben als

> The Cortex-A9 processor is a high-performance, low-power, ARM macrocell with an L1 cache subsystem that provides full virtual memory

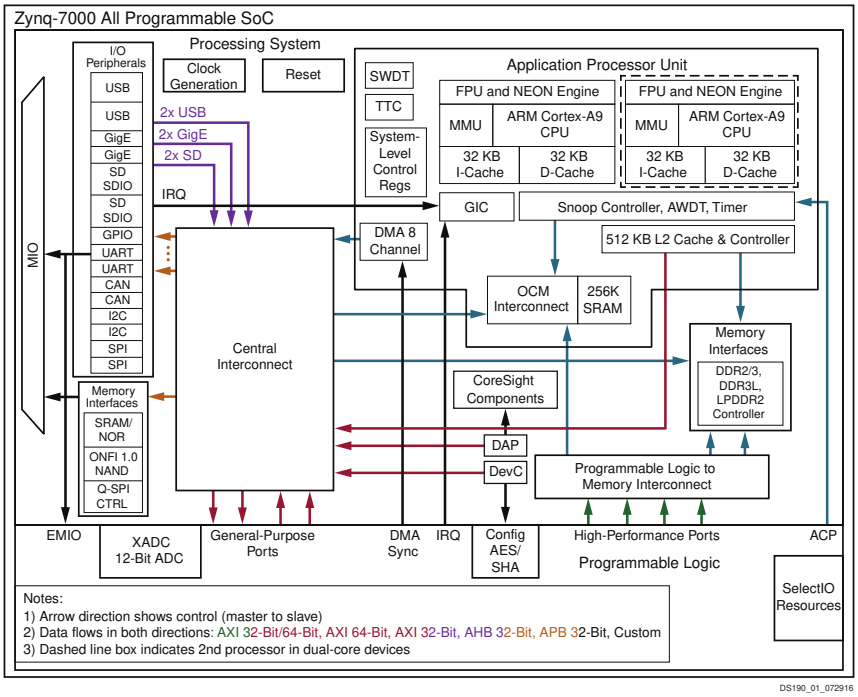

Abbildung 2.2.: Architektur der Zynq-7000 Familie. Grafik aus dem Datenblatt [45]

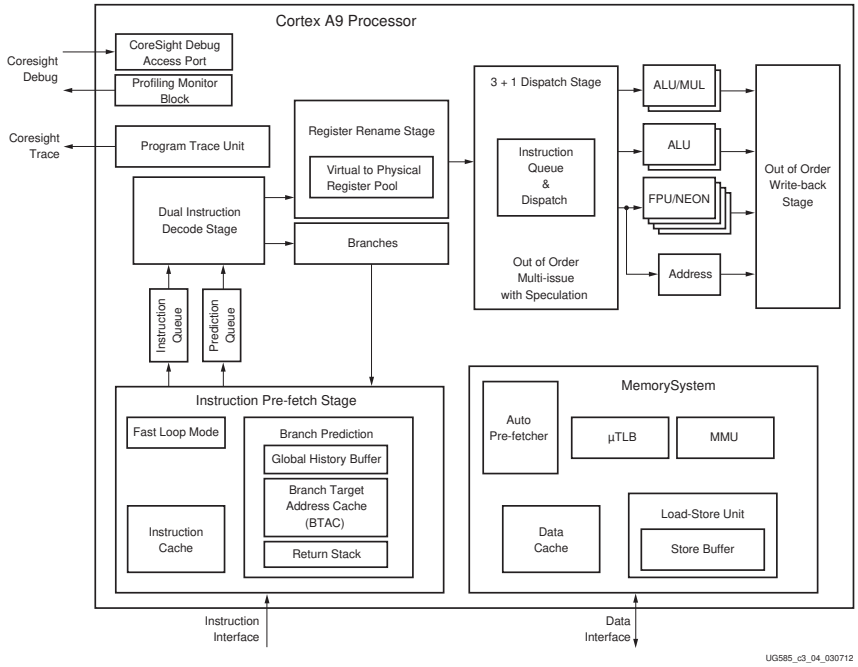

Abbildung 2.3.: Architektur von ARM Cortex-A9 im Zynq SoC. Grafik aus [46]

> capabilities. The Cortex-A9 processor implements the ARMv7-A architecture and runs 32-bit ARM instructions[...]

Die ARMv7-A RISC (Reduced Instruction Set Computer) Architektur besitzt zahlreiche optionale Erweiterungen, von denen einige nachfolgend genannt sind und von Xilinx verwendet werden. Das „A“ in ARMv7-A steht dabei für *Application* und bezeichnet eins von drei Profilen, zu denen auch *Realtime* (ARMv7-R) und *Microcontroller* (ARMv7-M) gehören. Das *Application* Profil ist die allgemeine Variante mit Unterstützung für virtuellen Speicher und die Standard 32 Bit ARM ISA (Instruction Set Architecture).

Zu erkennen sind in Abbildung 2.3 des Cortex-A9 die NEON Einheiten, welche die *Advanced SIMD* Erweiterung der ARMv7-A Architektur implementieren. Diese Erweiterung unterstützt das Arbeiten auf Vektoren von Ganz- und Fließkommawerten. Damit können Additionen, Multiplikationen und zahlreiche andere Operationen parallel ausgeführt werden. Die NEON Einheiten sind für die Ausführung von Instruktionen zuständig, die auf bis zu 128 Bit breiten Registern arbeiten, womit bis zu vier `float` Fließkommazahlen pro ausgeführter SIMD Instruktion abgearbeitet werden.

2.3. Entwicklungsumgebungen

In dieser Masterarbeit wird die Vivado HLS 2016.4[2] IDE (Integrated Development Environment) von Xilinx in der kostenlosen WebPACK Variante verwendet (nachfolgend verkürzend *VHLS* genannt). Dabei handelt es sich um eine Version mit einer reduzierten Zahl an unterstützten Geräten, die neben VHLS auch die Vivado IDE enthält. Letztere wird zum Entwickeln der handgeschriebenen VHDL-Implementierungen verwendet. Der beschriebene C++-Quelltext aus Kapitel 4 wird in ebendieser IDE entwickelt und funktionell getestet. Bei VHLS können die sogenannten *Testbenches* sowohl in Software (*C-Simulation*), als auch in simulierter Hardware (*C/RTL-Cosimulation*) ausgeführt werden. Bei der Co-Simulation konvertiert VHLS die durch C++ erstellten Testeingaben automatisch in Signale, die dann von einer RTL (Register Transfer Level) Simulation der HLS Architektur verarbeitet und danach wieder zurück konvertiert werden. Für die Entwicklung der optimierten Implementierungen aus Kapitel 5 wird die Eclipse IDE verwendet. Die im einzelnen verwendeten C++ Compiler sind in Kapitel 5 aufgeführt.

[2] Die Entscheidung für Version 2016.4 und gegen 2017.1, zum damaligen Zeitpunkt die aktuelle Version, fiel aufgrund einer Empfehlung des Messgeräteherstellers, nach dessen Erfahrung die erste Vivado Version eines neuen Jahres oft eine Reihe von Fehlern enthält, die im Laufe des Jahres verschwinden.

2.4. High-Level Synthese

Die HLS wird bei VHLS auf eine Hauptfunktion angewendet (engl. *toplevel function*). Der Quelltext dieser Hauptfunktion und alle von ihr (auch transitiv) verwendeten Funktionen werden durch die HLS verarbeitet und zu HDL-Code konvertiert. Nachdem der C++-Quelltext von einem C++ Frontend in eine geeignete Zwischenform überführt wurde, folgen die HLS-spezifischen Schritte:

Die *Allokation* ordnet Quelltextoperationen (z.B. ein Funktionsaufruf, oder eine Addition) im Zwischencode einzelne Operator-*Instanzen* zu. Ein solcher Operator kann z.B ein Addierer sein. Zwei Instanzen sind unabhängig und können prinzipiell parallel arbeiten. Werden einer Operator-Instanz mehrere Operationen zugeordnet, müssen diese zeitlich nacheinander ausgeführt und die Instanz wiederverwendet werden (engl. *time sharing*).

Das *Scheduling* ordnet den vorher allozierten Operationen Takt-Zyklen zu, unter Beachtung der vom Designer vorgegebenen Bedingungen (zum Beispiel der gewünschten Taktrate). Je nach Dauer einzelner Operationen können hiervon mehrere nacheinander im gleichen Zyklus berechnet werden (engl. *operator chaining*).

Danach werden die Instanzen durch die *Binding*-Phase konkreten FPGA-Ressourcen zugeordnet und miteinander verbunden. Eine Addition kann bspw. entweder über kombinatorische Logik oder über ein DSP (Digital Signal Processor) Hardwarebaustein realisiert werden.

Der Kontrollfluss wird aus den Kontrollstrukturen des Quelltextes extrahiert und für die Erstellung einer Zustandsmaschine benutzt. Diese koordiniert das Ansteuern der Operatoren sowie die Realisierung von Verzweigungen und Schleifen.

Die *Scheduling* und *Binding* Phasen iterieren solange, bis das Design allen Vorgaben des Designers und der Hardware entspricht. Kann ein solches Design nicht gefunden werden, wird das im Synthese-Report vermerkt (siehe Abschnitt 2.4.1) und versucht, die Vorgaben so gut wie möglich einzuhalten. Der HLS-Compiler kann bei der Synthese entweder auf einen möglichst geringen Platzbedarf auf dem FPGA (engl. *area*) oder auf eine möglichst hohe Geschwindigkeit optimieren. Die Parameter, welche die Geschwindigkeit beschreiben, sind die Latenz (engl. *latency*), das Initiierungsintervall (engl. *initiation interval*, II) und die zeitliche Länge einer Taktperiode (Kehrwert der Taktfrequenz). Die Latenz beschreibt die Anzahl der Takte, bis nach einer Eingabe die zugehörige Ausgabe erfolgt. Bei einer kurzen Taktperiode können nur wenige Operationen im gleichen Takt erfolgen, sodass im Allgemeinen bei

einer Erhöhung der Taktfrequenz die Latenz der Schaltung steigt, da Operationen über Takte hinweg verteilt werden müssen. Das Initiierungsintervall beschreibt die Anzahl der Takte zwischen der Annahme von zwei aufeinander folgenden Eingaben. Der Begriff *Durchsatz* (engl. *throughput*) beschreibt den Kehrwert des Initiierungsintervalls und ist ein Maß für die Geschwindigkeit eines synthetisierten Systems. Über *Direktiven* können einzelne der genannten Parameter maximiert, minimiert oder limitiert werden. In Kapitel 4 und Kapitel 6 wird auf diese Metriken und die Möglichkeiten der Einflussnahme bei Bedarf zurückgegriffen.

Der synthetisierte Code kommuniziert mit dem restlichen System über die Funktionsparameter der Hauptfunktion, die durch VHLS automatisch nach HDL I/O-Ports übersetzt werden. Hier bestimmen die Bitweiten der Funktionsparameter und das Lese- bzw. Schreibmuster der Funktion darüber, aus wie vielen Bits die I/O-Ports bestehen und welches Protokoll den Datenaustausch regelt. Außerdem kann zwischen zwei Handshake-Protokollen gewählt werden, welche die *Block-Level* Schnittstelle definieren. Dadurch kann die Latenz der Schaltung dynamisch von den Pegeln der I/O-Ports abhängen. Ein einfaches Protokoll mit *done*, *ready-for-input* und *idling* Ausgabeports sowie einem *start* Eingabeport ist dabei das Standardprotokoll. Das eingesetzte Protokoll und die Block-Level Schnittstelle können über Direktiven verändert werden.

2.4.1. Analyse von Synthese-Ergebnissen

Das Synthese-Ergebnis von VHLS wird im Synthese-Report angezeigt. Hier wird die maximale und minimale Latenz der Schaltung, sowie die erreichte Taktperiode angegeben. Der Entwickler muss bei VHLS eine gewünschte Taktrate in Form der Länge einer Taktperiode vorgeben, die der des später eingesetzten FPGA entsprechen sollte. Diese Taktperiode, abzüglich einer $12,5$ % Unsicherheit für die spätere Synthese der Netzliste und das Routing, wird als Zielvorgabe von VHLS übernommen. VHLS bietet eine Übersicht über die Zuordnung von Operationen des Quelltextes zu Zuständen der Zustandsmaschine, die aus dem Kontrollfluss des Programms abgeleitet wurden. Außerdem werden die Instanzen aufgelistet, die für diese Operationen alloziert wurden.

In dieser Masterarbeit wird stets 100 MHz, also eine 10 ns Taktperiode, eingestellt. Es ist angedacht, eine Taktfrequenz in diesem Bereich für das Kanal- und Base-FPGA bei zukünftigen Leistungsmessgeräten des Herstellers zu verwenden (siehe Abschnitt 2.1). Die erreichte Latenz wird oft in der Form $(\min, \max)$ angegeben, wobei der erste Wert die kleinste Latenz und der zweite Wert die größte Latenz ist. Die tatsächliche Latenz hängt von den dynamischen Parameterwerten der Hauptfunktion ab.

Neben den Zeitparametern zeigt VHLS auch eine *Abschätzung* des Ressourcen-

bedarfs des Syntheseergebnisses in Form von Block-RAMs (Random Access Memories), DSPs, LUTs (Lookup Tables) und FFs (Flip Flops) an. Wird mit Vivado aber tatsächlich eine Schaltung für die programmierbare Logik optimiert, platziert und verdrahtet (engl. *place and route*), fällt der endgültige Ressourcenbedarf normalerweise anders aus. Deshalb beziehen sich sämtliche Angaben in dieser Masterarbeit auf die Ergebnisse nach einer erfolgten Verdrahtung. Falls das wegen einer zu hohen Ressourcenlast nicht möglich ist, werden die Abschätzungen von VHLS angegeben und der Sachverhalt durch eine rote Textfarbe kenntlich gemacht.

2.5. Wahl der Zahldarstellung

Zur Speicherung von Zahlenwerten kann bei VHLS zwischen verschiedenen Typen gewählt werden. In diesem Abschnitt werden Typen vorgestellt, mit denen rationale und ganze Zahlen mit einer vorgegebenen Genauigkeit und aus einem bestimmten Wertebereich repräsentiert werden können.

2.5.1. Festkomma- und Fließkommazahlen

Weil FPGAs nicht für bestimmte Aufgaben, sondern gerade für den Aufbau beliebiger Systeme gedacht sind, sollte jedwede Komponente auf einem FPGA bei einer Mehrheit der Anwendungen von Nutzen sein. Da traditionell bei einer großen Zahl von FPGA-Anwendungen keine Fließkommaberechnungen gebraucht werden, besitzen zur Zeit nur die wenigsten FPGAs eingebaute Fließkomma-DSPs. In [25] wird die Beschleunigung eines künstlichen neuronalen Netzwerks mithilfe von FPGAs beschrieben und mit dem Ergebnis einer GPGPU-Beschleunigung verglichen. In der Zusammenfassung beschreiben die Autoren ihren Plan, mithilfe von neuen FPGAs wie dem Arria 10 und Stratix 10, die eingebettete Fließkomma-DSPs besitzen, die Performanz weiter steigern zu wollen. Es wird sich zeigen, ob FPGAs mit eingebetteten Fließkomma-DSPs einen Absatzmarkt finden.

Das verwendete Zynq besitzt *keine* eingebetteten Fließkomma-DSPs. Jegliche Fließkommaberechnungen werden also vom HLS-Compiler über die konfigurierbaren Logikblöcke und Ganzzahl-DSPs umgesetzt. Das folgende Beispiel benutzt zwei DSPs, 210 FFs und 391 LUTs für die Addition von zwei 32 Bit Fließkommazahlen.

```
1 void add(float a, float b, float *c) {
2   *c = a + b;
3 }
```

Quelltext 2.1: Addition mit Fließkommazahlen

Eine gute Alternative ist die Benutzung von Festkommadarstellungen, falls alle zu verrechnenden Werte einen ähnlichen Wertebereich haben. Im C++-Quelltext

kann ein Festkommawert durch ein spezielles Klassen-Template `ap_fixed<W, I>` deklariert werden. Dabei gibt `W` die Gesamtanzahl der Bits und `I` die Anzahl der Ganzzahlbits an, inklusive Vorzeichenbit. Das MSB (Most Significant Bit) hat also die Wertigkeit -2^{I-1} und das LSB (Least Significant Bit) die Wertigkeit 2^{I-W}. Im Folgenden ist das vorherige Beispiel mit der Festkommadarstellung implementiert. Die HLS hiervon benötigt keine DSPs oder FFs und nur 33 LUTs.

```
1 void add(ap_fixed<32, 1> a, ap_fixed<32, 1> b, ap_fixed<32, 2> *c) {
2   *c = a + b;
3 }
```

Quelltext 2.2: Addition von Festkommazahlen. `add(0.75, 0.75, &c)` benötigt ein Variable `c` mit 2 Ganzzahlbits (inklusive Vorzeichenbit), damit die Zahl $1,5$ repräsentiert werden kann.

Ein direkter Vergleich dieser Festkommatypen mit `int32_t` als Parametertypen ist naheliegend, das Verhalten dann aber nicht mehr äquivalent. Es wird dann zwar ein LUT weniger benötigt, allerdings werden in der `ap_fixed`-Variante für die Berechnung der rechten Seite der Zuweisung 33 Bits verwendet, die durch die Zuweisung auf ihre 32 höchstwertigen Bits gekürzt werden. Bei Bedarf kann das Rundungsverhalten des verworfenen LSB, das in der Standardeinstellung auf *truncate* gesetzt ist, durch weitere Templateargumente modifiziert werden. Bekommt `c` die Templateargumente `<32, 1>`, ist der generierte VHDL-Code identisch mit dem der `int32_t`-Version. Hierbei muss jedoch beachtet werden, dass das Verhalten von Überläufen in Berechnungen mit vorzeichenbehafteten Ganzzahltypen in C++[17] nicht definiert ist. Im Handbuch[43] von VHLS konnte der Autor keine Informationen zum Überlaufverhalten von primitiven Ganzzahltypen finden.

In der Untersuchung dieser Arbeit werden sowohl die Fließkomma- als auch die Festkommadarstellung verwendet. Damit der C++-Quelltext portabel bleibt, und die verschiedenen Varianten schnell austauschbar sind, werden die Algorithmen als C++ Templates implementiert und dann für die HLS jeweils mit den gewünschten Datentypen und anderen Spezifika instantiiert, wie das folgende Beispiel anhand von **`float`** demonstriert.

```
1 template<typename T>
2 void add(T  a, T  b, T *c) {
3    *c = a + b;
4 }
5
6 void add_float(float a, float b, float *c) {
7    add(a, b, c);
8 }
```

Quelltext 2.3: Addition mit parametrisierten Typen

2.5.2. Ganzzahlen mit vorgegebener Genauigkeit

Neben der Unterstützung von Festkommawerten durch `ap_fixed` stehen bei VHLS auch die beiden Klassen-Templates `ap_int<W>` und `ap_uint<W>` zur Verfügung. In Abschnitt 4.3 besteht z.B. die Notwendigkeit, ganze Zahlen bis 2^{17} zu speichern. Mit der Klasse `ap_uint<18>` können dann im Gegensatz zur Verwendung von `uint32_t` 14 Bit eingespart werden, was sich bei Operationen mit diesen Werten auch auf die Latenz auswirken kann.

3. Vorherige Arbeiten

Der Autor konnte keine Arbeit finden, die sich explizit mit dem Ziel beschäftigt, eine HLS auf portablen Quelltext anzuwenden, der somit auch weiterhin für Software-Entwicklung eingesetzt werden kann.

In [27] werden Formeln für die sogenannte *Designproduktivität* (engl. *Design Productivity*) von HLS Methoden entwickelt und in einer Studie getestet. Eine Designproduktivität von > 1.0 für eine HLS bedeutet ein Vorteil gegenüber einer reinen Entwicklung auf Basis einer HDL. Die Metrik wird durch $P_D = G_{NRE}/L_Q$ definiert, wobei NRE den „Non-Recurring Engineering Effort“ beschreibt, also die Zeitkosten für Entwicklung einschließlich Verifikation. Davon abgeleitet wird G_{NRE} als die Einsparung (engl. *Gain*) der HLS an diesen Kosten. Die Einsparung wird als Verhältnis der Kosten des HLS-Ansatzes zu den Entwicklungskosten des HDL-Ansatzes dargestellt. Demgegenüber steht der Verlust der Ergebnisqualität, der durch L_Q ausgedrückt wird. Als quantitative Qualitätsmerkmale werden sowohl Ressourcenverbrauch als auch Performanzkennwerte berücksichtigt. Im einzelnen sind dies die Anzahl der gebrauchten Register, LUTs, Block-RAMs und DSPs als Ressourcenkennwerte. Die Latenz und Taktperiodenlänge (engl. *Operating Period*) werden als Leistungskennwerte verwendet. Für die Studie wird als HLS Eingabesprache CAPH [30] übernommen, die aus dem akademischen Umfeld stammt und speziell für die HLS entwickelt wurde. Als HDL wird VHDL verwendet. Im Rahmen der Studie werden verschiedene Versionen eines Interpolationsfilters des Videokompressions-Algorithmus MPEG HEVC implementiert, der für die Interpolation von Pixelinhalten auf Nachbarpixel zuständig ist. Die Arbeit diskutiert Designproduktivität nur in Bezug auf die Entwicklung einer Lösung für FPGAs. In der vorliegenden Arbeit wird der HLS-Ansatz aus der Perspektive einer *zusätzlichen* Zielplattform betrachtet.

In [14] wird mithilfe von VHLS (2014.1) eine leicht abgeänderte und vereinfachte Variante der Referenzimplementierung des AES (Advanced Encryption Standard)-Algorithmus mit handgeschriebenen HDL-Quelltext verglichen. Die HLS Eingabesprache ist C, wie in dieser Masterarbeit. Der Algorithmus wurde wegen der relativ hohen Anzahl an Operationen gewählt, die sich nicht gut parallelisieren lassen. Die Referenzimplementierung wird im Wesentlichen auf die Unterstützung von 128 Bit Schlüsseln beschränkt. Dieser Algorithmus wird sowohl mit VHDL in der Grundform als auch mit HLS in jeweils drei verschiedenen Implementierungen eva-

luiert, wobei aufeinander aufbauend unterschiedliche Quelltextoptimierungen und HLS-spezifische Direktiven zum Einsatz kommen. Auch hier überschneidet sich das Thema teilweise mit dieser Masterarbeit. Zu den Optimierungen auf Quelltextebene zählt die Ersetzung von Multiplikationen, die in der Originalimplementierung durch eine Umsetzungstabelle (engl. *lookup table*) implementiert werden, durch Bitshifts und XORs. Dadurch kann Speicherplatz auf dem FPGA gespart werden. Eine weitere Optimierung entfernt die Vorausberechnung der geheimen Rundenschlüssel am Anfang der Ausführung, und berechnet die Rundenschlüssel stattdessen *on-the-fly* beim Abarbeiten der einzelnen Runden. Diese Variante (HLS_1) reduziert den Ressourcenverbrauch je nach getesteten FPGA um 26 % bis 44 % und die Latenz um 56 %, verglichen mit der nicht optimierten Variante. Nachfolgend werden HLS Direktiven eingesetzt, um bspw. Schleifen abzurollen, Funktionsaufrufe mit deren Inhalt zu ersetzen (engl. *inline*) und Lese-/Schreibzugriffe auf `word8` Arrays in Zugriffe von 128 Bit zusammenzufassen, wodurch 128 Bit parallel abgearbeitet werden können. Diese Variante (HLS_2) verbessert die Latenz von 3224 Takte für HLS_1 auf 11 Takte, was exakt dem Wert der VHDL Implementierung entspricht. Der Ressourcenverbrauch ist bei der HLS für den Xilinx Spartan 6 etwas besser als bei handgeschriebenen VHDL (343 zu 354 Slices/LEs/ALMs) und beim Xilinx Virtex 4 schlechter (344 zu 317). Die Autoren stellen fest, dass weitergehende Verbesserungen der Durchsatzleistung für HLS sehr schwierig sind. Die Arbeit hat eine starke Überschneidung mit dieser Masterarbeit hinsichtlich des Vergleichs von HLS und VHDL Lösungen. Eine interessante Frage scheint zu sein, wie sich vorgenommenen Quelltextänderungen auf eine Software-Zielplattform auswirken. Dieser Frage wird mithilfe der implementierten Signalverarbeitungsalgorithmen im Rahmen dieser Masterarbeit nachgegangen.

In [49] wird, ebenfalls mit VHLS (2013.3), ein Algorithmus aus der Video- und Bildverarbeitung implementiert und mit einer handgeschriebenen VHDL-Variante verglichen. Der Algorithmus realisiert einen digitalen Glättungsfilter. Zur Evaluation der Performanz beider Lösungen wird die Xilinx-TRD Infrastruktur [44] verwendet. Diese enthält unter anderem einen digitalen Demo-Filter, der von Xilinx beispielhaft in die Architektur integriert wurde. Dieser Beispielfilter wird durch den Glättungsfilter ersetzt. Die durch den Autor selbst vorgenommene Implementierung der VHDL-Version wird bezüglich maximal möglicher Taktrate, Ressourcenverbrauch und dem NRE (Non-Recurring Engineering Effort) mit der HLS-Version verglichen. Damit die NRE-Angabe eingeordnet werden kann, beschreibt der Autor seine Erfahrung auf beiden Gebieten mit 4,5 Jahren Embedded-Software-Entwicklung und 3,5 Jahren Custom-Logic-Design mit HDLs. In der vorliegenden Arbeit gibt es noch eine etwas verstärkte Anforderung an die HLS, denn der Autor hat keine Erfahrung auf dem Gebiet der HDLs vorzuweisen.

4. Plattformunabhängige Implementierungen

Die Masterarbeit erforscht, inwieweit es möglich ist, die gleiche Implementierung der Algorithmen sowohl für CPUs als auch die HLS zu verwenden. Deshalb werden in diesem Kapitel Implementierungen vorgestellt, die keine Konstrukte enthalten, welche nur von einigen der Zielplattformen unterstützt werden.

4.1. Übersicht

Es werden zwei Algorithmen untersucht, die in Leistungsmessgeräten im Einsatz sind und in Abschnitt 1.2 bereits kurz beschrieben wurden. Für die Implementierung dieser Algorithmen werden C++ *Templates* verwendet. Templates erlauben die Parametrisierung von Funktionen und Klassen, unter anderem mit Typen und Ganzzahlkonstanten. Eine *Instantiierung* kann aus dem Template, unter Angabe von konkreten Typen oder Konstanten, eine *Spezialisierung* des Templates erstellen, die mit diesen Typen und Konstanten arbeitet. Hierdurch können unterschiedliche Varianten der gleichen Implementierung erstellt werden, die jeweils für bestimmte Zielplattformen besser geeignet sind als für andere.

Zum einen werden die Templates mit Typen instantiiert, die in typischen CPU-Implementierungen verwendet werden, z.B. mit dem Typen `float` und dem Ganzzahltypen `int`. Zum anderen werden Typen verwendet, die sich besser für FPGAs eignen. Wie in Abschnitt 2.5 demonstriert, kann es bspw. vorteilhaft sein, für die HLS eine Festkommadarstellung zu verwenden. Zum Ermitteln der Vorteile dieses Entwurfs wird die erste Variante mit `float` zusätzlich einer HLS unterzogen.

4.1.1. Funktionale Verifikation

Das korrekte Verhalten des synthetisierten VHDL-Codes wurde mithilfe von Testbenches (siehe Abschnitt 2.3) verifiziert. Für die Statistikberechnung wurden die errechneten Werte mit den erwarteten Ergebnissen verglichen und eine maximale

Abweichung von 10 ppm gefordert[1]. Die verwendeten Testeingaben sind durchgehend maximal- und minimale Abtastwerte, sowie wechselnd maximale und minimale Abtastwerte. Für die Abtastratenkonvertierung wurden die Ergebnisse anhand ihrer Kurvenform im Zeitbereich qualitativ mit den erwarteten Werten abgeglichen. Dabei wurden als Eingaben unterschiedliche Signalformen und Spezialfälle gewählt, z.B. eine Eingabewertanzahl von exakt 2^q, welche durch die Konvertierung nicht verändert werden sollte. Die generierten VHDL-Quellen wurden nicht auf einem FPGA ausgeführt, jedoch synthetisiert, platziert und verdrahtet (engl. *place and route*), um Performanz- und Ressourcenkennwerte zu erhalten. Es wird angenommen, dass eine erfolgreiche HLS auch zu einer funktionierenden und programmierbaren Schaltung führt.

4.2. Statistikberechnung

Die untersuchten Algorithmen umfassen die Berechnung der folgenden Größen.

- Quadratischer Mittelwert $U_{rms} = \sqrt{\frac{1}{n}\sum_{k=1}^{n} u_k^2}$
- Arithmetischer Mittelwert $U_{dc} = \frac{1}{n}\sum_{k=1}^{n} u_k$
- Absolutes Minimum $U_{pkp} = \min_{k=1}^{n} u_k$ und Maximum $U_{pkn} = \max_{k=1}^{n} u_k$
- Summe von Produkten $P = \sum_{k=1}^{n} u_k \cdot i_k$

Dabei bezeichnet n die Anzahl der Abtastwerte und u_k bzw. i_k den Wert von Spannung bzw. Strom an Position k. Im weiteren Verlauf werden diese Werte zusammenfassend *UIP-Werte* genannt. Die ersten drei Größen werden sowohl für u_k als auch für die Folge i_k berechnet (und werden dann entsprechend I_{rms} usw. genannt). Abschnitt 4.2.1 listet und erklärt den C++-Quelltext der Implementierungen. In Abschnitt 4.2.2 werden Schwierigkeiten genannt, die bei der Implementierung auftraten.

4.2.1. Quelltext

Die berechneten UIP-Werte werden in einem Objekt der Klasse `UipPowerResult` gespeichert. Dieses Objekt speichert den P-Wert und enthält zwei Unterobjekte der

[1] Diese Abweichung ergibt sich aus Vorgaben des Messgeräteherstellers. Erlaubt sind laut Messgerätehandbuch[48] 100 ppm. Wegen Messunsicherheiten der Hardware von 90 ppm ergeben sich 10 ppm maximale Abweichung für Softwareberechnungen.

Klasse `UipLineResult`. Diese Unterobjekte speichern jeweils rms, dc, pkp und pkn separat für Spannung und Strom.

Zunächst wird das Klassen-Template `UipLineResult` gelistet.

```
1  template<typename Traits>
2  class UipLineResult {
3     typedef Traits::in_type in_type;
4     typedef AccumulateSquares<
5        in_type, Traits::el_n
6     >::type sqsum_type;
7     typedef Accumulate<
8        in_type, Traits::el_n
9     >::type dcsum_type;
10
11 public:
12    sqsum_type sqsum;
13    dcsum_type dcsum;
14    in_type pkp;
15    in_type pkn;
16
17 public:
18    UipLineResult() {
19       clear();
20    }
21
22    void clear() {
23       sqsum = 0;
24       dcsum = 0;
25       pkp = -mst::numeric_limits<
26          in_type>::max();
27       pkn = +mst::numeric_limits<
28          in_type>::max();
29    }
30
31    void addSample(Traits::in_type s) {
32       sqsum += s * s;
33       dcsum += s;
34       pkp = pkp < s ? s : pkp;
35       pkn = pkn > s ? s : pkn;
36    }
37
38    void average(IntegerCounter<in_type,
39       Traits::el_n>::type count)
40    {
41       sqsum /= count;
42
43       DivideAndDiscard<
44          sqsum_type, Traits::el_n
45       >::type sqsum_divided = sqsum;
46       sqsum = mst::sqrt(sqsum_divided);
47
48       dcsum /= count;
49    }
50 };
```

Quelltext 4.1: Berechnung von Mittelwert sowie Maximum und Minimum

Die beabsichtigte Verwendungsweise dieses Templates ist die Erzeugung von Anfangswerten (Null), das dann folgende Hinzufügen von Abtastwerten mit `addSample` und ein abschließender Aufruf von `average`, welches die Division und Quadratwurzel implementiert. Die Anzahl der Abtastwerte, die an `average` übergeben wird, muss extern mitgezählt werden. Diese Aufgabe übernimmt die `UipPowerResult`-Klasse, zusammen mit der Berechnung von P.

```
1  template<typename Traits>
2  class UipPowerResult {
3     typedef Traits::in_type in_type;
4
5  public:
6     UipLineResult<Traits> u;
7     UipLineResult<Traits> i;
8
9     AccumulateSquares<in_type,
10       Traits::el_n>::type p;
11    IntegerCounter<in_type,
12       Traits::el_n>::type in_count;
13
14 public:
15    UipPowerResult() {
16       clearPower();
17    }
18
```

```
19      void clear() {
20          clearPower();
21          u.clear();
22          i.clear();
23      }
24
25      void clearPower() {
26          p = 0;
27          in_count = 0;
28      }
29
30      void addSample(
31              in_type usample,
32              in_type isample) {
33          in_count++;
34          u.addSample(usample);
35          i.addSample(isample);
36          p += usample * isample;
37      }
38
39      void average() {
40          u.average(in_count);
41          i.average(in_count);
42      }
43  };
```

Quelltext 4.2: Berechnung der Summe von Produkten und Zählen der Abtastwerte

Objekte dieser Klasse werden an der Schnittstelle der generierten Hauptfunktion verwendet. Die Member `u` und `i` speichern das Zwischen- oder Endergebnis für den Strom- und Spannungskanal. Wie in Abschnitt 2.1 beschrieben, ist der dynamische Wertebereich von $(-1.0 \ldots +1.0)$ und die Auflösung von 18 Bit von beiden Kanälen identisch. Deshalb wird in beiden Fällen der gleiche Datentyp verwendet. Als Template-Argumente für den Parameter `Traits` kann eine beliebige Klasse übergeben werden, solange die Member `Traits::el_n` und `Traits::in_type` wie in Tabelle 4.1 angegeben deklariert sind.

Tabelle 4.1.: Traitschnittstelle für die Implementierung des UIP-Algorithmus

Member	Kontrakt
`in_type`	Typ eines Abtastwertes
`el_n`	Die maximale Anzahl der Abtastwerte für `addSample`

Der Parameter `el_n` ist wichtig, weil er die Bitbreiten von sämtlichen Zwischenwerten beeinflusst. Damit beide in Abschnitt 4.1 vorgestellten Szenarien untersucht werden können, darf die Algorithmen-Implementierung keine direkte Typ-Auswahl treffen, sondern muss mithilfe der Eingabe-Typen (`in_type`) alle benötigten Typen für Zwischenwerte herleiten. Die dafür verwendeten *Meta-Funktionen* werden in Tabelle 4.2 aufgelistet. Einige dieser Funktionen werden auch für die Implementierung der Abtastratenkonvertierung verwendet.

Eine Meta-Funktion muss einen Member `type` deklarieren, der den Ergebnis-Typen liefert. Als Beispiel sei im Folgenden die Meta-Funktion `Accumulate` gelistet.

```
1  template<typename Type, int N>
2  struct Accumulate;
3
4  template<int N>
```

Tabelle 4.2.: Meta-Funktionen für die Implementierung des UIP-Algorithmus

Funktion	Ergebnis
`Accumulate<T,N>`	Typ, der $\sum_1^N t$ für beliebige Werte $t \in T$ repräsentieren kann.
`AccumulateSquares<T,N>`	Typ, der $\sum_1^N t^2$ für beliebige Werte $t \in T$ repräsentieren kann.
`Square<T>`	Typ, der t^2 für beliebige Werte $t \in T$ repräsentieren kann.
`DivideAndDiscard<T,N>`	Typ, der das höchstwertige Bit in t/N und niedrigstwertige Bit in t für beliebige Werte $t \in T$ repräsentieren kann.
`IntegerCounter<T,N>`	Ganzzahltyp, der N repräsentieren kann. Abhängig von `T` ist das entweder `int` oder `ap_uint<W>`.

```
5  struct Accumulate<float, N> {
6     typedef double type;
7  };
8
9  template<int N>
10 struct Accumulate<double, N> {
11    typedef double type;
12 };
13
14 template<int W, int I, int N>
15 struct Accumulate<ap_fixed<W, I>, N> {
16    static const int extra_bits = Log2Ceil<N>::value;
17
18 public:
19    typedef ap_fixed<W + extra_bits, I + extra_bits> type;
20 };
```

Quelltext 4.3: Beispiel für eine C++ Metafunktion. Dem Template `ap_fixed<W, I>` wird die Gesamtanzahl der Bits `W` und Anzahl der Ganzzahlbits `I` übergeben (siehe Abschnitt 2.5.1)

Das `UipPowerResult`-Template wird von den beiden Hauptfunktionen instantiiert, jeweils mit unterschiedlich definierten Traits. Die Hauptfunktion ist die Funktion, die den Einsprungspunkt der synthetisierten Architektur markiert (siehe Abschnitt 2.4). Entscheidet sich der Entwickler für eine HLS mit Fließkommaarithmetik oder soll die Ausführung auf dem CPU stattfinden, dann verwendet er das Trait `uip_float_traits`. Soll eine Festkommaarithmetik zum Einsatz kommen, wird das Trait `uip_apfixed_traits` verwendet. Das Template `UipHlsTraitsGenerator` erstellt eine geeignete Traitsklasse für eine Festkommaimplementierung mit `ap_fixed`, unter Angabe der Abtastwertauflösung in Bits (`AdcBits`), der maximalen Zykluszeit in Sekunden (`CycleS`) und der maximalen Abtastrate in Hertz (`MaxFreq`).

```
template<int NumberElements>
class UipFloatTraits {
public:
   static const int el_n =
      NumberElements;
   typedef float in_type;
};

typedef UipFloatTraits<60 * 1212121>
   uip_float_traits;

template<int AdcBits, int CycleS,
         int MaxFreq>
class UipHlsTraitsGenerator {
public:
   typedef struct UipHlsTraits {
      static const int el_n =
         CycleS * MaxFreq;
      typedef ap_fixed<AdcBits, 1>
         in_type;
   } type;
};

typedef UipHlsTraitsGenerator<
   18, 60, 1212121
>::type uip_apfixed_traits;
```

Quelltext 4.4: Definition der Fließkomma und Fixkomma UIP-Traits

Die Definition der Hauptfunktion für Festkommaarithmetik wird nachfolgend gezeigt. Diejenige für Fließkommaarithmetik ist mit dieser, bis auf das verwendete Trait, identisch. Der Parameter `lastSample` markiert den letzten Abtastwert. Bei einer Übergabe von **`true`** wird ein Aufruf an `average` ausgelöst, welches die abschließende Division- und Wurzeloperation durchführt.

```
void consume_apfixed_ui_sample(
   bool lastSample,
   uip_apfixed_traits::in_type usample,
   uip_apfixed_traits::in_type isample,
   UipPowerResult<uip_apfixed_traits> *result)
{
   static UipPowerResult<uip_apfixed_traits> result_;
   result_.addSample(usample, isample);
   if(lastSample) {
      *result = result_;
      result_.clear();
      result->average();
   }
}
```

Quelltext 4.5: Hauptfunktion der UIP-Berechnung

Daneben gibt es zum Vergleich für Kapitel 6 eine zusätzliche Funktionen, die hier nicht dargestellt ist, die zwei Arrays von jeweils 120 Abtastwerten erwartet. Die Funktion bekommt zwei Zeiger auf diese Strom- und Spannungsabtastwerte und übergibt jedes Abtastwertepaar in einer Schleife an `UipPowerResult::addSample`. Der Grund hierfür ist, dass eine Ausführung der UIP-Algorithmen auf der CPU anstatt auf dem FPGA ein Transfer von Abtastwerten aus dem FPGA in den Hauptspeicher bedingt und dieser Transfer aus Performanzgründen in Blöcken mehrerer Abtastwerten stattfindet (siehe 2.1).

4.2.2. Manuelle Eingriffe mit HLS Direktiven

Schwierigkeiten bereitet die Synthese für das Zynq 7010 der `average` Funktion von `UipLineResult<float>`, da es hierbei zu einem unverhältnismäßig starken Anstieg des Platzbedarfs auf dem FPGA kommt. Der Anstieg führt dazu, dass 163 % der verfügbaren LUTs benötigt werden und die Realisierung dieser Synthese deshalb nicht möglich ist. Eine nähere Untersuchung zu der Ursache ergab, dass die beiden Aufrufe von `average` innerhalb von `UipPowerResult<float>` von einer Inline-Optimierung expandiert werden. Damit eine erfolgreiche Synthese trotz des erhöhten Platzbedarfs für Fließkommaberechnungen dennoch unternommen werden kann, wird die HLS durch eine *Direktive* beauftragt, die Funktionsaufrufe nicht mehr zu expandieren. Durch diesen Eingriff kann das Problem aber nicht komplett gelöst werden, da die VHLS den HDL-Block, der die Abarbeitung von `average` innerhalb von `UipLineResult<float>` übernimmt, doppelt instantiiert. Der Grund hierfür ist, dass dadurch die Abarbeitung der beiden Aufrufe, die jeweils 67 Takte dauern, parallel erfolgen und die Gesamtzahl an benötigten Takten auf 80 reduziert werden kann. Erst durch den Einsatz einer weiteren Direktive, die das doppelte instantiieren verhindert, wird die Anzahl der benötigten LUTs auf 92 % reduziert.

```
1 #pragma HLS INLINE off
2 #pragma HLS ALLOCATION instances=average limit=1 function
```

Quelltext 4.6: HLS-Direktiven zum Verhindern der Inline-Optimierung. Das erste Pragma wird in Funktion `average` aus Listing 4.1, das zweite Pragma in Funktion `average` aus Listing 4.2 eingefügt.

In Zeile 1 wird die Expansion der beiden Aufrufe von `average` deaktiviert. Zeile 2 beschränkt die Anzahl der HDL-Funktionsblöcke für die nun nicht expandierten Aufrufe auf eine Instanz. Tabelle 4.3 fasst die Ergebnisse bezüglich des Ressourcenverbrauchs und der Latenz für das Zynq 7010 zusammen, wobei UIP-HLS-F0 die Variante bezeichnet, die aufgrund ihrer zu vielen LUTs nicht realisierbar ist. Die Variante UIP-HLS-F1 wurde daher um die angesprochenen HLS Direktiven erweitert. Die Variante UIP-HLS-T bezeichnet die Festkommaimplementierung ohne diese Direktiven. VHLS synthetisiert keine Architekturen mit Pipelining, es sei denn, es werden explizite Direktiven eingesetzt, die dies bewirken. Da hier aber untersucht wird, wie gut die Synthese auf unangepassten und weiterentwickelten Quelltext funktioniert, wird auf weitere Direktiven, die nicht zwingend erforderlich sind, verzichtet. Deshalb sind die Initiierungsintervalle, die hier nicht explizit angegeben sind, für alle Varianten jeweils einen Takt höher als deren Latenzen. In Abschnitt 7.2.2 wird diese weitergehende Optimierung untersucht und evaluiert.

Die Spalte f_{max} gibt die maximale Taktrate an, mit der das HLS-Syntheseergebnis auf dem Zynq-7010 betrieben werden kann. Zur Ermittlung dieses Wertes wurde die RTL-Netzliste auf die FPGA-Hardware platziert und verdrahtet. Wie unter Ab-

schnitt 2.4.1 beschrieben, wird VHDL mit einer Zielfrequenz von 100 MHz für die zu generierende VHDL-Architektur konfiguriert.

Tabelle 4.3.: Performanzkennwerte der UIP-Implementierungen für die HLS-Methodik auf dem Zynq-7010.

	DSPs	FFs	LUTs	Latenz		f_{max} (MHz)
(Total)	80	35 200	17 600	min	max	Ziel 100
UIP-HLS-F0	24	21 952	28799	11	77	n/a
UIP-HLS-F1	24	7936	9557	11	148	103
UIP-HLS-T	3	6660	6830	2	108	113
UIP-HLS-F-S	24	3075	4087	11	11	104
UIP-HLS-T-S	3	378	668	2	2	159

Die Latenz von UIP-HLS-F1 und UIP-HLS-T ist für den praktischen Einsatz für die Verarbeitung von Abtastwerten mit einer Abtastrate von 1,212121 MS/s zu hoch. Es müsste eine Latenz von maximal 82 Takten bei einer Taktlänge von 10 ns erzielt werden. Die Variante UIP-HLS-F0 erreicht eine Durchlaufzeit von $(77+1) \times 10$ ns $=$ 780 ns, kann jedoch aufgrund des zu hohen LUT-Bedarfs nicht verwendet werden. Die Latenz wird stark reduziert, wenn über die HLS nur aufsummiert wird. Die abschließende Durchschnittsbildung kann auf der CPU vorgenommen werden, zu der die Werte nach der Durchschnittsbildung ohnehin übertragen werden müssten. Da diese Übertragung nur am Ende eines Messzyklus passiert, und sich die Dauer eines Messzyklus im Bereich von Millisekunden bewegt, ist das die praktischere Vorgehensweise. Diese Implementierungen werden als UIP-HLS-F-S in der Float- und UIP-HLS-T-S in der Festkommavariante bezeichnet.

4.3. Polynominterpolation

Gegeben sind N Abtastwerte und es sollen mithilfe mehrerer Polynominterpolationen eine Zweierpotenz (im Folgenden $M = 2^q$) an Abtastwerten gewonnen werden, damit im Anschluss darauf eine FFT vorgenommen werden kann. Für diese Abtastratenkonvertierung erscheint zunächst naheliegend, das *zero padding*-Verfahren [24] mit einem nachgelagerten Tiefpassfilter einzusetzen. Dabei wird jedoch lediglich eine Abtastratenerhöhung um einen ganzzahligen Faktor realisiert, sodass die Abtastrate anschließend wieder um einen geeigneten, anderen ganzzahligen Faktor dezimiert werden muss, um die gewünschte Zielabtastrate zu erreichen. Dieses Verfahren scheidet zugunsten der Polynominterpolation aus, da die im Messgerät

geforderten Interpolationsfaktoren von z.B. 512/511 bei diesem Verfahren zu nicht praktikabel hohen Interpolations- und Dezimationsfaktoren führen können.

Eine Polynomfunktion (4.1) ist eine Linearkombination von Potenzen, wobei die Exponenten k natürliche Zahlen und die Koeffizienten a_k sowie t in dieser Arbeit reell sind. Die Potenzen t^k werden als Basis der Linearkombination, die Zahl n als *Grad* des Polynoms bezeichnet.

$$P(t) = \sum_{k=0}^{n} a_k t^k \tag{4.1}$$

Für die Polynominterpolation von Abtastwerten wird nach einem Polynom gesucht, das durch eine gegebene Anzahl von Abtastwerten verläuft, also $u_k = P(t_k)$, wobei u_k der Abtastwert und t_k der Abtastzeitpunkt ist. Die Zeitpunkte t_k werden *Stützstellen* genannt. Ein solches Polynom vom Grad/Ordnung n lässt sich durch die Aufstellung eines linearen Gleichungssystems eindeutig durch $n+1$ Abtastwerte $u_0, \ldots, u_n$ bestimmt, indem die Gleichungen jeweils nach a_k aufgelöst werden, z.B. durch das Gauß Eliminationsverfahren[13].

Falls die Abtastzeitpunkte im Voraus bekannt sind, kann eine günstigere Form der Linearkombination gewählt werden. Hier bietet sich die sogenannte *Lagrange*-Form [23] an (4.2).

$$L(t) = \sum_{k=0}^{n} u_k \ell_k(t) \tag{4.2}$$

$$\ell_k(t) = \prod_{\substack{0<=j<=n \\ j \neq k}} \frac{t - t_j}{t_k - t_j} \tag{4.3}$$

Die Funktion $\ell_k(t)$ nimmt bei t_k den Wert 1 und allen anderen Stützstellen $t_{j \neq k}$ den Wert 0 an. Damit verläuft das Polynom (4.2) an den Stellen t_k stets durch den Wert u_k. Damit für jeden Ergebniswert maximal viele Stützstellen verwendet werden können, wird für jeden der Werte jeweils eine neue Interpolation vorgenommen, wobei als Stützstellen die zeitlich um den Ergebniswert liegenden $n+1$ Eingabewerte verwendet werden (siehe Abbildung 4.1 im nächsten Abschnitt). Für die Masterarbeit wird ein Polynom der Ordnung 19 verwendet. Diese Zahl wurde durch eine Vorarbeit des Messgeräteherstellers festgelegt. Den 20 Stützstellen werden linear ansteigende Zeitpunkte zugeordnet, also $t_k = t_0 + kT$ mit $0 \leq k < n$, $T > 0$ und $T, t_0 \in \mathcal{R}$. Da die Abtastzeitpunkte nun im Voraus bekannt sind, können die Nenner $t_k - t_j$ innerhalb von ℓ_k vorausberechnet werden (im Folgenden wird das noch ausführlicher erklärt). Der genaue Wert von T und t_0 spielt für Quotienten aus

Gleichung (4.3) keine Rolle, da sich Zähler und Nenner durch Produkte der Form $(a - b)T$ darstellen lassen und sich T somit aus dem Bruch kürzen lässt. Für die Masterarbeit wurde $T = 1/19$ und $t_0 = -0.5$ gewählt.

4.3.1. Anschauliche Erklärung

In Abbildung 4.1 wird das Vorgehen anhand eines Sinussignals demonstriert. Sei N die Anzahl der Eingabeabtastwerte und M die Anzahl der Ergebniswerte. Der Algorithmus arbeitet vereinfachend nur mit ungeraden Ordnungen n, womit für jede Interpolation eine gerade Anzahl an Stützstellen zur Verfügung steht. Das Polynom für den Ergebniswert mit Index m ($0 \leq m < M$) verläuft durch die Eingabewerte mit Indices $r-n_h-1, \ldots, r-1, r, r+1, \ldots, r+n_h$, wobei $r = \lfloor N \cdot \frac{m}{M} \rfloor$ und $n_h = (n+1)/2$. Damit diese symmetrische Ausrichtung der Polynome um den Ergebniswert auch am Anfang und Ende funktioniert, wo ansonsten negative oder zu hohe Eingabeindices auftreten würden, werden der eigentlichen Eingabewertliste zusätzliche Abtastwerte angefügt, die zeitlich vor und nach den Eingabewerten abgetastet wurden.

Eine direkte Implementierung der Rechenvorschrift aus Gleichung (4.2) und (4.3) benötigt für jede Interpolation eine Summierung über $n + 1$ Terme und für jeden dieser $n + 1$ Terme ein Multiplikation über n Faktoren. Zwar können die n Quotienten $1/t_k - t_j$ für alle k und j vorausberechnet werden, wie im vorherigen Abschnitt erwähnt wurde. Das Ergebnis von $t - t_j$ ist aber nicht im Voraus bekannt, da sich der Ergebniswert an einem beliebigen Zeitpunkt zwischen den mittigen beiden Stützstellen befinden kann. Die Zeitkomplexität dieses Algorithmus beträgt also für M Ergebniswerte und $n + 1$ Polynom-Stützstellen $\mathcal{O}(Mn^2)$. Für $n = 19$ und $M = 131072 = 2^{17}$ werden zirka $50 \cdot 10^6$ einzelne Rechenoperationen benötigt.

Die Zahl der benötigten Rechenoperationen lässt sich um den Faktor n verbessern, wenn die Funktion $\ell_k(t)$ aus Gleichung (4.3) für alle k, aber nur bestimmte, diskrete t vorausberechnet wird. Da die Polynome stets so konstruiert werden, dass sich ein zu ermittelnder Ergebniswert zwischen den mittleren beiden Stützstellen befindet, und $\ell_k(t)$ daher nur für ein t zwischen diesen Stützstellen evaluiert wird, hält sich der Speicherbedarf in Grenzen. Dafür wird $\ell_k(t)$ für jedes k und t äquidistant im Intervall $t_{(n+1)/2-1} \leq t < t_{(n+1)/2}$ evaluiert und die Ergebnisse als konstante Koeffizienten im Programm abgelegt. Je feiner die Schrittweite von t ist, desto genauer wird die Approximation. In der Masterarbeit werden 512 Zwischenwerte berechnet. Die Herleitung der Zwischenwertanzahl stammt aus Vorarbeiten des Messgeräteherstellers. Insgesamt werden also $512(n + 1)$ Werte vorausberechnet. In Abbildung 4.2 wird die Interpolation für den zweiten Ergebniswert veranschaulicht, mit 64 statt 512 Zwischenwerten.

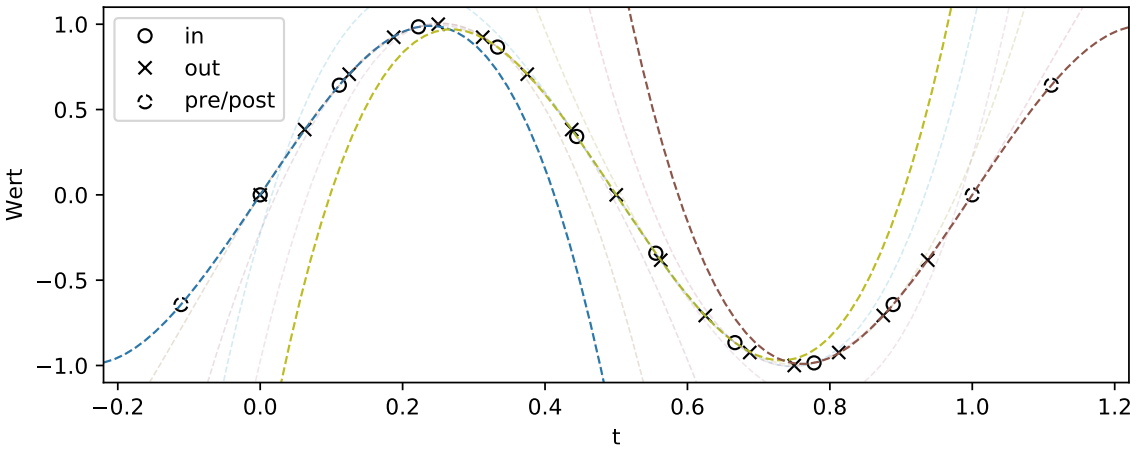

Abbildung 4.1.: Veranschaulichung der Abtastratenkonvertierung. Dargestellt ist ein Sinussignal, das mit neun Abtastwerten abgetastet wurde und auf 16 Abtastwerte konvertiert wird. Damit die Darstellung übersichtlich bleibt, werden Polynome der Ordnung drei verwendet. Die Ergebnispolynome des ersten (blau), achten (ocker) und letzten (braun) Ergebniswertes sind dargestellt, die übrigen Polynome sind abgeschwächt gezeichnet. Eingabewerte sind durch Kreise gekennzeichnet, Ergebniswerte durch Kreuze. Der erste und die letzten beiden Eingabewerte (*pre/post*) werden zusätzlich zu den neun Abtastwerten benötigt, damit die Polynome der Ordnung drei auch im ersten und letzten Abtastwert genügend Stützstellen besitzen.

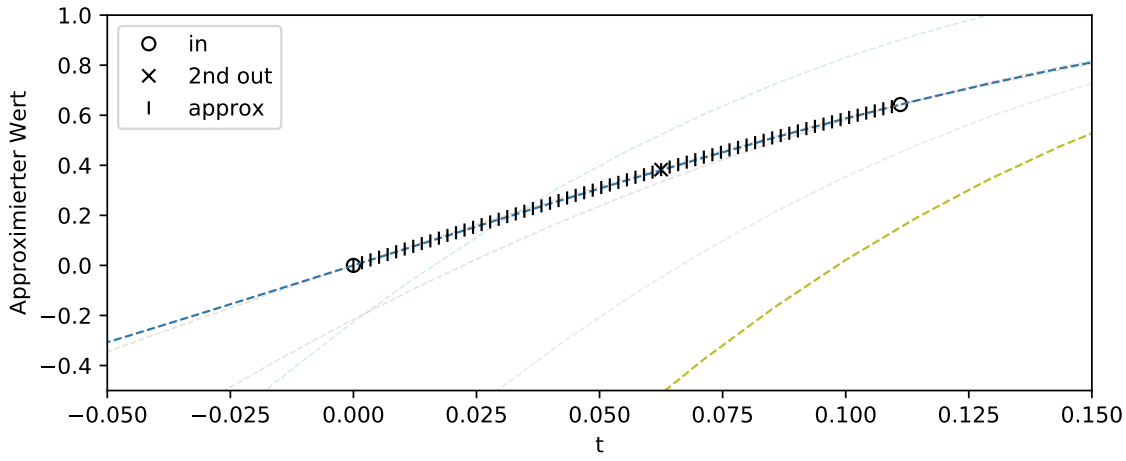

Abbildung 4.2.: Nähere Betrachtung des Bereiches um den zweiten Ergebniswert. Dargestellt sind 64 Zwischenpositionen. Zu jeder Zwischenposition t existiert ein Satz von $n = 20$ Koeffizienten, der die vorausberechneten Werte von $\ell_k(t)$ für $k = 0, \ldots, 19$ bereitstellt.

4.3.2. Quelltext

Wie bereits bei der Implementierung des UIP-Algorithmus, wird auch bei der Polynominterpolation eine Trennung der Berechnungslogik von Detailfragen zu zugrundeliegenden Repräsentationen von Eingabe- und Ergebniswerten vorgenommen. Die Traitschnittstelle für die Interpolation besteht aus einer Reihe von **`typedef`**s und Konstanten, die in Tabelle 4.4 dokumentiert sind.

Tabelle 4.4.: Traitschnittstelle für die Interpolation

Member	Kontrakt
`el_n`	Maximale Anzahl 2^q der Ergebniswerte.
`el_n_exp`	Der Exponent q. Momentan fixiert auf 17.
`coeff_type`	Der Typ der vorberechneten Werte von Gleichung (4.3).
`in_type`	Der Typ der Eingabewerte (Abtastwerte).
`resolution`	Anzahl der vorberechneten Zwischenpositionen.
`x_n`	Anzahl der Stützstellen.
`coeffs`	Die Tabelle der vorausberechneten Werte (siehe 4.3.2).
`filter_type`	Ergebnistyp von `filter_factor()`.
`filter_factor()`	Kontrolle des Quotienten $^{\text{Ergebniswerte}}/_{\text{Eingabewerte}}$ (siehe unten).

Die gesamte Interpolation wird durch das Klassentemplate `Interpolator` erledigt. Zunächst definiert dieses einige abgeleitete Typen und Konstanten. Die wichtigsten sind im Folgenden aufgelistet.

```
1  static const int el_n = Traits::el_n;
2  static const int el_n_exp = Traits::el_n_exp;
3  static const int x_n_half = Traits::x_n / 2 - 1;
4  static const int x_n_pre = x_n_half;
5  static const int x_n_post = x_n_half + 1;
6  static const int el_n_prepost = el_n + x_n_pre + x_n_post;
7
8  typedef Traits::in_type in_type;
9  typedef IntegerCounter<in_type, el_n_exp + 1>::type el_n_exp_type;
10 typedef IntegerCounter<in_type, el_n>::type el_n_type;
11 typedef IntegerCounter<in_type, el_n_prepost>::type el_idx_type;
12 typedef RealCounter<in_type, el_n_prepost>::type el_idx_realtype;
13
14 typedef Traits::coeff_type coeff_type;
15 typedef Multiply<in_type, coeff_type>::type multiplied_type;
16 typedef Accumulate<multiplied_type, Traits::x_n>::type accu_type;
17 typedef PowerOf2Table<el_n_type> powers_of_2;
```

Quelltext 4.7: Einige der Konstanten und Typen für die Polynominterpolation

Tabelle 4.5.: Hinzukommende Meta-Funktionen für die Interpolation

Funktion	Ergebnis
`RealCounter<T,N>`	Ein Fließ- oder Festkommatyp, der N repräsentieren kann. Diese Meta-Funktion wird im Zusammenspiel mit `Divide` zur Berechnung von $1/N$ gebraucht.
`Multiply<T1,T2>`	Typ, der das Ergebnis von $\texttt{t1} \cdot \texttt{t2}$ für beliebige Werte $\texttt{t1} \in \texttt{T2}, \texttt{t2} \in \texttt{T2}$ repräsentieren kann.
`Reciprocal<T1, T2>`	Typ, der $1/\texttt{t1}$ für beliebige $t1 \in \texttt{T1} \subset \mathcal{N}^+$ repräsentieren kann, wobei die Anzahl an Ergebnis-Dezimalstellen (ohne führende Nullen) gleich denen von $t2 \in T2 \subset \mathcal{Q}$ ist. Z.B. kann `Reciprocal<ap_uint<2>, ap_fixed<4, ...>>::type` den Wert 0.01111_2 repräsentieren. Die Metafunktion wird verwendet, um damit $\texttt{t2}/\texttt{t1}$ in $\texttt{t2} \cdot 1/\texttt{t1}$ zu ersetzen.
`PowerOf2Table<T>`	Eine Liste von Zweierpotenzen 2^k, mit $0 \leq k \leq 17$. Der Parameter `T` kann entweder `int` oder `ap_uint` sein (siehe Abschnitt 2.5).

Der Typ `accu_type` stellt dabei den Ergebnistyp der Interpolation dar (engl. *accumulate*, aufsummieren), der sich aus dem Ergebnis der Multiplikation $u_k \ell_k(t)$ (Gleichung (4.2)) ableitet. Es werden auch neue Metafunktionen definiert, die in Tabelle 4.5 aufgelistet sind. Die Liste `powers_of_2` von Zweierpotenzen kann prinzipiell auch direkt lokal innerhalb der `interpolate` Funktion angelegt werden. Das Auslagern in eine Tabelle erlaubt lediglich die Wiederverwendung dieser Werte durch andere Algorithmen.

Den Einsprungpunkt für Benutzer bildet die Funktion `interpolate`, die als Parameter die Abtastwerte (`in`), deren Anzahl (`count`), sowie ein Array für die Ergebniswerte (`out`) erhält. Zurückgegeben wird die Anzahl der Ergebniswerte. Zunächst wird ermittelt, auf wie viele Werte interpoliert werden muss. Da die Interpolation nur bis zu einer Ergebniswert-Anzahl von 2^{17} Werte arbeitet (diese Zahl ergibt sich aus der maximalen Abtastrate von ca. $150kS/s$ und einer minimal unterstützten Signalfrequenz von 1 Hz, siehe Abschnitt 2.1), kann hier eine einfache Iteration verwendet werden. Zugunsten einer einfacheren Implementierung wurde auf trickreiche Bitmaskierungen oder Standardbibliotheksfunktionen für die Errechnung des Zweierlogarithmus verzichtet und stattdessen diese Schleifenform gewählt.

```
1  static el_n_type interpolate(in_type in[el_n_prepost],
2                               el_n_type count,
3                               accu_type out[el_n])
4  {
5     const el_n_exp_type max_exp = el_n_exp;
6     auto count_filtered = Traits::filter_factor() * count;
7     for(el_n_exp_type e = 0; e <= max_exp; ++e) {
8        if(powers_of_2::values[e] >= count_filtered) {
9           interpolateToPowerOf2(in, count, powers_of_2::values[e], out);
10          return powers_of_2::values[e];
```

```
11        }
12     }
13     interpolateToPowerOf2(in, count,
14                           powers_of_2::values[max_exp], out);
15     return powers_of_2::values[max_exp];
16  }
```

Quelltext 4.8: Ermittlung der Anzahl an auszugebenden Ergebniswerten und Aufruf der eigentlichen Interpolation

Das Trait kann über die Funktion `filter_factor()` beeinflussen, um welchen Faktor sich die Ergebniswertanzahl von der Eingabewertanzahl unterscheidet. Liefert `filter_factor()` beispielsweise $0,5$, wird der Interpolator die Abtastwerte immer auf die nächstkleinere Zweierpotenz abrunden, anstatt auf die nächstgrößere Zweierpotenz aufrunden. Dies kann nützlich sein, wenn ein zuvor angewandter Tiefpass-Filter hochfrequente Signalfrequenzanteile aus den Abtastwerten entfernt hat. Ohne Risiko, das Abtasttheorem zu verletzen, kann der Interpolator dann also die Abtastrate verringern, anstatt sie zu erhöhen.

```
1  static void interpolateToPowerOf2(in_type in[el_n_prepost],
2                                    el_n_type count,
3                                    el_n_type outcount,
4                                    accu_type out[el_n]) {
5     typedef Reciprocal<el_n_type, el_idx_realtype>::type recip_type;
6     recip_type recip = recip_type(1.0) / outcount;
7     el_idx_type count_zeroext = count, x_n_pre_zeroext = x_n_pre;
8     for(el_n_type i = 0; i < outcount; ++i) {
9        auto rpos = recip * i;
10       auto spos = rpos * count_zeroext + x_n_pre_zeroext;
11
12       el_idx_type sposi = el_idx_type(spos);
13       el_idx_type sposi_strt = sposi - x_n_half;
14
15       resolution_type coeffposi = (spos - sposi) *
16           resolution_type(Traits::resolution);
17       auto interpol = interpolateAroundSample(in, count, sposi_strt, coeffposi);
18       out[i] = interpol;
19    }
20 }
```

Quelltext 4.9: Interpolationsfunktion mit Schleife über Ergebniswerte

Die Funktion `interpolateToPowerOf2` enthält die Hauptschleife über alle Ergebniswerte, und berechnet für jeden Ergebniswert den Index des links davon liegenden Eingabewertes. Die Indexdistanz $t_\Delta = (\texttt{spos} - \texttt{sposi})$ zum linken Nachbarn ($0 \leq t_\Delta < 1$) bestimmt das approximierende `coeffposi` (siehe Abbildung 4.2). Damit kann Gleichung (4.3) bereits ausgewertet werden, denn

$$\ell_k(t) \approx \texttt{Traits} :: \texttt{coeff}[\texttt{coeffposi}][\texttt{k}]$$

Die Funktion `interpolateAroundSample` berechnet nun das Vektor-Skalarprodukt

$$\sum_{k=0}^{19} \texttt{in}[\texttt{sposi\_strt} + \texttt{k}] \cdot \texttt{Traits} :: \texttt{coeff}[\texttt{coeffposi}][\texttt{k}]$$

4.3.3. Manuelle Eingriffe mit HLS Direktiven

Die Performanzkennwerte der HLS für das Zynq 7010 sind in Tabelle 4.6 angegeben. Mit Poly-HLS-F und Poly-HLS-T werden jeweils die Fließ- und Festkommavarianten bezeichnet. Die Latenz hängt von der Anzahl an Ergebniswerten ab. Da die Implementierung auf einem Block von vielen Abtastwerten arbeitet, ist es sinnvoll, die Hauptschleife innerhalb der Funktion `interpolateToPowerOf2` auf einer Pipeline abzuarbeiten. Diese Implementierungen sind in der Tabelle mit einem „P“-Suffix gekennzeichnet.

Tabelle 4.6.: Performanzkennwerte der Polynominterpolation, mit und ohne Pipeline.

	BRAMs (18Kb)	DSPs	FFs	LUTs	Latenz		f_{max} (MHz)
(Total)	120	80	35 200	17 600	min	max	Ziel 100
Poly-HLS-F	32	16	2002	2072	290	33 816 770	104
Poly-HLS-T	18	5	838	331	46	8 912 978	155
Poly-HLS-F-P	40	25	4055	4397	109	1 310 981	105
Poly-HLS-T-P	16	19	986	915	47	1 179 739	135

5. Zielplattformoptimierte Implementierungen

Hier werden zielplattformoptimierte Varianten der bereits entwickelten *portablen* Implementierungen vorgestellt. Hierdurch soll herausgefunden werden, wie hoch die Performanzverluste der portablen Implementierungen gegenüber den optimierten Varianten aus diesem Kapitel sind. Zunächst werden die Implementierungen beschrieben, die dann in Kapitel 6 mit den Varianten aus Kapitel 4 verglichen werden. Dabei werden zwei Intel und ARM CPUs, sowie ein FPGA als Zielplattformen verwendet. Diese wurden in Kapitel 2 beschrieben.

Als Softwarecompiler kommen beim Intel CPU der GNU GCC Compiler in Version 7.2.1 und Intel ICC in Version „17.0.4 20170411“ zum Einsatz. Für den ARM CPU wird der GNU GCC Compiler in Version 6.3.1 verwendet. In diesem Kapitel werden einige wichtige verwendete Compileroptionen beschrieben. Alle weiteren Kommandozeilenparameter sind im Anhang B aufgeführt.

Das Vorgehen in diesem Kapitel ist es, zunächst den Quelltext derart zu ändern, dass er von Prozessoren prinzipiell schneller ausgeführt werden kann. Dann wird überprüft, inwiefern die Compiler den Quelltext bereits selbstständig optimieren können und ob ein manuelles Eingreifen erforderlich ist, um bessere Ergebnisse zu erzielen. Schlussendlich wird eine Implementierung der Algorithmen mit handgeschriebenen VHDL-Quelltext vorgestellt.

5.1. CPU-freundlichere Quelltextstruktur

Zunächst wird die generische Implementierung der UIP-Berechnung etwas umgeformt. Bei der UIP-Implementierung wird jeweils ein Abtastwert konsumiert und direkt auf das Ergebnis addiert (Listing 4.5). Für CPU-Implementierungen ist diese Vorgehensweise gegenüber der HLS-Implementierung nicht fair, da der CPU-Quelltext die Abtastwerte von der Hardware in Blöcken mehrerer Abtastwerte erhält. Die erste Änderung ist deshalb, dass die CPU-Ausführung ein Array von 120 Abtastwerten erhält und in einer einzigen Schleife akkumuliert (vgl. Abschnitt 2.1). Da es sich hierbei nur im eine kleine Erweiterung handelt, die sich nicht auf den

Quelltext der eigentlichen UIP-Implementierung auswirkt, wird beim Vergleich in Kapitel 6 diese Block-Version verwendet, um die Performanz der CPU-Ausführung im nicht optimierten Szenario zu ermitteln.

Zusätzlich zu dieser blockweisen Verarbeitung der Abtastwerte ist es sinnvoll, die 120 Abtastwerte separat zu akkumulieren und das Ergebnis auf die laufende **`double`**-Summe zu addieren, da diese 120 Werte dann mit **`float`** *Vektor-Instruktionen* akkumuliert werden können (siehe Kapitel 2). Mit **`float`**-Werten zu arbeiten hat zwei Vorteile. Cortex-A9 SIMD-Instruktionen können nur **`float`**-Werte parallelisiert verarbeiten. Außerdem können mit **`float`** bei gleicher SIMD-Registerbreite doppelt so viele Werte verarbeitet werden wie mit **`double`**. Diese Änderung der UIP-Implementierung, zunächst eine Zwischensumme in **`float`** zu bilden, ist eine dedizierte Optimierung hinsichtlich einer CPU-Architektur und wird deshalb in Kapitel 6 als optimierte Implementierung angesehen.

Für die Zwischensumme mit **`float`** muss geprüft werden, ob und wie viele signifikante Bits der Abtastwerte durch die Akkumulation verworfen werden. Falls SIMD Instruktionen zum Einsatz kommen, die vier Werte parallel verarbeiten, werden vier Summenreihen mit jeweils $120/4 = 30$ Termen im Typen **`float`** berechnet, deren Endergebnisse danach auf eine Summe reduziert wird. Da die Eingabewerte 18 Bit haben und **`float`** eine Präzision von 24 Bit bietet, können im schlechtesten Fall, wenn alle Eingabewerte maximal groß sind, bis zu $2^{24-18-1} = 2^5$ Werte addiert werden, bevor der **`float`**-Typ für die danach folgenden Additionen die niedrigstwertigen Bits der Summe nicht mehr repräsentieren kann. Da die vier Reihen jeweils 30 Terme haben, bewirkt eine Zwischenakkumulation von 120 **`float`** Werten also keinen Genauigkeitsverlust.

Die folgenden beiden Listings veranschaulichen den veränderten Kontrollfluss. Die Zwischenwerte `mU`, `mI` und `mP` stellen jeweils eine Akkumulation der 120 Abtastwerte unter Verwendung von **`float`** dar. Diese werden nach der Aufsummierung auf die **`double`** Endsumme addiert.

```
1  for(int idx = 0;
2      idx < 120;
3      idx++) {
4    r->addSample(u[idx],
5                 i[idx]);
6  }
```

Quelltext 5.1: Ungünstige Akkumulation auf **`double`**-Summe

```
1  for(int idx = 0;
2      idx < 120;
3      idx++) {
4    mU->addSample(u[idx]);
5    mI->addSample(i[idx]);
6    mP->addSample(u[idx], i[idx]);
7  }
8  r->u.addIntermediate(&mU);
9  r->i.addIntermediate(&mi);
10 r->p.addIntermediate(&mP);
```

Quelltext 5.2: Sinnvollere Zwischenakkumulation auf **`float`**-Summe

5.2. Automatisch vektorisierte Versionen

In diesem Abschnitt werden automatisch vektorisierte Implementierungen beschrieben. Bei einer automatischen Vektorisierung handelt es sich um eine Compileroptimierung, die einen auf Skalarwerten operierenden Zwischencode transformiert, sodass der resultierende Code auf Vektoren mit SIMD Instruktionen operiert.

Damit ein C++-Compiler eine automatische Vektorisierung vornehmen kann, müssen gewisse Abstriche bei der IEEE754 [16] Konformität gemacht werden. Da eine Vektorisierung die Reihenfolge der Additionen verändert, muss durch den Compiler angenommen werden können, dass die `float`-Addition assoziativ ist, entgegen den Regeln der Fließkommaarithmetik. Deshalb wird dem GCC-Compiler die Option `-funsafe-math-optimizations` übergeben. Für den Intel-Compiler sind keine speziellen Parameter erforderlich, da der Compiler automatisch ein Fließkommamodell wählt, das schnell, aber nicht konform ist. Experimente mit der Option `-fp-model=precise` zeigen, dass dann auch der Intel-Compiler auf eine Vektorisierung verzichtet.

5.2.1. Automatisch vektorisierte UIP-Berechnung

Für die UIP-Implementierung gibt es einen Problemfall bei der Autovektorisierung von Ausdrücken der Form `a < b ? a : b`, die bei der Berechnung von Minimum und Maximum zum Einsatz kommen. Vergleiche mit NaN (Not a Number) liefern laut IEEE754[16] immer `false`, sodass die Auswertung von `a < b ? a : b` mit $a = \text{NaN}$ immer `b` liefert. Bei beiden hier untersuchten Prozessoren liefert die Minimum- und Maximumbildung bei der Verwendung von SIMD, falls einer der Operanden NaN ist, allerdings immer NaN. Die SIMD-Semantik ist also inkompatibel mit der Semantik in C++ unter Verwendung der IEEE754. Für die Berechnungen von Minimum und Maximum verwendet GCC daher keine Vektorinstruktionen. Um dennoch eine Vektorisierung zu erreichen, kann die GCC-Option `-ffinite-math-only` verwendet werden, die angibt, dass keine NaN oder $\pm$ inf Werte im Programm vorkommen. Bei der UIP-Berechnung ist dies der Fall.

Nachdem diese Option verwendet wird, können beide Compiler, GCC und Intel ICC, die UIP-Berechnung vektorisieren. Überprüft wurde das anhand eines Disassemblers, mit dem die jeweiligen Vektorinstruktionen für Addition, Multiplikation, Minimum und Maximum an den erwarteten Stellen gefunden wurden.

5.2.2. Automatisch vektorisierte Polynominterpolation

Die Polynominterpolation ließ sich für beide Prozessoren problemlos automatisch vektorisieren. Dafür wurden zunächst sämtliche Metafunktionen für die Ableitung

von Zwischentypen entfernt und durch `float` ersetzt. Der ursprünglich generischen Implementierung entsteht durch die Verwendung der Metafunktionen ein Nachteil, da diese konservativ arbeiten und daher für sämtliche Akkumulationen, selbst wenn es sich dabei nur um 20 Elemente handelt, den Typen `double` liefern.

Da sich der vektorisierte Datenpfad der UIP-Berechnung über die gesamte Implementierung erstreckt, wurde entschieden, dass für die UIP-Implementierung eine manuell vektorisierte Implementierung erstellt wird. Die nächsten beiden Abschnitte beschreiben diese Vektorisierung für beide Prozessoren.

5.3. Intel Atom E3845

Bei der Kompilierung für den Intel-Prozessor wird dem GCC-Compiler der Parameter `-march=slm` übergeben, das „slm“ steht für „Silvermont“. Dem Intel Compiler wird der Parameter `-mtune=silvermont` übergeben. Die weiteren Optionen sind in Abschnitt 5.2, Abschnitt 5.2.1 und Anhang B beschrieben.

Explizit vektorisierte UIP-Berechnung

Hier sind die Initialisierung und Aufsummierung relevant, da sich diese Teile gegenüber der generischen Implementierung geändert haben.

```
1  void clear() {
2     sqsum = _mm_set_ps1(0.f);
3     dcsum = _mm_set_ps1(0.f);
4     pkp = _mm_set_ps1(-std::numeric_limits<float>::max());
5     pkn = _mm_set_ps1(+std::numeric_limits<float>::max());
6  }
7  void addSamplesx4(__m128 s) {
8     sqsum = _mm_add_ps(sqsum, _mm_mul_ps(s, s));
9     dcsum = _mm_add_ps(dcsum, s);
10    pkp = _mm_max_ps(pkp, s);
11    pkn = _mm_min_ps(pkn, s);
12 }
```

Quelltext 5.3: Berechnung von Mittelwerten, sowie Minimum und Maximum mit SSE

Dabei wird die portable *Instrinsics* API [19] von Intel verwendet, die von mehreren Compilern unterstützt wird. Dabei bedeuten die Suffixe „ps“ jeweils *packed single*, da IEEE754 single-precision Fließkommazahlen verarbeitet werden. Die Funktion `_mm_set_ps1` erzeugt einen Vektor mit dem gleichen übergebenen Wert für alle vier Elemente. Die Ergebnisvektoren müssen nach der Akkumulation jeweils auf einen einzigen `float` reduziert werden. Dafür werden jeweils die beiden ersten und beiden letzten Vektorelemente addiert. Die Funktion `_mm_hadd_ps` führt diese Operation wiederum auf zwei übergebene Vektoren aus. Für den zweiten Vektor wurde ein

Dummy-Vektor übergeben. Beide Summen des ersten Vektors werden danach noch einmal addiert, um die Endsumme zu erhalten. Als Beispiel sei hier die Reduzierung für `sqsum` angegeben.[1]

```
1 void addIntermediate(const UipLineResultIntermediate *other) {
2    ...
3    __m128 sqsum2 = _mm_hadd_ps(other->sqsum, _mm_set_ps1(0.f));
4    sqsum += sqsum2[0] + sqsum2[1];
5    ...
6 }
```

Quelltext 5.4: Reduktion der vier SIMD Partialsummen in eine Endsumme mit SSE

5.4. ARM Cortex-A9

Der eingesetzte GNU GCC Compiler erzeugt ausführbaren Code für die *arm-linux-gnueabihf* Zielplattform (diese dreiteiligen Namen werden auch als *target triplet* bezeichnet). Dabei bedeutet *gnueabihf*, dass als Zielplattform ein Linux-System mit entsprechenden Bibliotheken, anstelle eines Bare-Metal-Systems, verwendet wird. Der Suffix *hf* bedeutet, dass der Cross-Compiler annimmt, dass das Zielsystem einen Prozessor mit Hardware-Fließkommasupport besitzt. ARM stellt auch einen proprietären Compiler auf Basis des LLVM-Frameworks bereit, den *ARM Compiler*. Dieser wird aber wegen einer fehlenden Lizenz hier nicht behandelt.

Dem GCC Compiler werden die Optionen `-mcpu=cortex-a9` und `-mfpu=neon` übergeben. Die weiteren Optionen sind auch für den ARM-Prozessor in Abschnitt 5.2, Abschnitt 5.2.1 und Anhang B beschrieben.

Explizit vektorisierte UIP-Berechnung

Zunächst wird die Akkumulation der `float` Zwischenwerte aufgelistet. Auch hier sind die Initialisierung und die Aufsummierung relevant, da sich diese Teile gegenüber der generischen Implementierung geändert haben.

```
1 void clear() {
2    sqsum = vmovq_n_f32(0.f);
3    dcsum = vmovq_n_f32(0.f);
4    pkp = vmovq_n_f32(-std::numeric_limits<float>::max());
5    pkn = vmovq_n_f32(+std::numeric_limits<float>::max());
6 }
7
```

[1] Hier wäre es auch möglich gewesen, mit `_mm_store_ps` den Vektor zunächst in ein normales `float[4]`-Array zu exportieren, um dieses dann explizit zu reduzieren. Diese Möglichkeit wurde aber erst später entdeckt und wird in der Implementierung deshalb nicht verwendet.

```
8  void addSamplesx4(float32x4_t s) {
9     sqsum = vmlaq_f32(sqsum, s, s);
10    dcsum = vaddq_f32(dcsum, s);
11    pkp = vmaxq_f32(pkp, s);
12    pkn = vminq_f32(pkn, s);
13 }
```

Quelltext 5.5: Berechnung von Mittelwerten, sowie Minimum und Maximum mit ARM NEON

Alle vier Zwischenwerte sind dabei Vektoren von vier `float` Werten. Die Funktion `vmovq_n_f32` initialisiert die Vektoren mit dem gleichen übergebenen Wert für jedes Element. Die eigentlichen Operationen sind jeweils *Vector Multiply-Accumulate*, *Vector Add*, *Vector Max* und *Vector Min*. Der Suffix *q* steht dabei für *quad* und beschreibt, dass vier Werte verrechnet werden sollen. Auch hier müssen die Ergebnisvektoren nach der Akkumulation auf einen einzigen `float` reduziert werden. Dafür werden jeweils die beiden ersten und beiden letzten Vektorelemente addiert. Beide Summen werden danach noch einmal addiert, um die Endsumme zu erhalten. Als Beispiel sei hier die Reduzierung für `dcsum` angegeben.

```
1  void addIntermediate(const UipLineResultIntermediate *other) {
2     ...
3     float32x2_t dcsum2 =
4        vpadd_f32(vget_high_f32(other->dcsum),
5                  vget_low_f32(other->dcsum));
6     dcsum += dcsum2[0] + dcsum2[1];
7     ...
8  }
```

Quelltext 5.6: Reduktion der vier SIMD Partialsummen in eine Endsumme mit ARM NEON

5.5. Xilinx Zynq-7010 SoC-FPGA

Damit der Ressourcenbedarf und die Laufzeiteffizienz der HLS-Implementierungen beurteilt werden können, beschreibt dieser Abschnitt handgeschriebene VHDL-Lösungen der implementierten Algorithmen. Es wurden keine öffentlich verfügbaren Implementierungen gefunden, sodass eine Eigenentwicklung erforderlich war. Der Autor musste sich hierfür zunächst in die VHDL-Beschreibungssprache einarbeiten. Dieser Sachverhalt wird in Kapitel 8 berücksichtigt, wenn die Vergleichsergebnisse aus Kapitel 6 interpretiert werden.

Um die VHDL-Beschreibungen zu testen, wurden die entwickelten Module mithilfe des sogenannten Vivado IP-Integrators [42] in eine Architektur integriert, die dann erfolgreich synthetisiert, implementiert und auf das Zynq 7010 programmiert werden konnte. Das Vorgehen dabei war, zunächst mit dem IP-Integrator ein sogenanntes Block-Diagramm mit verschiedenen IP-Cores (Intellectual Properties) zu erstellen.

Dabei genügt es, einen sogenannten *Processing-System* IP-Core [47] in das Diagramm zu ziehen und eine sogenannte *Block-Automation* auszuführen. Dabei werden verschiedene Ports erstellt, die beim Zynq für eine Implementierung erforderlich sind (z.B. für das Taktsignal). Der Takt wurde von der Zynq-Taktquelle `FCLK_CLK0` bezogen und auf dem Standardwert von 125 MHz des eingesetzten Entwicklerboards belassen, damit kein neuer Bootloader zum Konfigurieren der Taktquelle installiert werden musste[2]. Für dieses Block-Diagramm wurde dann ein sogenannter *HDL-Wrapper* erzeugt, der in ein manuell erstelltes Hauptmodul instantiiert wurde. Die zuvor automatisch erstellten Ports des HDL-Wrappers wurden weitergereicht und teilweise aus dem Hauptmodul als I/O Ports herausgeführt. In dieses Modul wird dann auch das entwickelte HDL-Modul der eigentlichen Implementierung instantiiert. Für die UIP-Berechnung und Polynominterpolation werden jeweils noch weitere IP-Cores benötigt, die in den nächsten Abschnitten beschrieben sind.

5.5.1. Hardwarebeschreibung der UIP-Berechnung

Die Entwicklung der UIP-Implementierung im VHDL-Quelltext wurde inklusive Simulation und Test auf dem FPGA innerhalb von etwa drei Wochen fertiggestellt. Diese Zeit schließt die Einarbeitung in die Methodik der Hardwarebeschreibung mit VHDL ein. Zum Einsatz kommen IP-Cores von Xilinx für die Berechnung der Quadratwurzel und Division. Zusätzlich werden für die Multiplikationen und Akkumulationen DSPs verwendet. In den Abbildungen 5.1, 5.3 und 5.4 sind vereinfachte Schaltpläne dargestellt. Diese Diagramme sind nur eine funktionale Übersicht und lassen weniger wichtige Details aus. Z.B. ist das UIP-Average-Modul tatsächlich das Hauptmodul, welches alle anderen Module enthält. Im Diagramm wird aber nur derjenige Teil dargestellt, der gegenüber dem nicht-durchschnittsbildenden UIP-Modul hinzugekommen ist.

Die korrekte Funktionsweise wurde mithilfe des Vivado Simulators und Vivado Virtual In/Out Moduls verifiziert. Letzteres ist ein IP-Core, der es erlaubt, über die JTAG (Joint Test Action Group) Schnittstelle beliebige Werte an zuvor ausgewählte Pins anzulegen, sowie auf den PC zu übertragen.

[2] Für den Vergleich mit den HLS-Lösungen in 6.2 wurde der Takt für Synthese, Platzierung und das Routing auf 100 MHz herabgesetzt.

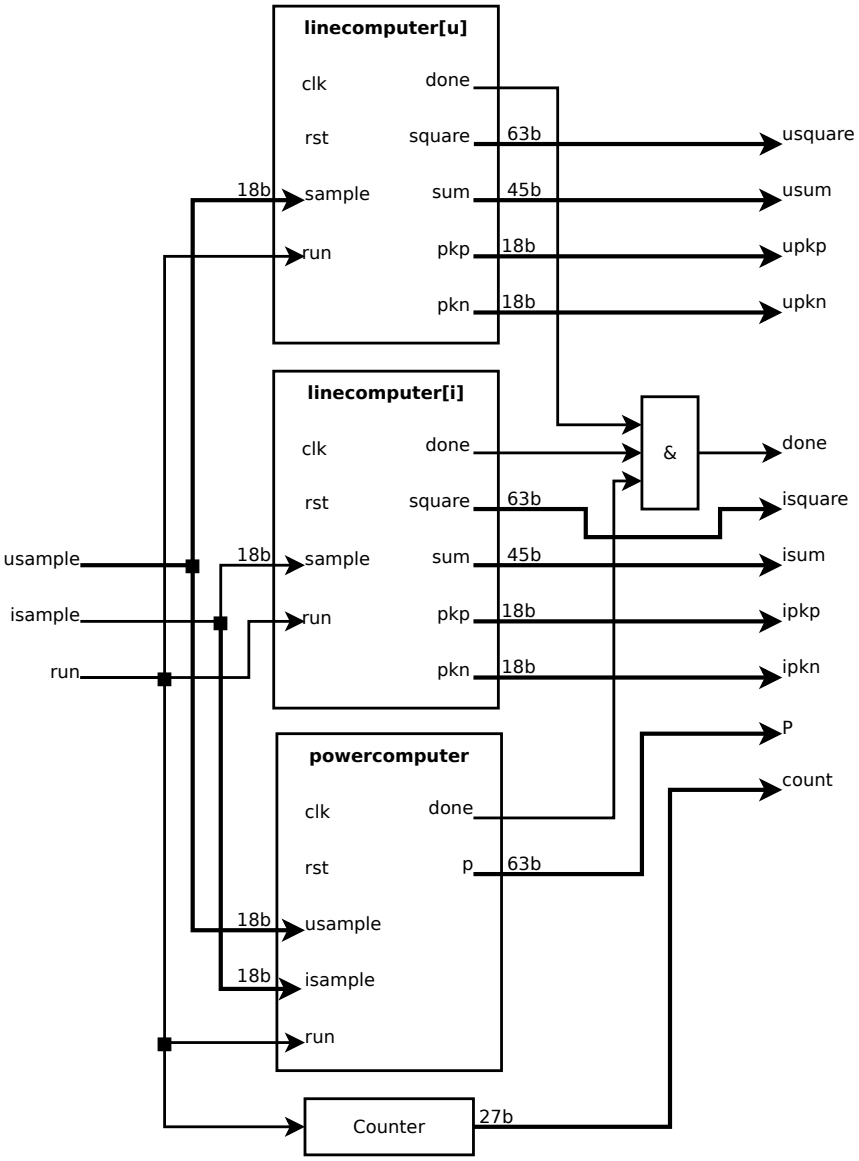

Abbildung 5.1.: Vereinfachtes Blockschaltbild des UIP-Moduls. Die Blockschaltbilder für den Linecomputer, Powercomputer sowie die abschließende Durchschnittsbildung sind separat dargestellt.

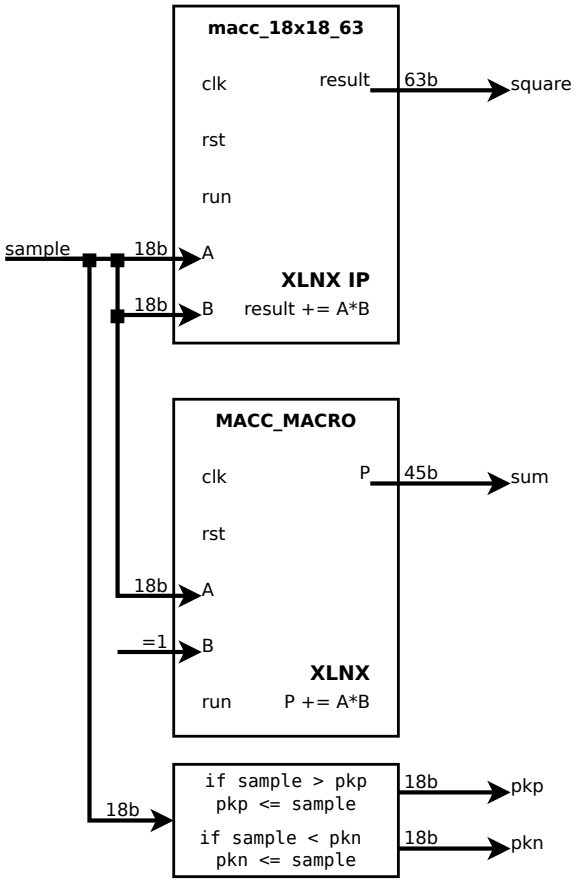

Abbildung 5.2.: Vereinfachtes Blockschaltbild des Linecomputer-Moduls

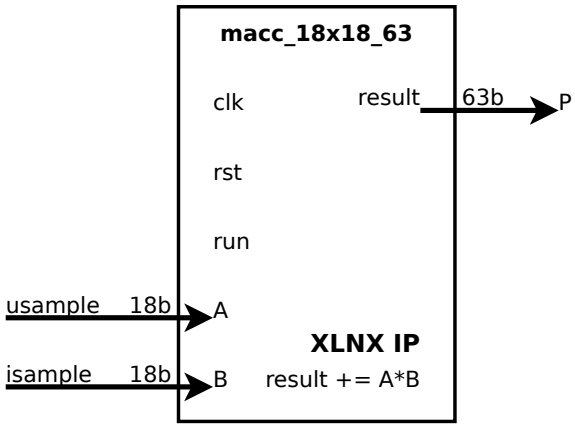

Abbildung 5.3.: Vereinfachtes Blockschaltbild des Powercomputer-Moduls

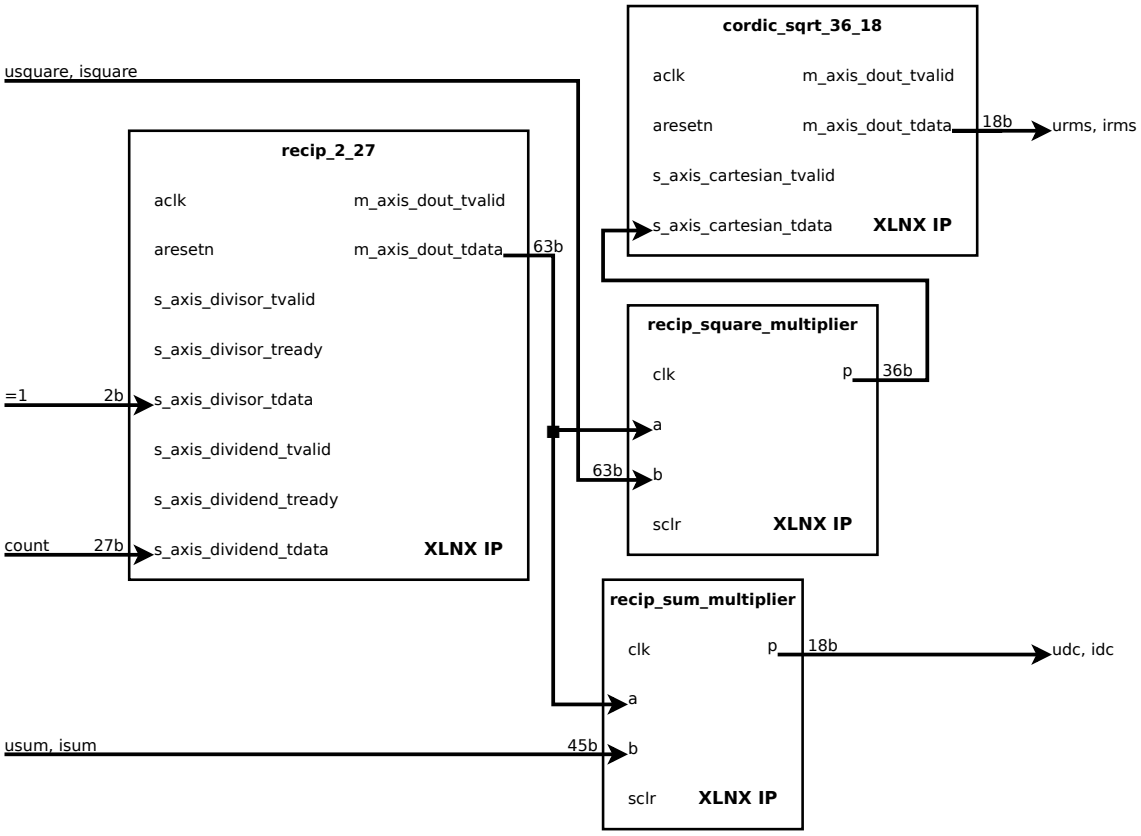

Abbildung 5.4.: Vereinfachtes Blockschaltbild des UIP-Average Moduls. Leitungen mit mehrfacher Beschriftung werden im Zeitmultiplexverfahren angesteuert, um Ressourcen zu sparen.

5.5.2. Hardwarebeschreibung der Polynominterpolation

Die Entwicklung der Polynominterpolation mit VHDL-Quelltext wurde inklusive Simulation und Test auf dem FPGA innerhalb von drei Wochen fertiggestellt. Die Implementierung lässt sich in die folgenden Aufgabenbereiche unterteilen (siehe Abschnitt 4.3.1 und Abbildung 4.2 für die Bedeutung der Koeffizienten)

1. Ein Modul zum Berechnen des Skalarprodukts von zwei 20-elementigen Vektoren mit jeweils 18 Bit. Die Eingangsvektoren bestehen aus 20 Eingabeabtastwerten und 20 Koeffizienten, das Ergebnis liefert den nächsten Ausgabewert. Dies entspricht der Funktion `interpolateAroundSample` aus Listing 4.9.

2. Ein Schieberegister mit 20 Werten von jeweils 18 Bit. Das Register speichert die Eingabeabtastwerte, die als nächstes an das Skalarproduktmodul übergeben werden.

3. Ein Block-RAM mit 512 Wörtern von jeweils 20×18 Bit. Diese Wörter speichern je 20 Koeffizienten. Für jedes Skalarprodukt wird von Adresse $512 \cdot {}^{Z}/_{M}$ ein Wort mit 20 Koeffizienten gelesen (die Division kann als Bitshift implementiert werden, da M eine Zweierpotenz ist – siehe nächster Absatz).

4. Einem Taktteiler mit rationalem Teilungsverhältnis ${}^{N}/_{M}$ (N, M ist jeweils die Eingabe- und Ausgabewertanzahl). Am Anfang und mit jedem Ausgabewert wird ein synchroner Taktimpuls angelegt. Der Teiler hält einen Zähler Z und setzt mit jedem Impuls $Z = Z + N$. Er erzeugt einen synchronen Ausgabeimpuls, falls $Z \geq M$, woraufhin er $Z = Z - M$ setzt. Mit jedem Ausgabeimpuls wird das Schieberegister der Eingabewerte eine Stelle weitergeschoben und ein neuer Eingabewert gelesen.

5. Ein Modul, das die Eingangswerte aus einem RAM-Block serialisiert an das Schieberegister weitergibt und ein anderes Modul, das die Ergebniswerte aus dem Skalarprodukt seriell empfängt und in einen anderen RAM-Block schreibt.

Die Koeffizienten haben die gleichen Werte und wurden mit dem gleichen Programm erzeugt wie bei der Softwareimplementierung. Aber anstatt die Koeffizienten formatiert für die Verwendung im Quelltext auszugeben, wurden mithilfe von `ap_fixed<18, 1>` und `ap_uint<512>` alle Koeffizienten binär aneinandergehängt und das Ergebnis der Vivado Konfigurations-GUI für den Block-RAM übergeben. Die Struktur der Implementierung ist ähnlich der eines digitalen FIR-Filters, wofür es von Xilinx auch einen fertigen IP-Core gibt. Für diesen Core konnte aber keine Möglichkeit gefunden werden, eine Abtastwertverzögerung mit rationalem Teilungsverhältnis vorzugeben. Damit eine Implementierung in der verfügbaren Zeit vorgenommen werden konnte, wurden zwei Kompromisse gemacht, welche den Entwicklungsaufwand verringern.

1. Der in Abschnitt 4.3.2 erklärte Filterfaktor, welcher eine gewisse Einflussnahme auf das Verhältnis von Ergebniswertanzahl zu Ausgabewertanzahl erlaubt, ist nicht implementiert.

2. Die gewünschte Anzahl von Ergebniswerten wird der Schaltung übergeben und muss nicht selbst ermittelt werden. Auch gibt es stets mindestens so viele Ergebniswerte wie Eingabewerte. Die Abtastrate kann also nur erhöht, nicht jedoch gesenkt werden.

Aufgrund dieser Vereinfachungen ist ein Vergleich der Lösungen nur begrenzt aussagekräftig. Die Implementierung des Taktteilers macht sich der Tatsache zunutze, dass die höhere der beiden Zahlen N und M (Anzahl der Eingabe- und Ausgabwerte) stets eine Zweierpotenz ist. Bei einer Abtastratensenkung wäre das nicht mehr gegeben, sodass dann eine zusätzliche Division zur Bildung des Kehrwerts der Eingabewerteanzahl und jeweils für jeden Ausgabewert eine Multiplikation erfolgen müsste.

Die korrekte Funktionsweise wurde mithilfe des Vivado Simulators und einer Software auf dem ARM Cortex-A9 Prozessor verifiziert. Die Software generiert Eingabewerte und schreibt diese in einen bestimmten Speicherbereich. Dieser Speicherbereich ist auf dem Zynq für einen Port reserviert, über den der Prozessor die Peripherie innerhalb der programmierbaren Logik ansprechen kann. Der Port verwendet das AXI (Advanced eXtensible Interface) Protokoll zum Versenden und Bestätigen von Datenpaketen. In diesem Fall dienen die Datenpakete dazu, Daten an eine Speicheradresse zu schreiben, daher kommt hier das AXI Memory-Mapped Protokoll zum Einsatz. Über ein sogenanntes *Interconnect* wird der Block-RAM, der die Eingabedaten speichert, mit dem Port verbunden. Die Software liest die Ergebniswerte nach Fertigstellung der Abtastratenkonvertierung über das gleiche Verfahren aus einem anderen Block-RAM und gibt diese auf der Konsole aus. Die Werte wurden dann auf ihre Plausibilität hin untersucht. Die einzelnen IP-Cores, die hierfür benötigt werden, sind: AXI GPIO [37] zur Konvertierung von AXI 32 Bit Schreibzugriffe in einzelne Signale pro Bit und umgekehrt, AXI BRAM-Controller [36] zur Konvertierung des Memory-Mapped Protokolls in die Block-RAM Schnittstelle und AXI-Interconnect [38] zur Anbindung des GPIO und BRAM Controllers an den AXI-Bus. Der AXI-Interconnect wurde mit dem AXI-GP0 Master-Port (*general purpose*) des Processing-System IP-Cores verbunden. Die Block-Automation des IP-Integrators und eine graphische Konfiguration der zu assoziierenden Speicherbereiche mit den BRAM-Controllern und AXI-GPIO Cores erleichtern die Systemintegration.

6. Vergleich der Implementierungen

In diesem Kapitel werden die generischen Implementierungen aus Kapitel 4 mit den optimierten Implementierungen aus Kapitel 5 verglichen. Zum Ende des Kapitels werden die generischen Software- und HLS-Implementierungen bezüglich ihrer Laufzeiten verglichen, da es in manchen Situationen wünschenswert ist, die Laufzeit eines HLS-Kerns zu maximieren, anstatt lediglich einen Wert unterhalb einer bestimmten Grenzen zu erreichen, z.B. der Abtastperiode der ADCs. In Tabelle 6.1 werden zunächst verkürzende Bezeichnungen für die verschiedenen Varianten eingeführt.

Tabelle 6.1.: Verglichene Implementierungen mit verkürzenden Bezeichnungen

UIP-GEN-XI	Generisch, x86, ICC	Poly-OPT-XI	Optimiert, x86, ICC
UIP-GEN-XG	Generisch, x86, GCC	Poly-OPT-XG	Optimiert, x86, GCC
UIP-GEN-AG	Generisch, ARM, GCC	Poly-OPT-AG	Optimiert, ARM, GCC
Poly-GEN-XI	Generisch, x86, ICC	UIP-HLS-T	HLS, Festkomma
Poly-GEN-XG	Generisch, x86, GCC	UIP-HLS-T-S	HLS, Festkomma, ohne Wurzel/Division
Poly-GEN-AG	Generisch, ARM, GCC	UIP-VHDL	Handgeschriebenes VHDL
UIP-OPT-XI	Optimiert, x86, ICC	UIP-VHDL-S	Handgeschriebenes VHDL, ohne Wurzel/Division
UIP-OPT-XG	Optimiert, x86, GCC	Poly-HLS-T-P	HLS, Festkomma, mit Pipeline
UIP-OPT-AG	Optimiert, ARM, GCC	Poly-VHDL	Handgeschriebenes VHDL

Einige Stellen dieser Arbeit beziehen sich auf Gruppen dieser Implementierungen, z.B. die optimierten UIP-Implementierungen für die CPU-Ausführung, unabhängig vom verwendeten Compiler. In solchen Fällen wird der entsprechende Suffix einfach weggelassen und z.B. UIP-OPT verwendet.

6.1. Performanz der Softwareimplementierungen

Damit die Performanz der CPU-Implementierungen beurteilt werden kann, werden die folgenden Metriken experimentell ermittelt

- Ausführungszeit (T_r, in Millisekunden)
- Bauzeit (T_t, in Sekunden)
- Dateigröße der ausführbaren Datei (M_i, in Kilobyte)
- Arbeitsspeichernutzung beim Ausführen (M_r, in Megabyte[1])
- Arbeitsspeichernutzung beim Bauen (M_t, in Megabyte)

Zunächst werden die Testmethodik erklärt und die zu prüfenden Implementierungen festgelegt. Dann folgen die Ergebnisse und eine kurze Diskussion. Falls für ein genaueres Ergebnis bei einer Metrik mehrere Werte ermittelt werden, wird der Mittelwert und die Standardabweichung angegeben. Die Anzahl der Werte werden im Tabellenkopf vermerkt. In Anhang B wird die Ausführungsumgebung für die Ermittlung von T_r genauer spezifiziert, inklusive der verwendeten Compiler-Optionen.

6.1.1. Testwerkzeuge und Umgebungen

Die Werte für T_r werden mit dem C++ Benchmark-Tool Nonius [29] ermittelt. Dazu wird jeweils ein sogenanntes *Benchmark* definiert, das den auszuführenden C++-Quelltext enthält. Ein Benchmark wird 1000 mal ausgeführt und es wird jeweils die Ausführungszeit gemessen. Danach werden die ermittelten Zeiten ausgegeben.

Die Werte T_t, M_i und M_t wurden ohne Nonius-Abhängigkeit ermittelt, um die Ergebnisse nicht durch dessen Quelltextkomplexität zu beeinflussen. Die Werte für T_t, M_r und M_t wurden durch das GNU `time` Werkzeug ermittelt. Dieses misst für ein vorgegebenes Kommando die gebrauchte Zeit und die sogenannte „Maximum resident set size“, die höchste im Arbeitsspeicher gehaltene Anzahl an Kilobytes. Diese Größenangabe beinhaltet für M_r den Speicher von $2 \cdot 720000 \cdot 4$ B $= 5,760$ MB Testdaten bei UIP und $2 \cdot 65535 \cdot 4$ B $= 524,288$ kB Testdaten bei der Polynominterpolation. Für M_i sind diese Zahlen irrelevant, da der Speicher mit `0x00` initialisiert ist und daher keinen Platz in der Datei beansprucht.

Die Software wurde auf einem Intel Core-i5 Quad-Core (3570K) mit $3,4$ GHz und 32GB Arbeitsspeicher gebaut. Dazu wurden zunächst mithilfe von Eclipse GNU `make`-Dateien konfiguriert und diese dann über ein Python-Skript ausgeführt. Die Bauzeiten beinhalten die komplette Zeit, die `make` braucht, um die Ausgabedatei zu produzieren. Auf dem Build-Host lief Linux und es wurde darauf geachtet, dass keine anderen I/O- oder CPU-intensiven Prozesse während den Messungen liefen.

[1] Aus der Dokumentation geht nicht hervor, ob GNU `time` für M_r und M_t Kibibytes oder Kilobytes ausgibt. Es wurden kB-Einheiten angenommen und daher die Ausgabe auch durch 10^3 geteilt.

Auf den Ausführungssystemen der Intel- und ARM-Prozessoren lief ebenfalls Linux. Auf diesen Systemen wurde jeweils nur der erste CPU-Kern für den Prozess-Scheduler genutzt. Ein Benchmark hat sich also prinzipiell den CPU mit allen anderen lauffähigen Programmen geteilt. Um zu garantieren, dass nur der Prozess des Benchmarkprogramms vom Prozessscheduler ausgeführt wird, wurde dem Benchmarkprozess eine sogenannte Echtzeitpriorität zugeordnet. Durch diese Methodik wurde erreicht, dass kein anderer CPU-Kern mit der Benchmarkausführung interferiert, z.B. durch die konkurrierende Nutzung geteilter CPU-Caches, da keine Prozesse auf anderen CPU-Kernen liefen. Zusätzlich wurden alle Interrupts, die dies unterstützen, mit CPU-Kern 1 assoziiert, um die Anzahl der Zeitausreißer zu reduzieren.

6.1.2. Getestete Implementierungen

Die getesteten Programme sind jeweils die in Abschnitt 4.2 und Abschnitt 4.3 entworfenen plattformunabhängigen Implementierungen, sowie die unter Kapitel 5 optimierten Varianten. Für UIP-GEN (und UIP-OPT ohnehin) wird die erste Erweiterung aus Abschnitt 5.1 verwendet, um die Abtastwerte blockweise verarbeiten zu können. Für UIP-OPT werden die explizit vektorisierten, bei Poly-OPT die automatisch vektorisierten Implementierungen verwendet.

6.1.3. Testergebnisse

Für alle Ergebnisse, mit Ausnahme der Dateigröße, wird die Standardabweichung angegeben. Zusätzlich sind im Anhang A die Verteilung der 1000 Werte von T_r dargestellt. In Tabelle 6.2 sind die Werte für den UIP-Algorithmus gelistet. Es wurden 720000 Abtastwerte, jeweils für I und U, verwendet. Zunächst wurde versucht, die maximal mögliche Anzahl von $72 \cdot 10^6$ Abtastwerten zu benutzen. Da die Speicherung innerhalb des Benchmark-Programms aber zu einem Speicherüberlauf führte, wurde die Zahl um zwei Größenordnungen verringert. In Tabelle 6.3 sind die Werte für die Polynominterpolation aufgelistet. Es wurden 65535 Abtastwerte auf 65536 Ergebniswerte konvertiert.

Global fällt auf, dass der Intel-Compiler deutlich größere Dateien als GCC für die Intel Zielplattform erzeugt. Der Assemblercode zeigt, dass hiervon ein großer Teil auf automatisch benutzte und optimierte Bibliotheken von Intel entfällt, z.B für die Implementierung der Funktion `sin`. Diese Bibliotheken werden statisch in das Binary gelinkt und verursachen große Dateien. In der Ausführungszeit ist zwischen dem Intel-Compiler und GCC konsistent eine um etwa 20 % kürzere Laufzeit beim Intel-Compiler zu beobachten. Bei der UIP-Berechnung kann eine starke Verbesserung der Laufzeitgeschwindigkeit für die optimierten Varianten beobachtet werden. In der

Tabelle 6.2.: Performanzkennwerte der Softwareimplementierungen von UIP. Es wurden 720000 Abtastwerte akkumuliert.

(N)	T_r ms ×1000	T_t s ×10	M_r MB ×10	M_t MB ×10	M_i kB ×1
UIP-GEN-XI	13.374 ± 0.006	35.719 ± 0.276	8.715 ± 0.074	2940.903 ± 0.111	452.608
UIP-GEN-XG	16.253 ± 0.004	16.386 ± 0.172	8.600 ± 0.042	914.155 ± 0.767	232.284
UIP-GEN-AG	39.391 ± 0.009	16.028 ± 0.151	7.907 ± 0.040	875.827 ± 1.112	180.548
UIP-OPT-XI	1.908 ± 0.008	0.431 ± 0.005	7.670 ± 0.084	104.038 ± 0.090	40.212
UIP-OPT-XG	1.944 ± 0.008	0.700 ± 0.008	7.549 ± 0.068	118.942 ± 0.434	11.760
UIP-OPT-AG	12.965 ± 0.013	0.512 ± 0.004	7.764 ± 0.019	87.232 ± 0.743	13.904

Tabelle 6.3.: Performanzkennwerte von Softwareimplementierungen der Polynominterpolation. Es wurden 65535 Abtastwerte auf 65536 Werte interpoliert.

(N)	T_r ms ×1000	T_t s ×10	M_r MB ×10	M_t MB ×10	M_i kB ×1
Poly-GEN-XI	3.476 ± 0.003	36.081 ± 0.265	4.074 ± 0.095	2923.960 ± 0.134	537.660
Poly-GEN-XG	4.214 ± 0.002	16.874 ± 0.146	3.962 ± 0.060	907.422 ± 0.979	314.392
Poly-GEN-AG	31.299 ± 0.007	16.023 ± 0.123	3.233 ± 0.040	863.755 ± 0.999	260.896
Poly-OPT-XI	2.446 ± 0.002	0.560 ± 0.006	2.787 ± 0.105	54.964 ± 0.065	120.992
Poly-OPT-XG	3.044 ± 0.002	0.307 ± 0.004	2.646 ± 0.057	34.836 ± 0.104	89.588
Poly-OPT-AG	13.928 ± 0.009	0.297 ± 0.006	2.425 ± 0.073	29.694 ± 0.055	94.320

generischen Version der UIP-Implementierung können beide Compiler keine SIMD-Instruktionen verwenden, weshalb die Laufzeitperformanz für beide Compiler hier sehr gering ist. Für die generischen Versionen der Polynominterpolation können hingegen bereits SIMD-Instruktionen emittiert werden.

Die stark gestiegenen Übersetzungszeiten bei den generischen Implementierungen werden durch die HLS-Headerdateien verursacht. Diese Abhängigkeiten werden durch die Metafunktionen verursacht (siehe Tabelle 4.2 und Tabelle 4.5), die ihre Ergebnistypen für die möglichen Eingabetypen `float`, `ap_fixed` und `ap_uint` definieren (siehe Abschnitt 2.5). Technisch gesehen wird an diesen Stellen nur eine Vorwärtsdeklaration (engl. *forward declaration*) dieser HLS-Templates benötigt. Der Versuch, diese Klassentemplates vorwärts zu deklarieren, schlug allerdings fehl, da die Templateparameter `enum`-Typen verwenden, die sich nicht vorwärts deklarieren lassen. Ein alternatives Design der Metafunktionen kann dieses Problem beheben.

Dazu müssten die Metafunktionen in den Traitsklassen definiert werden, die dann ihrerseits (bei einem `float`-Trait) keine Abhängigkeiten zu HLS-Headern benötigen. Der Typ `sqsum_type` aus Listing 4.1 würde dann wie im folgenden Listing abgeleitet werden.

```
1 template<typename Traits>
2 class UipLineResult {
3    typedef Traits::in_type in_type;
4    typedef Traits::AccumulateSquares<
5                in_type,
6                Traits::el_n>::type sqsum_type;
7    ...
```

Quelltext 6.1: Vorgeschlagene Änderung an Listing 4.1 zum Verbessern der Baugeschwindigkeit

Allgemein kann die Situation auch durch Xilinx verbessert werden, indem eine Vorwärts-Deklaration der besagten Templates ermöglicht wird. Ein **`enum`** kann bspw. ab C++11 durch die Angabe eines *underlying types* vorwärts deklariert werden. Alternativ kann ein separater Header verwendet werden, in dem nur **`enum`**-Typen deklariert werden.

6.2. Performanz der FPGA-Implementierungen

Die Kennwerte der FPGA-Implementierungen werden im Folgenden für die HLS-Implementierungen anhand von Angaben der VHLS IDE ermittelt. Für die optimierte UIP-Berechnung wird die Latenz anhand einer manuellen Analyse des geschriebenen VHDL-Quelltextes ermittelt, da die einfache Struktur eine statische und händische Analyse zulässt. Der Datenfluss der Polynominterpolation ist etwas komplexer, sodass die Latenz hier anhand der Simulation ermittelt wurde. Es werden jeweils nur die Festkommavarianten der HLS-Implementierungen mit den manuell optimierten Versionen verglichen, da für letztere keine Fließkommavarianten implementiert sind. Die Zahlen für den Ressourcenbedarf, die Latenzen und maximalen Taktraten der optimierten Versionen stammen aus den Post-Routing Angaben von Vivado, wobei jeweils die UIP- und Polynominterpolationsmodule als Hauptmodule für ein *out-of-context*-Routing ausgewählt wurden, damit die anderen notwendigen Module (siehe Abschnitt 5.5) die Messungen nicht beeinflussen. Um die minimal mögliche Taktperiode zu erhalten, wurde der *slack*-Wert aus der *max delay paths*-Tabelle im *timing report* von der 10 ns Taktlänge subtrahiert.

Tabelle 6.4 vergleicht die Kennwerte der UIP-Implementierungen, inklusive der Varianten, bei denen die Durchschnittsbildung entfällt (siehe Abschnitt 4.2.2). Zu erkennen ist eine kleinere Latenz bei der optimierten Variante mit Durchschnittsbildung. Vermutlich liegt dies an der Division, für die VHLS in der Performanzanalyse eine Latenz von 66 Takten für den quadratischen und 49 Takten für den arithmeti-

Tabelle 6.4.: Performanzkennwerte der Hardwareimplementierungen von UIP, mit und ohne Durchschnittsbildung. Die maximale Latenz beschreibt die Anzahl der Takte, die ein Abtastwert benötigt, um aufaddiert und durch Division sowie Wurzelbildung gemittelt zu werden. Falls der Abtastwert nur aufaddiert wird, entsteht eine minimale Latenz.

	DSPs	FFs	LUTs	Latenz		II		f_{max} (MHz)
(Total)	80	35 200	17 600	min	max	min	max	Ziel 100
UIP-HLS-T	3	6660	6830	2	108	3	109	113
UIP-HLS-T-S	3	378	668	2	2	3	3	159
UIP-VHDL	2	7765	5612	68	68	68	68	115
UIP-VHDL-S	2	967	513	9	9	1	1	228

schen Mittelwert angibt. Dabei überlappen sich die Divisionen für beide Werte auch nur jeweils um acht Takte, sodass die Durchschnittsbildung wegen den Divisionen insgesamt 107 Takte benötigt. Anstrengungen, die Latenz bzw. die Pipelinestufen des HLS Dividieres mittels der `RESOURCE`-Direktive herabzusetzen, schlugen mit einer Fehlermeldung fehl, derzufolge die Latenz nicht verringert werden kann. Bei den optimierten Varianten wurden keine Anstrengungen unternommen, die minimale Latenz zu reduzieren, da die Ausgabewerte ohnehin nur sinnvoll verwendet werden können, falls sie auch die Durchschnittsbildung durchlaufen hatten. Mit einer maximalen Latenz von 68 Takten bei einer Taktlänge von 10 ns kann eine Aktivierung zudem auch innerhalb von 825 ns abgearbeitet werden, sodass die Schaltung stets wieder bereit ist für den nächsten Abtastwert.

Die Kennwerte der Polynominterpolation sind in Tabelle 6.5 angegeben. Bei der Polynominterpolation beschreibt die minimale und maximale Latenz die Anzahl der Takte, die eine vollständige Interpolation auf einen bzw. 2^{17} Ergebniswerte benötigt. Dieser Wert wurde für die VHDL-Version mithilfe der Simulation ermittelt. Für die Latenz der VHDL-Version wurde diese Zahl mit eins bzw. 2^{17} skaliert und $3 \cdot 20$ Takte für das vorbereitende Einlesen der ersten 20 Abtastwerte aufaddiert. Als Latenzvorgabe für Block-RAMs wurde die HLS-Version auf der Standardeinstellung belassen, während für die VHDL-Version eine Latenz von einem Takt konfiguriert wurde. Der Wert für die Anzahl der Block-RAMs bei der optimierten Version stammt aus der graphischen Konfiguration des Block-Memory-Generators [39] von Xilinx.

Falls, wie in Abschnitt 5.5.2 erklärt, für jeden Ergebniswert eine Multiplikation erfolgen müsste, kann das den Durchsatz von Poly-VHDL durchaus um einige Takte verschlechtern und an den von Poly-HLS-T-P angleichen.

Tabelle 6.5.: Performanzkennwerte von Hardwareimplementierungen der Polynominterpolation. Die Spalte „Clk/S“ beschreibt die Anzahl der Takte, die zwischen zwei Ergebniswerten liegen.

(Total)	BRAMs 120 (×18 Kb)	DSPs 80	FFs 35 200	LUTs 17 600	Latenz min	Latenz max	Clk/S	f_{max} (MHz) Ziel 100
Poly-HLS-T-P	16 (×18 Kb)	19	986	915	47	1 179 739	9	135
Poly-VHDL	5 (×36 Kb)	20	3047	1018	64	524 348	4	134

6.3. Vergleich der Laufzeiten von Software- und HLS-Implementierungen

In bestimmten Fällen, z.B. falls ein existierendes Softwareprogramm auf die Abarbeitung einer Aufgabe durch das FPGA wartet, ist es wünschenswert, dass das FPGA seine Aufgaben so schnell wie möglich abarbeitet und eine möglichst geringe Latenz aufweist. Dies steht im Kontrast zu der bisherigen Überlegung, dass die HLS-generierte Architektur nur so schnell wie *nötig* sein sollte, um möglichst wenig Ressourcenbedarf zu erzeugen. Um herauszufinden, ob sich das Auslagern von Aufgaben des Softwareprogramms in das FPGA lohnen könnte, werden in Tabelle 6.6 die Verarbeitungszeiten für einen einzelnen Abtastwert ermittelt.

Tabelle 6.6.: Laufzeitwerte der generischen Software- und HLS-Implementierungen, errechnet für einen Abtastwert.

	Nanosekunden pro Abtastwert
UIP-HLS-T	30
UIP-GEN-XI	18.575
UIP-GEN-XG	22.574
UIP-GEN-AG	54.709
Poly-HLS-T-P	90
Poly-GEN-XI	53.04
Poly-GEN-XG	64.3
Poly-GEN-AG	477.585

Für die Softwareimplementierungen werden als Laufzeiten T_r/N aus Tabelle 6.3 und 6.2 angenommen, wobei N jeweils die Abtastwertanzahl bei UIP-GEN und Ergebniswertanzahl bei Poly-GEN bezeichnet. Für UIP-HLS-T wird der Wert mithilfe des minimalen Initiierungsintervalls, bei Poly-HLS-T-P durch die Clk/S-Angabe

(Takte, die zwischen zwei Ergebniswerten liegen) ermittelt (siehe Tabelle 6.4 und 6.5). Diese Angaben spiegeln nicht die vollständigen Latenzen der Implementierungen wider, da Laufzeiten, die nicht von der Anzahl an Abtastwerten abhängen, nicht berücksichtigt werden können. Diese sind 108×10 ns bei UIP-HLS-T und 47×10 ns bei Poly-HLS-T-P. Es genügen aber bereits 360 Abtastwerte, sodass diese Laufzeiten weniger als 10 % der vollständigen Latenz von UIP-HLS-T verursachen. Für Poly-HLS-T-P genügen bereits 53 Ergebniswerte.

Als zugrundeliegende Taktperiode kann hier sowohl die HLS Design-Taktperiode mit 10 ns, als auch das ermittelte f_{max} der vorangegangenen Abschnitten verwendet werden. Hier wurden die 10 ns genutzt, da das Setzen und Routing der Netzliste auf einem stärker besetzten FPGA mit weniger freier Fläche vermutlich zu einem kleineren f_{max} führen wird.

Die Tabelle zeigt, dass zumindest in den hier untersuchten Taktkonfigurationen und nicht explizit optimierten Implementierungen der Intel Atom E3845 Prozessor kürzere Laufzeiten einhält als eine HLS-Implementierung für die programmierbare Logik des Zynq-7010. Der ARM Cortex-A9 kann bei dieser Geschwindigkeit nicht mithalten, zumindest wenn die generischen Varianten verwendet werden.

7. Auswirkung generischer Quelltextmodifikationen auf die Performanz für verschiedene Zielarchitekturen

In diesem Kapitel werden Quelltextmodifikationen an den generischen Implementierungen aus Kapitel 4 vorgenommen und deren Auswirkungen verglichen. Es werden für jede Modifikation der Ressourcenbedarf und die Laufzeitperformanz der CPU- und HDL-Implementierungen evaluiert. Bevor die Modifikationen beschrieben und analysiert werden, erklärt der nächste Abschnitt zunächst einige Grundlagen zu den möglichen Optimierungen durch VHLS, die mit den Quelltextmodifikationen wechselwirken können. Das allgemeinen Vorgehen der HLS wurde bereits in Abschnitt 2.4 beschrieben.

7.1. Optimierende Transformationen bei VHLS

Einige der Möglichkeiten, durch die sich das Ergebnis der HLS infolge von Quelltextänderungen verbessert oder verschlechtert, sind Zwischencodetransformationen. Wie unter Abschnitt 2.4 beschrieben, wird der C++-Quelltext zunächst von einem Frontend in eine Zwischenform überführt. Auf dem Zwischencode werden Optimierungen (Transformationen) durchgeführt, mit dem Ziel, dass die spätere Hardware leistungsfähiger oder ressourcensparender ist. In diesem Kapitel relevante Optimierungen sind:

Abrollen von Schleifen Hierbei wird die Schleife durch eine Aneinanderreihung von Kopien des Schleifenrumpfes ersetzt. VHLS führt kein automatisches Abrollen aus, sodass der Entwickler die Schleife entweder manuell durch entsprechenden Quelltext ersetzen oder mit der `unroll`-Direktive ein Abrollen erreichen kann. Dabei kann die Schleife auch teilweise um den Faktor k abgerollt werden.

Die Anzahl der Schleifendurchläufe sinkt dann um den Faktor k, während der Rumpf durch k Kopien des ursprünglichen Rumpfes ersetzt wird. Jede Kopie des Rumpfes kann dann individuell transformiert werden und beispielsweise eigene HLS Operator-Instanzen verwenden.

Pipelining von Schleifen Beim Pipelining von Schleifen kann der nächste Durchlauf bereits beginnen, bevor der vorherige Durchlauf vollständig abgearbeitet wurde.

Expansion von Funktionsaufrufen Hierbei ersetzt VHLS bei einem Funktionsaufruf den Aufruf mit dem Inhalt der aufgerufenen Funktion (engl. *inlining*). Laut dem HLS-Handbuch von Xilinx [43] expandiert VHLS die Aufrufe von kleineren Funktionen automatisch. Der Entwickler kann mit der `inline`-Direktive verhindern oder erzwingen, dass eine Expansion stattfindet.

Nach der Optimierung findet durch die Allokation (siehe Abschnitt 2.4) insbesondere eine Zuordnung von Funktionsaufrufen zu Funktionsmodulinstanzen statt, falls die Aufrufe im Zwischencode nicht expandiert wurden. Werden für zwei Funktionsaufrufe jeweils separate Instanzen erzeugt, können beide Aufrufe parallel ablaufen, da diese dann jeweils eigene Signalpfade besitzen. Dadurch steigt aber auch der Ressourcenbedarf.

7.2. Modifikationen

Sämtliche untersuchten Effekte basieren auf den Standardeinstellungen von VHLS. Jede Modifikation erhält in ihrer Abschnittsüberschrift eine kürzende Bezeichnung. Da in den Quelltexten beider Implementierungen aus Abschnitt Kapitel 4 zu wenig Möglichkeiten für geringfügige Modifikationen gefunden wurden, wird auf eine weitere Implementierung zurückgegriffen, die nicht in Kapitel 4 vorgestellt und vom Messgerätehersteller zur Verfügung gestellt wurde. Dabei handelt es sich um eine Reihe von digitalen Tiefpassfiltern, mit der Möglichkeit, dynamisch andere Grenzfrequenzen einstellen zu können. Die Filter sind aus einem Teil einer Implementierung des *Flickermeters* aus der Norm IEC 61000-4-15[10]. Dabei handelt es sich um ein Instrument zum Messen der Störintensität von Spannungsschwankungen im Stromnetz, die periodische Helligkeitsschwankungen in Glühlampen verursachen können. Das Flickermeter besteht aus fünf Funktionsblöcken, die das Lampe/Auge/Gehirn System derart nachbilden, dass im Ergebnis ein Wert entsteht, der etwas über die Störintensität aussagt. Der hier verwendete Teil des Instrumentes entspricht den Blöcken eins bis zur Hälfte von Block drei. Zunächst wird in Block eins der Spannungshalbwelleneffektivwert auf einen Bereich von $0, \ldots, 1$ normiert. Block zwei quadriert den Wert und modelliert das Leuchtverhalten der Glühlampe. Der hier

verwendete Teil von Block drei modelliert das Verhalten des menschlichen Auges durch einen digitalen Tiefpassfilter mit einer Grenzfrequenz von 35 Hz. Dieser Filter wird durch drei in Reihe geschaltete Tiefpassfilter implementiert. Im folgenden Pseudoquelltext ist die Funktionsweise skizziert.

```
1 float meter(float f) {
2    float f1 = normalize(f);
3    float f2 = f1 * f1;
4    float f3 = butterworth(f2);
5    return f3;
6 }
```

Quelltext 7.1: Pseudoquelltext der Flickermeter-Implementierung. Nachdem der Eingabewert auf einen festen Bereich normalisiert wurde, wird er für die Modellierung der Glühlampe und des menschlichen Auges quadriert und gefiltert.

Das Flickermeter kann sowohl für 230 V als auch 120 V Netzspannung konfiguriert werden. Außerdem kann die Netzfrequenz entweder auf 50 Hz oder 60 Hz eingestellt werden. Bei 120 V und 60 Hz wird der Tiefpassfilter auf eine Grenzfrequenz von 42 Hz gestellt. Hierfür existieren `configure` Funktionen, die die Filter entsprechend konfigurieren. Diese werden als Filter mit unendlicher Impulsantwort mit analogen Filterkoeffizienten konfiguriert. Danach folgt der Aufruf einer `digitise` Funktion, die diese Koeffizienten in eine digitale Form für konvertiert.

Da die Filter einen hohen dynamischen Wertebereich repräsentieren müssen, wird das *single-precision* IEEE754 Fließkommaformat verwendet. Das Entwurfsmuster aus Kapitel 4, Festkommadatentypen für Zwischenwerte aus den Eingabetypen abzuleiten, konnte hier nicht angewendet werden, da die Ausgaben der Filter in deren Eingaben rückgekoppelt sind. Diesen Implementierungen werden die Bezeichner Flicker-HLS für die HLS-Variante und Flicker-GEN für die Softwarevariante zugeordnet.

7.2.1. Herausziehen von Berechnungen aus einer Schleife (LICM)

Normalerweise wird in der Softwareentwicklung der Gültigkeitsbereich von Variablen so klein wie möglich gehalten. Entsprechend sollten z.B. globale Variablen vermieden werden und C++ Programmierer können Variablen dort definieren und initialisieren, wo sie verwendet werden, anstatt wie in älteren C Versionen am Anweisungsblockanfang. Dazu kommt die Tatsache, dass minimale Quelltextmodifikationen in der Softwareentwicklung meistens keine spürbaren Auswirkungen auf die Laufzeitperformanz oder den Ressourcenverbrauch haben, auch weil die vom Compiler generierten Instruktionen zur Laufzeit nur vorübergehend während ihrer Ausführung Ressourcen beanspruchen. Zusammengenommen kann das dazu führen, dass Berechnungen

vom Entwickler am Ort der Verwendung durchgeführt werden, selbst wenn dadurch die selbe Berechnung innerhalb einer Schleife mehrfach durchlaufen wird und das gleiche Ergebnis liefert.

In Listing 4.9 der Polynominterpolation wird die Variable `recip` vor der Schleife berechnet und dann mehrfach verwendet. Im Folgenden ist das noch einmal dargestellt

```
1 ...
2 recip_type recip = recip_type(1.0) / outcount;
3 ...
4 for(el_n_type i = 0; i < outcount; ++i) {
5    ...
6    auto rpos = recip * i;
7    ...
8 }
```

Quelltext 7.2: (Original) Vorausberechnung eines Wertes

Wird die Berechnung in die Schleife gezogen, muss sie zur Laufzeit in jedem Durchlauf erneut ausgewertet werden. Dabei spielt es keine Rolle, ob der Variablen ein Name zugewiesen wird oder ob die Berechnung direkt anstelle der Verwendung in `recip*i` eingesetzt wird. Im Beispiel wird eine benannte Variable verwendet.

```
1 ...
2 for(el_n_type i = 0; i < outcount; ++i) {
3    ...
4    recip_type recip = recip_type(1.0) / outcount;
5    auto rpos = recip * i;
6    ...
7 }
```

Quelltext 7.3: (Modifiziert) Berechnung des Wertes am Verwendungsort

VHLS zieht die Berechnung, die unabhängig von `i` ist, nicht aus der Schleife heraus (engl. *loop-invariant code motion*). Das führt, abhängig davon, ob die Schleife eine Pipeline verwendet oder abgerollt wird, zu einem erhöhten Ressourcenbedarf. In Abschnitt 4.3.3 wurde das Pipelining für die Schleife aktiviert, daher wächst der Bedarf besonders an LUTs auf mehr als das Doppelte. Ohne Pipeline wächst stattdessen die maximale Latenz der Schleife von $\approx 8 \cdot 10^6$ auf $\approx 14 \cdot 10^6$ Taktzyklen. Für einen Softwarecompiler kann es durchaus vorteilhaft sein, die Berechnung in der Schleife zu belassen. Besonders, falls die Zielarchitektur wenige Register zur Verfügung hat, der Schleifenrumpf viele Register benötigt und die Berechnung wenig komplex oder, verglichen mit den anderen Operationen im Schleifenrumpf, vernachlässigbar trivial ist. Dann können Register für die Speicherung von `recip` eingespart werden.

7.2.2. Vom Xilinx High-Level Synthese Handbuch empfohlene Quelltextstruktur (MANUAL)

Im HLS-Handbuch von Xilinx sind eine Vielzahl von Hinweisen aufgelistet, die man befolgen sollte, damit VHLS guten HDL-Quelltext synthetisieren kann. Dazu gehören allgemeine Hinweise zu Quelltextkonstrukten und Wechselwirkungen zwischen verschiedenen Quelltextstilen und HLS-Direktiven. Im Folgenden wird die Implementierung der Polynominterpolation und UIP-Berechnung gemäß diesen Richtlinien geändert, wobei darauf geachtet wird, dass der Quelltext kompatibel mit den Softwarecompilern bleibt. Zu beachten ist, dass es sich hierbei um die Interpretation des Autors handelt, und es sicherlich andere Möglichkeiten gibt, die Synthese weiter zu verbessern.

Zur Erinnerung sei die Schleifenstruktur der Implementierung aus Listing 4.9 gegeben. Der dortige Funktionsaufruf von `interpolateAroundSample` wird im folgenden Listing expandiert, damit die Schleifenstruktur erkennbar wird.

```
1   // Iteriere von 0 bis Anzahl Ausgabewerte
2   for(i = 0 to N-1) {
3      // Berechne Indices in[i-9..i+10] der Stuetzstellen
4      ...
5      // Berechne Index des Koeffizienzensatzes fuer das Polynom
6      ...
7      // Evaluiere das Polynom mit Stuetzstellen und Koeffizienten
8      for(j = 0 to 19) {
9         ...
10     }
11  }
```

Quelltext 7.4: (Original) Schleifenstruktur der Polynominterpolation

Folgende Verbesserungen für die HLS werden vorgenommen:

Perfekte Schleifenverschachtelung VHLS kann bei einer sogenannten *perfekten Verschachtelung* (engl. *perfect nesting*) die innere mit der äußeren Schleife vereinen (engl. *flattening*). Das Ergebnis ist eine einzige Schleife über $N*20$ Schritte, wodurch der HDL-Verwaltungsaufwand für die innere Schleife entfällt. Bei einer perfekten Verschachtelung dürfen keine Anweisungen zwischen den beiden Schleifenköpfen vorkommen. Der bereits vorhandene Quelltext an dieser Stelle kann durch bedingte Ausführung bei `j==0` der inneren Schleife ausgeführt werden.

```
1      // Iteriere von 0 bis Anzahl Ausgabewerte
2      for(i = 0 to N-1) {
3         // Vorher: Code hier ...
4         for(j = 0 to 19) {
5            if(j == 0) {
6               // Nachher: Code hier ...
```

```
7                }
8                ...
9            }
10       }
```

Quelltext 7.5: (Modifiziert) Perfekte Verschachtelung

Pipelining der inneren Schleife Die innere Schleife kann durch ein Pipelining mehrere Iterationen überlagern. In Kombination mit der Schleifenvereinigung wird dann die komplette Schleifenstruktur auf einer Pipeline ausgeführt, da sich die gesamte Berechnung innerhalb der inneren Schleife befindet. Ein Pipelining der äußeren Schleife wäre auch möglich. Das hätte aber den Nachteil, dass laut dem HLS-Handbuch dann alle inneren Schleifen komplett abgerollt werden müssten.

Teilweises Abrollen der inneren Schleife Nach dem Pipelining wird in einer Iteration der inneren Schleife nun jeweils ein Eingabewert und Koeffizient gelesen. Der RAM-Block im FPGA bietet aber die Möglichkeit, von zwei Ports zeitgleich zu lesen (engl. *dual port*). Das kann ausgenutzt werden, indem die innere Schleife partiell abgerollt wird. VHLS bietet dafür die `unroll factor=2` Direktive. Die Synthese erkennt dann automatisch, dass ein Dual-Port RAM verwendet werden kann. Dadurch halbiert sich die Latenz der gesamten Implementierung.

Veränderungen am Quelltext bestehen hier also durch eine etwas andere Verschachtelung der beiden Schleifen. Die duale Block-RAM-Schnittstelle stellt eine untere Schranke für die Gesamtlatenz der Polynominterpolation dar. Eine weitere Verbesserung der Latenz kann erwirkt werden, indem VHLS mehrere Block-RAM-Schnittstellen für die Eingabewerte und Koeffizienten inferiert, sodass sich die Speicherzugriffslatenzen auf diese Block-RAMs aufteilen. Dabei kann bspw. jeder Zugriff auf gerade Indizes aus einem ersten Block-RAM und ungerade Indices aus einem zweiten Block-RAM lesen. Da sich dadurch aber die Schnittstelle des generierten IP-Cores stark ändert, werden die entsprechenden Direktiven hier nicht angewendet. Eine Alternative zum Aufteilen der Block-RAMs ist, die Lesebreite von Wörtern zu erweitern, sodass mit einem Lesezugriff bereits mehr als ein Abtastwert gelesen werden kann. Trotz zahlreicher Versuche mit den entsprechenden Direktiven konnte VHLS aber mit dieser zweiten Alternative keine Verbesserung im Durchsatzverhalten erzielen.

In Abschnitt 4.2.2 wurde erklärt, dass durch ein Pipelining der UIP-Implementierung das Initiierungsintervall reduziert werden kann. Diese Änderung benötigt keine Quelltextanpassungen, sondern lediglich explizite Direktiven, damit VHLS eine verbesserte Schaltung synthetisieren kann. Hierfür muss in die Hauptfunktion

aus Listing 4.5 die Direktive `pipeline` eingesetzt werden. Außerdem müssen die Direktiven aus Listing 4.6 nun auch auf die Festkommaimplementierung angewendet werden, da das Pipelining sonst zu einem zu großen Ressourcenbedarf führt.

7.2.3. Abrollen von Schleifen (UNROLL)

Laut dem HLS-Handbuch lässt VHLS Schleifen im Quelltext standardmäßig in ihrer iterativen Form. Ist es vorteilhafter, die Schleife abzurollen, muss manuell eingegriffen werden. Der Entwickler kann die Schleife entweder explizit abrollen. Oder er weicht auf Direktiven wie im vorherigen Beispiel aus. In Kombination mit Funktionsaufrufen kann es zu interessanten Effekten kommen. In der folgenden Schleife werden Koeffizienten für ein analoges Filter in digitale Filterkoeffizienten konvertiert. Die dafür aufgerufene Funktion enthält eine Reihe von Multiplikationen, Addition und Subtraktionen. Vermutlich weil der Schleifenkörper als einziger Aufrufer dieser Funktion auftritt, hat sich VHLS entschieden, diesen Aufruf zu expandieren.

```
1 for(int k = 0; k < 3; k++) {
2    butterWort[k].digitise(samplePerSec);
3 }
```

Quelltext 7.6: (Original) Nicht abgerollte Schleife mit Funktionsaufruf im Rumpf.

Ein explizites Abrollen führt dazu, dass sich VHLS gegen eine automatische Expansion des Aufrufs entscheidet. Es wird also, wie unter Abschnitt 2.4 beschrieben, ein HDL-Modul für die `digitise` Funktion generiert. Allerdings produziert VHLS eine Architektur, die das HDL-Modul der `digitise`-Funktion dreifach instantiiert. Das gleiche Verhalten konnte für `average` bei der UIP-Implementierung beobachtet werden (siehe Abschnitt 4.2.2).

```
1    butterWort[0].digitise(samplePerSec);
2    butterWort[1].digitise(samplePerSec);
3    butterWort[2].digitise(samplePerSec);
```

Quelltext 7.7: (Modifiziert) Abgerollte Schleife mit dreimal kopiertem Rumpf

Die drei Aufrufe können wegen der dreifachen Ressourceninstantiierung parallel ausgeführt werden. Tatsächlich zeigt die Analyseansicht, dass die beiden letzten Aufrufe parallel erfolgen und sich teilweise mit dem ersten Aufruf überschneiden. Die vollständige Parallelisierung wird dadurch verhindern, dass `butterWort` ein Array-Member ist. Dadurch synthetisiert VHLS eine Block-RAM Schnittstelle für den Zugriff der `digitise` Memberfunktion auf das Array. Diese Block-RAM Schnittstelle kann für maximal zwei zeitgleiche Zugriffe verwendet werden.

7.2.4. Wiedereintrittsfähige (reentrante) Schnittstelle (REENTRANT)

Es gibt grundsätzlich zwei Arten von Implementierungen, mit denen Zustände zwischen zwei aufeinanderfolgenden Aufrufen von Funktionen gehalten werden können. Der Zustand kann entweder durch die aufgerufene Funktion gesichert werden. Alternativ kann der Aufrufer eine Speicheradresse übergeben, an der die Funktion den Zustand sichern kann. Ein wichtiges Designkriterium, das oft bereits eine der beiden Implementierungsarten ausschließt, ist, ob die Funktion auf mehrere Zustände zeitgleich oder im Wechsel zugreifen muss. Diese funktionale Eigenschaft wird in der Softwareentwicklung als Wiedereintrittsfähigkeit (engl. *reentrancy*) noch etwas strenger dergestalt formuliert, dass die Funktion an beliebiger Stelle während ihrer Ausführung unterbrochen werden können muss. Die Funktion wird dann ein zweites mal ausgeführt. Die ursprüngliche erste Ausführung muss danach fortgeführt werden können, ohne dass die Funktion ihre Spezifikation verletzt.

Für die Hauptfunktion der HLS gibt es diesbezüglich die beiden folgenden Varianten:

```
1    float flicker_float(bool restart, ActiveConfig<Traits> config,
2       float sample)
3    {
4       // save to 'state'
5       static Impl state;
6       ...
7    }
```

Quelltext 7.8: (Original) Nicht wiedereintrittsfähige Schnittstelle

```
1    float flicker_float_r(bool restart, ActiveConfig<Traits> config,
2       float sample, Impl *state)
3    {
4       // save to *state
5       ...
6    }
```

Quelltext 7.9: (Modifiziert) Wiedereintrittsfähige Schnittstelle

Die funktionalen Eigenschaften der wiedereintrittsfähigen Variante sind für die Software- und HLS-Übersetzung insofern identisch, als dass der Aufrufer bei jeder Aktivierung bestimmt, an welchem Ort der Zustand gespeichert wird. Zu dem Verhalten der nicht wiedereintrittsfähigen Variante bei VHLS beschränkt sich das HLS-Handbuch nach Kenntnisstand des Autors darauf, dass eine lokal-statische Variable mit FFs oder Block-RAMs umgesetzt wird. Nicht klar beschrieben wird das Verhalten für den Fall, dass der HDL-Block der umgebenden Funktion von VHLS mehrfach instantiiert wird (das war bspw. im obigen Beispiel der abgerollten Schleife gegeben) und es sich bei der betreffenden Funktion um eine Unterroutine der Hauptfunktion

handelt. Damit das Verhalten denen von Softwarecompilern gleicht, muss der HDL-Block für die Funktion das Register über Signale mit Ports veröffentlichen, damit mehrere Instanzen des HDL-Blocks ihre Registerwerte miteinander synchronisieren können. Die Analyseansicht von VHLS zeigt, dass ein solcher Port für Unterfunktionen der Hauptfunktion tatsächlich generiert wird. Anders ist das Verhalten bei der Hauptfunktion. Hier wird von VHLS kein Port erzeugt, sodass das erzeugte HDL-Modul für die Hauptfunktion den Zustand autonom verwaltet. Wird das HDL-Modul in einem übergeordneten Modul mehrfach instantiiert (z.B. als IP-Cores), erhält jede Instanz ihren eigenen Zustand.

Eine mögliche Nebenwirkung dieser Änderung tritt auf, wenn die Zustandsstruktur Arrays enthält. Die wiedereintrittsfähige Variante synthetisiert für Zugriffe auf Arraymember von Zeigerparametern der Hauptfunktion stets Block-RAM Schnittstellen. Wird der Zustand als lokal-statische Variable gesichert, zeigt die Analyseansicht, dass VHLS das folgende Array aus drei Elementen, das bereits in der `UNROLL`-Modifikation verwendet wurde, mit FFs realisiert.

```
1 struct Impl {
2    ...
3    FilterDirectFormII<Traits> butterWort[3];
4    ...
5 }
```

Quelltext 7.10: Arraymember einer Struktur zur Zustandsspeicherung

Durch die Benutzung von FFs wird mehr Logik verwendet. Im Gegenzug kann die Latenz sinken, weil es nun im Vergleich mit dem Zugriff auf einen Block-RAM keine Beschränkung der Anzahl an gleichzeitig lesbaren Wörtern gibt.

7.2.5. `if` und `if-else` (IFELSE)

Beim Schreiben von bedingt ausgeführten Anweisungen stellt sich unter anderem die Frage, ob **`if`** oder **`if-else`** verwendet werden sollte. Falls die Bedingungen sich gegenseitig ausschließen, kann auf ein **`if-else`** zurückgegriffen werden. Dadurch wird der wechselseitige Ausschluss direkt im Quelltext ausgedrückt und kann vom Compiler für eine bessere Codegenerierung verwendet werden. Zugunsten einer subjektiv ansehnlicheren Formatierung oder während Restrukturierungsarbeiten am Quelltext wird aber stattdessen auf zwei einfache **`if`**-Anweisungen zurückgegriffen.

```
1 void configure(int netzFrequenz) {
2    if(netzFrequenz == 50) {
3       gewichtungButterworth.configure(samplePerSec, 35);
4    } else if(netzFrequenz == 60) {
5       gewichtungButterworth.configure(samplePerSec, 42);
6    }
```

```
7 }
```

Quelltext 7.11: (Original) Tests von zwei sich gegenseitig ausschließenden Bedingungengen mit `if-else`

```
1 void configure(int netzFrequenz) {
2    if(netzFrequenz == 50) {
3       gewichtungButterworth.configure(samplePerSec, 35);
4    }
5    if(netzFrequenz == 60) {
6       gewichtungButterworth.configure(samplePerSec, 42);
7    }
8 }
```

Quelltext 7.12: (Modifiziert) Tests von zwei sich gegenseitig ausschließenden Bedingungen, ohne `else`

Es besteht das Risiko, dass der Compiler die beiden sich gegenseitig ausschließenden Kontrollflüsse nicht erkennt. Bei VHLS ist das der Fall und führt zu einer Verdopplung der Latenz und einem steigenden Ressourcenbedarf der Schaltung.

7.3. Evaluation

Hier werden die in Abschnitt 6.1 aufgelisteten Performanzkennwerte für jede Modifikation und Implementierung ermittelt, soweit sich der jeweilige Quelltext für die Modifikation eignet (`UNROLL` lässt sich z.B. nur anwenden, falls Schleifen vorhanden sind). Als Testimplementierungen kommen die plattformunabhängigen Software- und HLS-Implementierungen UIP-GEN, Poly-GEN, UIP-HLS-T und Poly-HLS-T-P aus Kapitel 4 zum Einsatz (siehe Tabelle 6.1). In Anhang A sind zusätzlich die Stichprobenverteilungen der Laufzeiten für jede Modifikation und jeden Compiler dargestellt.

In Tabelle 7.1, 7.2 und 7.4 am Ende des Abschnitts werden die Ergebnisse aufgelistet. Bei den CPU-Varianten waren Auswirkungen der Modifikationen auf die Performanzkennwerte oft nicht messbar, da die Compiler entweder den gleichen Assemblercode generierten oder die Auswirkungen sich nicht mit den Messmethoden aus Kapitel 6 reproduzierbar nachweisen ließen. Statt sämtliche Kennwerte in tabellarischer Form anzugeben, werden daher nur die Ausführungszeiten und Dateigrößen ermittelt. Der p-Wert dafür, dass der angegebene Durchschnittswert für die Ausführungszeit einer modifizierten Variante tatsächlich auch mit der unmodifizierten Variante hätte erzielt werden können, wird in der letzten Spalte mit zwei Nachkommastellen angegeben. Dieser Wert wurde durch einen statistischen Signifikanztest mit dem *Welch's t-test* ermittelt (SciPy v1.0.0, *ttest_ind*[32]).

7.3.1. Auswirkungen auf die UIP-Implementierung

Bei der Akkumulation von Werten innerhalb der UIP-Implementierung müssen beim letzten Abtastwert eines Zyklus (`lastSample`) der Durchschnitt und die Quadratwurzel berechnet werden, um die Zyklusergebnisse zu erhalten. Der Akkumulationspuffer muss danach für den nächsten Zyklus auf 0 zurückgesetzt werden. Daher kann der Akkumulationspuffer nicht zum Speichern der Ergebnisse verwendet werden. Die UIP-Implementierung eignet sich daher für die Modifikation `REENTRANT`, bei welcher ein Zeiger auf den Akkumulationspuffer übergeben wird (vom gleichen Typ wie Parameter `result` in Listing 4.5). Die CPU-Implementierung eignet sich nicht für diese Modifikation, da die Akkumulation hier auf dem Ergebnispuffer arbeitet und der verwendende Quelltext den Puffer selbstständig zurücksetzt und für den Aufruf von `average` verantwortlich ist (siehe Listing 4.2).

7.3.2. Auswirkungen auf die Polynominterpolations-Implementierung

Es können die Modifikationen `LICM` und `MANUAL` vorgenommen werden, wie in Abschnitt 7.2 anhand der Polynominterpolation erklärt wird. Bei der CPU-Variante wird für `LICM` bei GCC und ICC der gleiche Code wie für die unmodifizierte Variante generiert. Bei `MANUAL` kann beim GCC für Intel eine Verschlechterung der Laufzeitperformanz um 87 % festgestellt werden. Der Wert für den Intel-Compiler verschlechtert sich hier nur um etwa 18 %. Die Verschlechterungen kommen dadurch zustande, dass bei beiden Compilern keine SIMD-Instruktionen mehr verwendet werden. Stattdessen generiert GCC einen erheblich längeren Code als der Intel-Compiler. Dies macht sich zur Laufzeit an der Zahl der ausgeführten Instruktionen bemerkbar, die bei GCC bei $95 \cdot 10^6$ und beim Intel-Compiler bei $65 \cdot 10^6$ liegt. Für den ARM Prozessor kann zwar eine geringe Verschlechterungen beobachtet werden, die aber mit etwa 1 % deutlich kleiner ist. Der Grund für den kleineren Unterschied im Vergleich zum Intel-Prozessor ist, dass der ARM keine `double`-Vektorinstruktionen besitzt, und daher die unmodifizierte Variante keinen Vorteil aus SIMD-Instruktionen ziehen kann.

7.3.3. Auswirkungen auf die Flickermeter-Implementierung

Die Performanzkennwerte bei der Flickermeter-Implementierung sind in Tabelle 7.4 beschrieben. Die minimale Latenz entspricht dem Durchlaufen eines Abtastwertes durch die digitalen Filter, während der Fall mit maximaler Latenz eintritt, wenn die Schaltung neu konfiguriert wird, z.B. um die Eingangsabtastrate umzustellen.

Global lässt sich beobachten, dass GCC unabhängig von der Modifikation für

den Intel-Prozessor um etwa 3 % schnelleren Code erzeugt als der Intel-Compiler. Für beide Compiler zeigen die Messungen der Modifikationsauswirkungen keine eindeutigen Ergebnisse für den Intel-Prozessor. Eine Ausnahme bildet `REENTRANT` bei GCC, der eine leichte Verschlechterung der Laufzeitperformanz von etwa 2 % zeigt, während der Intel-Compiler zwar auch hier eine Verschlechterung zeigt, die jedoch mit 0,5 % sehr klein ist. Bei dieser Modifikation hängt es vom Layout der Datensektionen des Programms ab, ob und wie hoch ein Performanzverlust oder Gewinn erzielt wird, da die Adressen von Daten bestimmen, in welche Cachelines des Prozessors sie geladen werden und mit welchen anderen Daten sie in Konkurrenz um Cachelines stehen.

Beim ARM-Prozessor zeigt die `UNROLL`-Modifikation starke Auswirkungen. Hier kann eine Verbesserung von ca. 10 % beobachtet werden. Die abgerollte Schleife (siehe 7.2.3) befindet sich in einer Funktion, die nur einmal vor der Nutzung des Flickermeters aufgerufen wird und für die Rekonfiguration der digitalen Filter zuständig ist. Dennoch bewirkt ein Abrollen auch eine Performanzverbesserung im eigentlichen Filterbetrieb, der für jeden Abtastwert aktiviert wird. Der Grund hierfür ist, dass in der nicht modifizierten Variante der Funktionsaufruf innerhalb des Schleifenkörpers von GCC expandiert wird, so wie bereits bei der HLS. Dieser expandierte Aufruf wird dann zusätzlich durch ein automatisches Abrollen der Schleife zwei mal dupliziert. Die Expansion führt dazu, dass vier zusätzliche 64 Bit Register auf dem Stack gesichert werden müssen, da diese von den expandierten Funktionsaufrufen überschrieben werden, obwohl im eigentlichen Filterbetrieb dieser Code nicht durchlaufen wird. In der modifizierten Variante werden die (nach dem händischen Abrollen) drei Funktionsaufrufe hingegen nicht expandiert, sodass keine zusätzlichen Register gesichert werden müssen.

Um zu untersuchen, wie dieses Performanzproblem im Originalquelltext gelöst werden kann, wurden zwei Compilerspezifische Varianten ausprobiert. In Listing 7.13 wurde dies mit `__builtin_expect` versucht. Diese Funktion teilt dem Compiler mit, dass das erste Argument vermutlich den Wert der übergebenen Konstante haben wird. Für `if`-Anweisungen wird das benutzt, um dem Compiler mitzuteilen, welche der beiden Verzweigungen wahrscheinlicher ist. Dadurch kann der wahrscheinlichere Codeblock direkt hinter die *branch*-Instruktion emittiert werden, sodass dieser bereits spekulativ durch den Prozessor ausgeführt werden kann, bevor der bedingte Sprung durchgeführt wurde. Trotz dieses Hinweises hatte sich aber am generierten Assemblercode für beide Verzweigungen nichts geändert, sodass das Performanzproblem weiterhin bestand. Die zweite Variante verwendet eine C++11-Lambda-Funktion, zusammen mit einem GCC-spezifischen Funktionsattribut, das die Funktion als *cold* markiert. Die Lambda-Funktion ist eine unbenannte Funktion, die in diesem Fall nach ihrer Definition sogleich aufgerufen wird. Durch die Markierung wird dem Compiler mitgeteilt, dass die Lambda-Funktion nur selten aufgeru-

fen wird. Durch diese Änderung konnte eine Expansion des `restart`-Aufrufs in die Hauptfunktion effektiv verhindert und das Performanzproblem der unmodifizierten Variante behoben werden.

```
1 if(__builtin_expect(restart, 0)) {
2    impl.config = config;
3    impl.restart();
4 } else {
5    impl.addSample(sample);
6 }
```

Quelltext 7.13: Markierung einer Verzweigung von `if` als *unwahrscheinlich*

```
1 if(restart) [&]() __attribute__((cold)) {
2    impl.config = config;
3    impl.restart();
4 }(); else {
5    impl.addSample(sample);
6 }
```

Quelltext 7.14: Markierung einer aufgerufenen unbenannten Funktion als *kalt*

Tabelle 7.1.: Performanzkennwerte der UIP-Implementierung unter dem Einfluss der Modifikationen

	DSPs	FFs	LUTs	Latenz		II		f_{max} (MHz)
(Total)	80	35 200	17 600	min	max	min	max	Ziel 100
UIP-HLS-T	3	6660	6830	2	108	3	109	109
-MANUAL	3	10 252	12 328	107	107	2	2	116
-REENTRANT	3	6845	7440	2	108	3	109	119

Tabelle 7.2.: Performanzkennwerte der Polynominterpolations-Implementierung unter dem Einfluss der Modifikationen

(a) Kennwerte für CPU-Variante

	T_r ms		M_i kB	p-Wert
(N)	×1000		×1	
Poly-GEN-XI	3.476	± 0.003	537.660	1.0
Poly-GEN-XG	4.214	± 0.002	314.392	1.0
Poly-GEN-AG	31.299	± 0.007	260.896	1.0
-XI-LICM	3.475	± 0.002	537.660	0.0
-XG-LICM	4.215	± 0.002	314.392	0.0
-AG-LICM	31.374	± 0.007	260.888	0.0
-XI-MANUAL	4.125	± 0.007	537.672	0.0
-XG-MANUAL	7.731	± 0.002	314.452	0.0
-AG-MANUAL	31.705	± 0.009	260.704	0.0

(b) Kennwerte für HLS-Variante

	BRAMs	DSPs	FFs	LUTs	Latenz		Clk/S	f_{max} (MHz)
(Total)	120	80	35 200	17 600	min	max		Ziel 100
Poly-HLS-T-P	16	19	986	915	47	1 179 739	9	135
-LICM	16	20	2597	2681	6	1 179 731	9	141
-MANUAL	18	9	919	449	4	1 310 816	10	138

Tabelle 7.4.: Performanzkennwerte der Flickermeter-Implementierung unter dem Einfluss der Modifikationen

(a) Kennwerte für CPU-Variante. Das Flickermeter wurde mit 500000 Eingabewerten aufgerufen.

(N)	T_r ms ×1000	M_i kB ×1	p-Wert
Flicker-GEN-XI	26.465 ± 0.006	459.648	1.0
Flicker-GEN-XG	25.550 ± 0.006	236.496	1.0
Flicker-GEN-AG	71.422 ± 0.010	181.380	1.0
-XI-LICM	26.464 ± 0.007	459.648	0.05
-XG-LICM	25.550 ± 0.006	236.496	0.40
-AG-LICM	71.422 ± 0.010	181.386	0.07
-XI-UNROLL	26.460 ± 0.006	459.648	0.0
-XG-UNROLL	25.550 ± 0.006	232.492	0.13
-AG-UNROLL	63.347 ± 0.013	179.848	0.0
-XI-REENTRANT	26.593 ± 0.007	459.680	0.0
-XG-REENTRANT	26.089 ± 0.006	236.548	0.0
-AG-REENTRANT	78.938 ± 0.010	181.108	0.0
-XI-IFELSE	26.465 ± 0.006	459.648	0.31
-XG-IFELSE	25.547 ± 0.006	236.496	0.0
-AG-IFELSE	70.669 ± 0.008	180.956	0.0

(b) Kennwerte für HLS-Variante

	DSPs	FFs	LUTs	Latenz		II		f_{max} (MHz)
(Total)	80	35 200	17 600	min	max	min	max	Ziel 100
Flicker-HLS	59	9977	8173	35	205	36	206	116
-LICM	59	9977	8173	35	205	36	206	116
-UNROLL	107	19 018	27523	35	125	36	126	n/a
-UNROLL & -REENTRANT	70	9204	8900	34	168	35	169	115
-REENTRANT	43	5354	5729	34	210	35	211	115
-IFELSE	59	10 964	9376	37	377	38	378	107
-IFELSE & -REENTRANT	43	5356	5743	35	388	36	389	115

8. Interpretation, Zusammenfassung und Ausblick

In diesem Kapitel wird zunächst eine Auswahl der sinnvollsten Zielplattform aus der Sicht des Messgeräteherstellers getroffen. Nach einer kurzen Zusammenfassung der Masterarbeit werden anschließend offene Fragen genannt, die in zukünftigen Arbeiten thematisch aufgegriffen werden könnten.

8.1. Wahl einer Zielplattform

Die Auswahl der ökonomisch sinnvollsten Zielplattform geschieht unter Berücksichtigung der *Performanzkennwerte* aus Kapitel 6 und Kapitel 7. Dafür werden in Tabelle 8.1 zunächst die Hauptkategorien CPU-Ausführung (*Software*), FPGA-Ausführung mit *HLS* und FPGA-Ausführung mit *VHDL* anhand der qualitativen Kriterien *Wartbarkeit*, *Skalierbarkeit*, *Integrierbarkeit* in das Gesamtsystem und funktionaler *Verifizierbarkeit* beurteilt. Hierzu wird ein subjektives Bewertungsschema von jeweils eins bis fünf Punkten verwendet.

Für die Wartbarkeit der HLS wurden drei Punkte vergeben, da es bei leichten Änderungen im Quelltext zu einer Synthese mit zu viel Ressourcenbedarf kommen kann. Außerdem müssen für Softwareprogramme die nötigen Schnittstellen program-

Tabelle 8.1.: Subjektive Einschätzung der Eignung von reiner Softwareentwicklung, HLS und VHDL aus der Sicht des Autors

	Wartbarkeit	Skalierbarkeit	Integrierbarkeit	Verifizierbarkeit
Software	●●●●○	●●●●○	●●●●○	●●●●○
HLS	●●●●○	●●●●○	●●●○○	●●●●○
VHDL	●●○○○	●●●●○	●●●●○	●●●●○

miert werden, falls als Beschleuniger HLS oder handgeschriebenes VHDL eingesetzt wird, damit die Eingabedaten an das FPGA geschickt und die Ergebnisdaten wieder empfangen werden können. Die Wartbarkeit bei VHDL wurde mit zwei bewertet, da die zusätzlich notwendige Beschreibung des Zeitverhaltens es erschwert, das funktionale Verhalten der Schaltung nachzuvollziehen. Die Skalierbarkeit für alle Plattformen wurde mit vier bewertet. Die Skalierbarkeit bezeichnet hier die Skalierung einer variablen Mengenanzahl innerhalb des generierten IP-Cores oder Programms. Hier bestimmt die zur Verfügung stehende Rechenzeit und die verfügbaren Ressourcen das Skalierungslimit. Die Integrierbarkeit bezieht sich auf die Möglichkeit, mehrere unterschiedliche Implementierungen in das System integrieren zu können. Für die HLS gab es hier nur eine Bewertung mit drei Punkten, da VHLS jeweils nur separate IP-Cores erstellt, die voneinander isoliert aus C++-Quelltext synthetisiert werden. Bei mehreren unterschiedlichen IP-Cores führt das zu Problemen, da das Tool die zur Verfügung stehenden Ressourcen nicht bedarfsgerecht auf die generierten IP-Cores verteilen kann. Nach Rücksprache mit den Mitarbeitern des Messgeräteherstellers wurden für die funktionale Verifizierbarkeit bei VHDL vier Punkte vergeben, weil Testbenches mit generischem VHDL unter Mitbenutzung eines VHDL-Simulators es gestatten, die entwickelte Schaltung zu testen. Ähnliches gilt auch für HLS, bei der mit VHLS C++-Testbenches für diesen Zweck eingesetzt werden können (siehe Abschnitt 2.3).

Nach einer Rücksprache mit dem Messgerätehersteller können die Laufzeitperformanzen der entwickelten Software-Implementierungen auf dem ARM-CPU aus Kapitel 5 als ausreichend schnell eingestuft werden. Die bisherige Einschätzung hierzu basierte auf einem Test vor einigen Jahren, der eine Laufzeit von ca. 1 s für die Verarbeitung von $1,2 \cdot 10^6$ Abtastwerten ergab. Das entsprach bei einer Abtastrate von 1,2 MS/s einer Systemauslastung von 100 %. Da dort aber versehentlich eine GCC-Version verwendet wurde, die Fließkomma-Operationen in Software berechnete (siehe Abschnitt 5.4), kann die optimierte Implementierung dieser Arbeit mit ca. 20 ms für $1,2 \cdot 10^6$ Abtastwerte einen deutlichen Geschwindigkeitsvorteil erzeugen (siehe Tabelle 6.2). Es wird also zunächst keine Beschleunigung mithilfe des FPGA benötigt.

Unter Berücksichtigung der qualitativen Merkmale und Performanzkennwerte ist zur Zeit die Benutzung von Implementierungen zur CPU-Ausführung nach Meinung des Autors am sinnvollsten.

8.2. Zusammenfassung

Die zentrale Fragestellung dieser Masterarbeit war, ob die HLS für ein SoC mit FPGA als weitere Zielplattform für ein sich möglicherweise weiterentwickelndes Softwareprojekt neben existierenden CPU-Zielplattformen aufgenommen werden kann.

Außerdem wurde untersucht, inwieweit bereits eine Softwareentwickler-freundliche Arbeitsweise durch die HLS gegeben ist, sodass sich Entwickler auf Spezifikationen und Korrektheit konzentrieren können, anstatt auf Eigenheiten des Syntheseprozesses. Die HLS-Software Vivado HLS von Xilinx wurde für diesen Zweck wie in Abschnitt 2.3 beschrieben verwendet.

Damit die entwickelte Implementierung als Software und Hardware gleichermaßen gut läuft, wurde in Abschnitt 4.1 auf C++ Templates zurückgegriffen. Den Templates wurden, je nach Zielplattform, geeignete Typen übergeben, welche für die Speicherung von Werten verwendet werden (vgl. Listing 2.3). Von diesen Typen wurden durch die Verwendung sogenannter Metafunktionen auch alle benötigten Zwischentypen zur Speicherung von Zwischenergebnissen abgeleitet. Abschnitt 6.1 zeigt, dass speziell für die eingesetzten Intel und ARM CPUs optimierter Quelltext im Ergebnis kleinere Binärdateien, weniger Arbeitsspeicher und schnelleren Maschinencode erzeugt. Dabei hängt es stark von der konkreten Implementierung ab, wie hoch der Geschwindigkeitsgewinn ausfällt.

Während der Bearbeitung sind Probleme mit der HLS aufgefallen. VHLS synthetisiert teilweise Architekturen, die zu viele Ressourcen benötigen und daher nicht realisiert werden können, obwohl häufig alternative Architekturen realisierbar sind. Vermutlich hängt das mit zu früh und häufig allozierten Operator-Instanzen zusammen (Abschnitt 2.4), auf die zulasten einer gesteigerten Latenz verzichtet werden müsste. Schon durch kleine Änderungen am C++-Quelltext, z.B. ein mehrfaches Aufrufen von Funktionen, anstatt gleiches in Form einer Schleife, kann es zu solchen nicht realisierbaren Architekturen kommen. Ein weiteres Hindernis, die HLS effizient als Methodik für die Beschleunigung von ausgewählten Softwareimplementierungen zu verwenden, liegt in der festen Schnittstelle von generierten IP-Cores (vgl. Abschnitt 7.2.4 mit fixer Block-RAM Schnittstelle an der Modulgrenze). Dies ist wichtig, um die Kompatibilität mit angrenzenden Modulen in einer existierenden HDL-Strukturbeschreibung zu gewährleisten. Es limitiert aber die Gestaltungsfreiheiten des HLS-Tools. Ein alternatives Entwicklungsmodell, bei dem dieses Problem nicht zu bestehen scheint, ist die Verwendung einer systemübergreifenden Entwicklungsmethodik (siehe Abschnitt 8.4).

8.3. Vergleich mit vorherigen Arbeiten

In dieser Masterarbeit wurde keine quantifizierende Analyse der Designproduktivität [27] mit HLS vorgenommen. Angesichts der vergleichsweise geringen Expertise des Autors im Bereich der HDLs hätte eine solche Analyse auch keine große Aussagekraft. Trotzdem lässt sich als Ergebnis eine hohe Effizient der HLS feststellen, weil durch die hoch abstrahierenden Direktiven Einfluss auf das Syntheseergebnis von VHLS genommen werden kann.

Wie bereits in [14], konnte auch in dieser Arbeit beobachtet werden, dass durch einen gezielten Einsatz von Direktiven der Ressourcenbedarf bei HLS signifikant reduziert werden kann (vgl. Abschnitt 7.2.2). Bei der UIP-Implementierung konnte durch den Einsatz der Direktiven zudem eine 50-fache Verbesserung der Durchsatzleistung erzielt werden. Hervorgehoben muss die zum Teil deutliche Diskrepanz der Performanzkennwerte zwischen den HLS und VHDL Implementierungen (vgl. Tabelle 6.4), die in [14] deutlich geringer ausfällt. Die Gründe liegen zum einen daran, dass der hier getestete C++-Quelltext plattformunabhängig geschrieben ist, während der Fokus in [14] auf bester Leistung hinsichtlich der HLS liegt. Zum anderen wurde die VHDL-Beschreibung in dieser Masterarbeit von einem Softwareentwickler geschrieben, der zuvor keine VHDL-Kenntnisse aufzuweisen hatte.

8.4. Ausblick

Ein Vergleich der generischen und optimierten Implementierungen der Polyninterpolation zeigt Schwachstellen der Vorgehensweise auf, mit der die Abstraktion von Fließ- und Festkommadarstellungen erreicht wurde. Da der Implementierung weder Angaben über die geforderte Genauigkeit, noch über den Dynamikumfang der Eingabewerte übergeben werden, wechselt die generische Implementierung an einigen Stellen auf Zwischenberechnungen mit `double`, obwohl auch die Genauigkeit von `float` mit sechs Dezimalstellen ausreichend genau wäre (vgl. Abschnitt 5.2.2). Hier könnten in zukünftigen Arbeiten noch weitere Verbesserungen erzielt werden.

Eine Verbesserung könnte vermutlich zusätzlich erzielt werden, wenn die geforderte Genauigkeit, also die letztlich weiterverwendeten Bits des Ergebnisses, angegeben wird. Darauf aufbauend könnte dann eine rückwärts gerichtete Datenflussanalyse über den Datenpfad der Implementierung iterieren und diese Angaben in die einzelnen Operatoranwendungen propagieren. Ähnliches wurde bereits wissenschaftlich untersucht[7], allerdings in der Form von Compiler-Optimierungen. Eine Datenflussanalyse zur Übersetzungszeit in der Form von C++ Templates könnte in zukünftigen Arbeiten ebenfalls eine Rolle spielen.

Die Wartbarkeit und Skalierbarkeit der HLS kann erhöht werden, indem eine integrierte Entwicklungsumgebung benutzt wird, die sowohl den C++ Software-Quelltext, als auch den C++ HLS-Quelltext verwaltet. Mit einer solchen Umgebung ist es möglich, einzelne Funktionen selektiv (engl. *partitioning*) über eine HLS durch das FPGA beschleunigen zu lassen. Die Schnittstellen für den Datentransport werden automatisch erstellt. Daher können von der HLS dann auch Architekturen synthetisiert werden, die keine festen Schnittstellen für die Hauptfunktion benötigen, aber die Datenverarbeitung beschleunigen oder ressourcensparender sind. Eine solche Umgebung ist SDSoC von Xilinx [40]. In zukünftigen Arbeiten könnte die Effektivität einer solchen Methodik untersucht werden.

A. Verteilung der Stichproben aus T_r

Hier sind die Stichproben-Verteilungen über die gemessenen Zeiten für jede Modifikation aus Kapitel 7 dargestellt. Zusätzlich eingezeichnet wird ein Box-Whisker-Plot mit dem Median, dem 25 % und 75 % Perzentilen, sowei unteren und oberen Whiskern mit dem 1,5-fachen des Interquartilabstands. Die optimierten Varianten der Implementierung zeigen stets zwei Peaks. Vermutlich gehen diese Eigenschaften auf Cache-Effekte im CPU zurück, die sich bei der vektor-basierten Lade- und Verarbeitungsstrategie zeitlich weniger gestreut bemerkbar machen könnten als bei den ohnehin speicherzugriffsintensiven, nicht explizit optimierten Varianten.

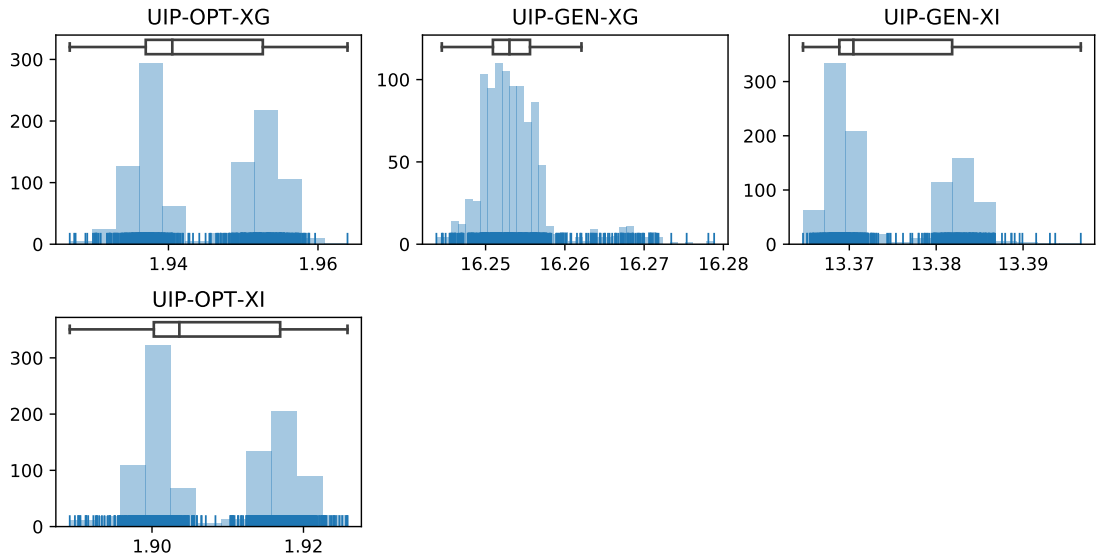

Abbildung A.1.: Verteilung der Stichproben aus T_r für UIP-GEN und UIP-OPT beim Intel Prozessor

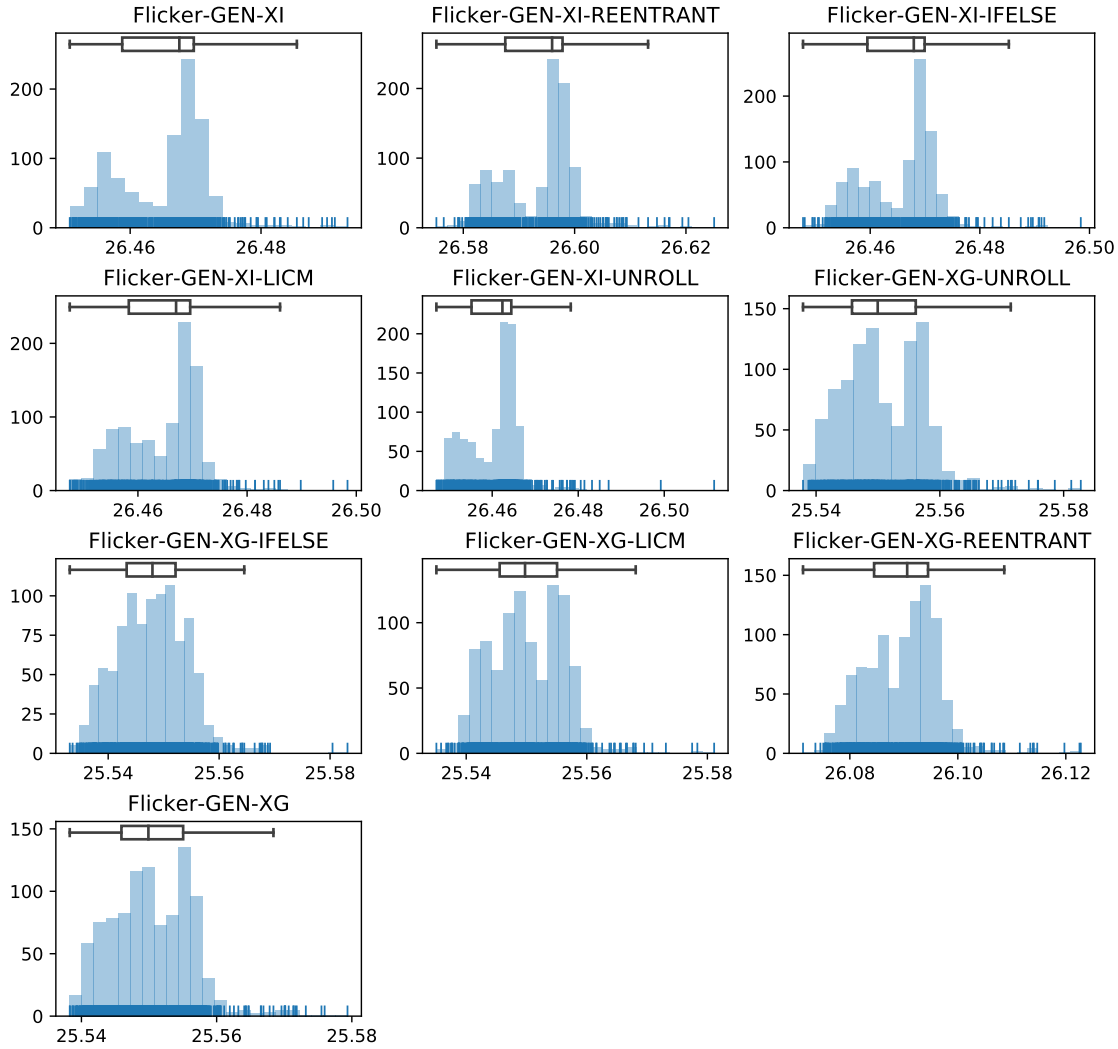

Abbildung A.2.: Verteilung der Stichproben aus T_r für Flicker-GEN beim Intel Prozessor

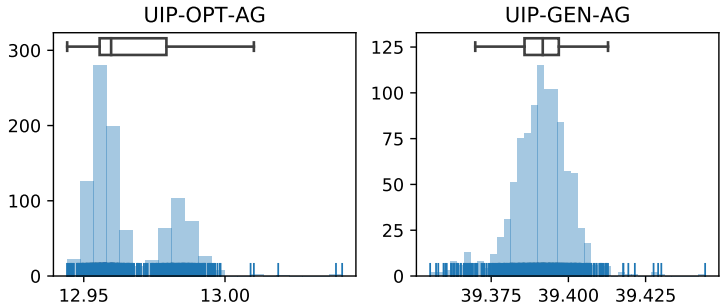

Abbildung A.3.: Verteilung der Stichproben aus T_r für UIP-GEN und UIP-OPT beim ARM Prozessor

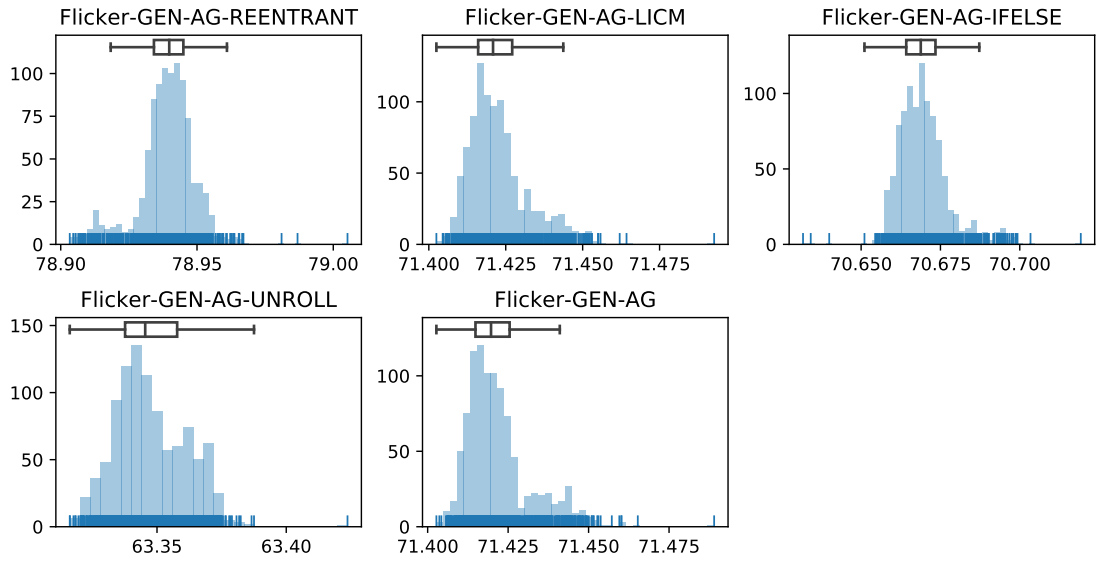

Abbildung A.4.: Verteilung der Stichproben aus T_r für Flicker-GEN beim ARM Prozessor

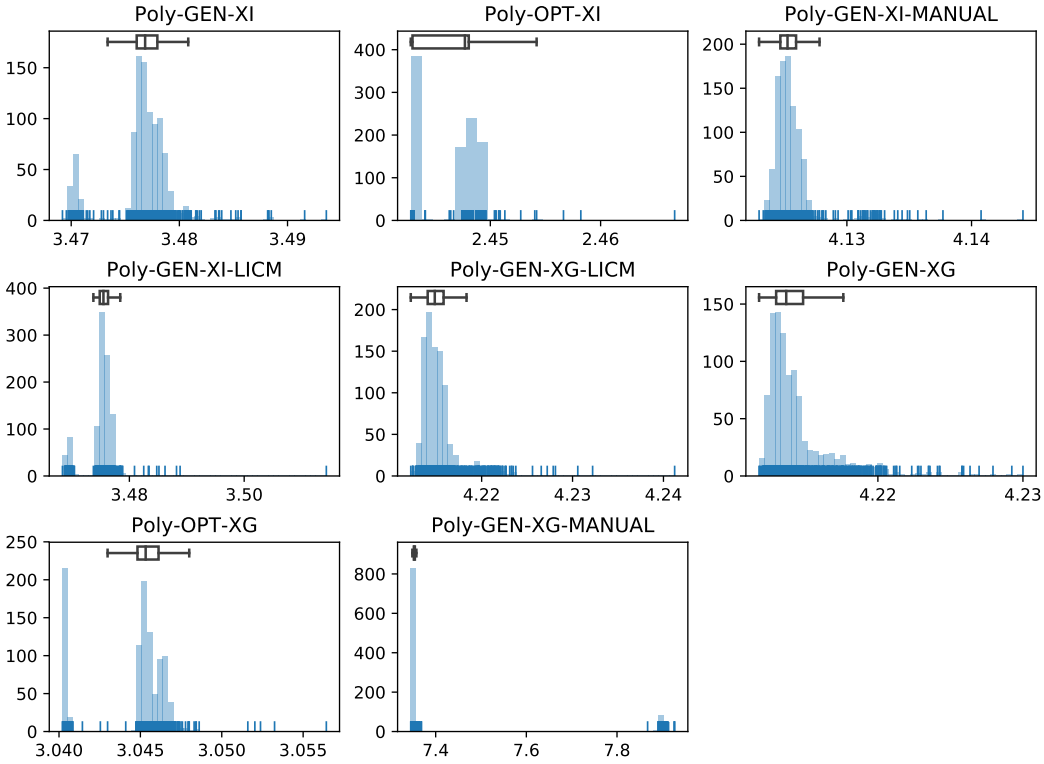

Abbildung A.5.: Verteilung der Stichproben aus T_r für POLY-GEN und POLY-OPT beim Intel Prozessor

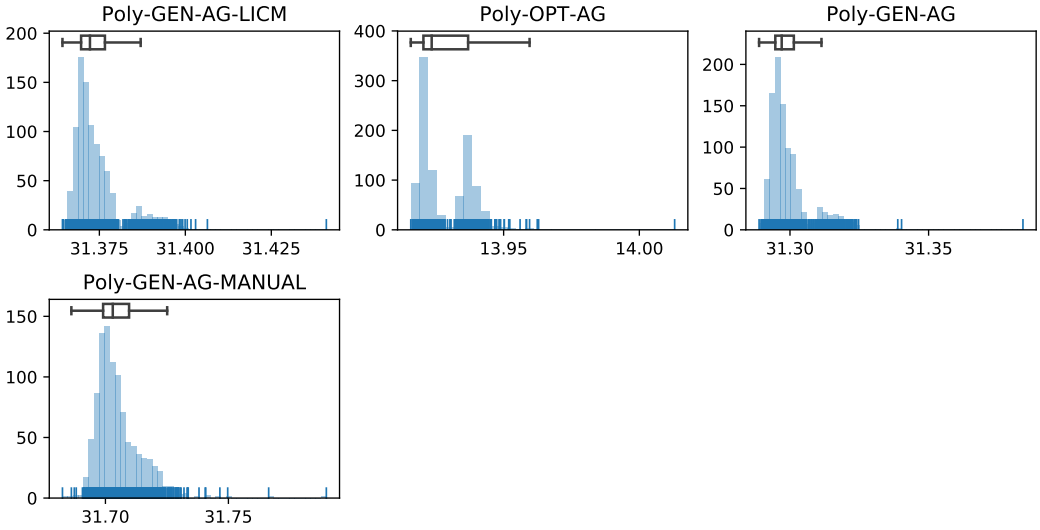

Abbildung A.6.: Verteilung der Stichproben aus T_r für POLY-GEN und POLY-OPT beim ARM Prozessor

B. Weiterführende Informationen zu Compiler und Testumgebungen

Um eine Reproduktion der Messergebnisse zu erleichtern, sind hier die Kommandozeilenparameter der verwendeten Compiler angegeben. Zusätzlich wird die Umgebung zum Messen der Laufzeitgeschwindigkeiten näher spezifiziert.

B.1. Verwendete Compiler-Optionen

Zum Compilieren der Implementierungen wurden den unterschiedlichen Compilern jeweils die folgenden Parameter übergeben.

```
1 -O3 -inline-level=2 -std=c++14 -xATOM_SSE4.2 -mtune=silvermont -m32 $(MODFLAG)
```

Quelltext B.1: Verwendete Optionen für den Intel-Compiler ICC

```
1 -O3 -Wall -c -march=slm -mtune=slm  -funsafe-math-optimizations -ffinite-math-only
2     -m32 -fno-PIE -mfpmath=sse  $(MODFLAG)
```

Quelltext B.2: Verwendete Optionen für den GCC beim Intel-CPU. Das Flag `-ffinite-math-only` wurde nicht beim Flickermeter eingesetzt.

```
1 -O3 -Wall -c -mcpu=cortex-a9 -mfpu=neon -funsafe-math-optimizations
2     -ffinite-math-only $(MODFLAG)
```

Quelltext B.3: Verwendete Optionen für den GCC beim ARM-CPU. Das Flag `-ffinite-math-only` wurde nicht beim Flickermeter eingesetzt

In Abschnitt 5.2 und Abschnitt 5.2.1 wurde die Bedeutung von `-funsafe-math-optimizations` und `-ffinite-math-only` beschrieben. Hier werden außerdem noch die Ziel-Mikroarchitektur an den Compiler übergeben und die explizite Generierung von 32 Bit Code angefordert. Die Option `-mfpmath=sse` fordert die Benutzung von Skalar-SSE Instruktionen beim GCC. Ohne diese Option generiert GCC zwar SSE-Instruktionen für SIMD-Operationen. Für skalare Operationen, bei denen nicht parallel verarbeitet wird, werden bei 32 Bit Zielplattformen aber standardmäßig Instruktionen der x86-FPU (auch als x87 bezeichnet) verwendet. Zwi-

schenzeitliche Performanzmessungen ergaben, dass dadurch Performanzverluste erfolgen. Beim Intel-Compiler wird auch ohne die Verwendung spezieller Flags auf SSE-Instruktionen für skalare Operationen zurückgegriffen.

Die Verwendung von `-fno-PIE` beim GCC für den Intel-CPU ist erforderlich, da ansonsten sogenannte *position-independent executables* erzeugt werden. Dabei kann der Code einer ausführbaren Datei an eine beliebige Stelle in den Speicher geladen und ausgeführt werden. Zugriffe auf das Datensegment des Prozesses zur Laufzeit müssen dann auf das Datensegment über eine PC-relative Adressierung zugreifen, da die absolute Adresse des Datensegments unbekannt ist (*program-counter relative*, die Addition eines Offsets zur Adresse der aktuell ausgeführten Instruktion). Unter anderem wird dadurch die Ausnutzung von Sicherheitslücken erschwert (*address space layout randomization*). Der verwendete GCC ist standardmäßig für eine 64 Bit Architektur konfiguriert und benutzt daher diese Technik. Bei der 32 Bit x86 Architektur kann auf das PC-Register aber nicht direkt zugegriffen werden, sodass diese Adressierung dort mit einem Mehraufwand verbunden ist und ein zusätzliches Register vom ohnehin knappen Registervorrat verwenden muss.

Durch die Umgebungsvariable `$(MODFLAG)` werden unterschiedliche Makros definiert, die im Quelltext die unterschiedlichen Modifikationen aus Abschnitt 7.2 umsetzen.

B.2. Näheres zur Laufzeitumgebung

Unter Abschnitt 6.1.1 wurde die Ausführungsumgebung bereits kurz beschrieben. Zusätzlich wurde dem Kernel der Bootparameter `isolcpus=1,2,...,N-1` übergeben, welcher alle Prozessorkerne außer den ersten *isoliert*. Sie stehen dann nicht mehr für die Ausführung von Programmen zur Verfügung, es sei denn, ein Programm wurde explizit auf einen der Kerne verschoben (*pinning*). Zum Messen der Laufzeiten wurden die Implementierungen wie im folgenden Listing angegeben ausgeführt.

```
1 schedtool -F -p 1 -e ./$k -q --samples=1000 --reporter=csv
```

Quelltext B.4: Kommandozeile zum Messen der Laufzeit einer Implementierung. Hierbei steht `$k` für den Namen der ausgeführten Datei

Der Parameter `-F` weist dem Prozess eine sogenannte *realtime* Scheduling-Policy mit Priorität eins zu (`-p 1`). Der Linux-Kernel reserviert standardmäßig nur 95 % der verfügbaren CPU-Zeit für Echtzeitprozesse und den Rest der Zeit für normale Prozesse. Dies wurde durch das folgende Kommando geändert, da es mit der Standardeinstellung zu vielen Ausreißern und stark erhöhten Latenzen kam.

```
1 echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

Quelltext B.5: Reservierung von 100 % der CPU-Zeit für Echtzeitprozesse

C. Performanz und Ressourcenbedarf für mehrere HLS Zielfrequenzen

Über die Arbeit hinweg wurde als Zielfrequenz für VHLS stets 100 MHz, also eine 10 ns Taktperiode eingestellt (siehe Abschnitt 2.4.1). Um herauszufinden, inwiefern sich die Performanzkennwerte und der Ressourcenbedarf ändern, falls diese Zielfrequenz angepasst wird, wurden Taktfrequenzen von 100 MHz bis 300 MHz in 50 MHz-Schritten eingestellt. Die Ergebnisse dieser Untersuchung sind in Tabelle C.1 aufgelistet. Die betrachteten HLS-Implementierungen sind die aus Kapitel 6, inklusive den Fließkommavarianten UIP-HLS-F1, UIP-HLS-F-S und Poly-HLS-F-P aus Kapitel 4 in Tabelle 4.3 und 4.6.

Tabelle C.1.: Latenz, Initiierungsintervall und Ressourcenbedarf bei unterschiedlichen HLS Zielfrequenzen. Die roten Einträge über- oder unterschreiten jeweils ihren geforderten Wert.

	f_{HLS} (MHz)	DSPs	FFs	LUTs	Latenz		f_{max} (MHz)
(Total)		80	35 200	17 600	min	max	
UIP-HLS-F1	100	24	7936	9557	11	148	102.987
	150	24	13 070	9367	14	263	157.953
	200	24	15 470	8998	23	276	221.239
	250	24	16 105	9035	29	282	223,164
	300	24	16 233	9267	29	282	224,770
UIP-HLS-F-S	100	24	3075	4087	11	11	104.275
	150	24	4222	3869	14	14	153.139
	200	24	6556	3568	23	23	220.604
	250	24	7220	3603	29	29	233,318
	300	24	7358	3714	29	29	229,148
UIP-HLS-T	100	3	6660	6830	2	108	112.867

Tabelle C.1.: (Fortführung)

	f_{HLS} (MHz)	DSPs	FFs	LUTs	Latenz		f_{max} (MHz)
(Total)		80	35 200	17 600	min	max	
	150	3	28 783	27420	2	138	n/a
	200	3	29 383	27400	5	144	n/a
	250	3	30 406	27408	5	149	n/a
	300	3	31 699	27414	7	152	n/a
UIP-HLS-T-S	100	3	378	668	2	2	158.730
	150	3	378	668	2	2	164.962
	200	3	729	355	5	5	215.146
	250	3	796	390	5	5	268.025
	300	3	1168	357	7	7	315.557
Poly-HLS-F-P	100	25	4055	4397	109	1 310 981	104.800
	150	25	6495	4490	153	1 311 079	156.006
	200	25	8145	4501	195	1 311 157	207.039
	250	25	9269	4827	228	1 311 190	207,598
	300	23	9679	5250	242	1 311 204	228,571
Poly-HLS-T-P	100	19	986	915	47	1 179 739	134.953
	150	19	1037	922	47	1 179 743	161.394
	200	19	3207	3316	48	1 179 768	186,637
	250	19	3492	3395	48	1 179 772	208,073
	300	19	3838	3412	48	1 179 774	235,460

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

ADC Analog Digital Converter

AES Advanced Encryption Standard

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

CPU Central Processing Unit

DSP Digital Signal Processor

FF Flip Flop

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

GPGPU General-Purpose Graphics Processing Unit

HDL Hardware Description Language

HLS High-Level Synthese

IC Integrated Circuit

IDE Integrated Development Environment

IP-Core Intellectual Property

ISA Instruction Set Architecture

JTAG Joint Test Action Group

LSB Least Significant Bit

LUT Lookup Table

MSB Most Significant Bit

NaN Not a Number

OoO Out of Order

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SIMD Single Instruction, Multiple Data

SoC System on Chip

VHDL Very High Speed Integrated Circuit Hardware Description Language

Literatur

[1] Analog Devices. *AD7643*. http://www.analog.com/media/en/technical-documentation/data-sheets/AD7643.pdf. [Zugegriffen am 29.01.2018].

[2] Analog Devices. *AD7691*. http://www.analog.com/media/en/technical-documentation/data-sheets/AD7691.pdf. [Zugegriffen am 29.01.2018].

[3] ARM. *Cortex-A9 NEON Media Processing Engine Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409i/DDI0409I_cortex_a9_neon_mpe_r4p1_trm.pdf. [Zugegriffen am 22.01.2018].

[4] ARM. *Cortex-A9 Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf. [Zugegriffen am 23.7.2017].

[5] B Bailey. „Is High-Level Synthesis Ready for Prime Time?“ In: *EDN Article, June* (2012).

[6] BERTEN DSP S.L. *GPU vs FPGA Performance Comparison*. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf. [Zugegriffen am 7.7.2017]. 2016.

[7] M. Budiu, M. Sakr, K. Walker und S. C. Goldstein. „BitValue inference: Detecting and exploiting narrow bitwidth computations“. In: *European Conference on Parallel Processing*. Springer. 2000, S. 969–979.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown und T. Czajkowski. „LegUp: high-level synthesis for FPGA-based processor/accelerator systems“. In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, S. 33–36.

[9] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers und Z. Zhang. „High-level synthesis for FPGAs: From prototyping to deployment“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), S. 473–491.

[10] *Electromagnetic compatibility (EMC) - Part 4-15: Testing and measurement techniques - Flickermeter - Functional and design specifications*. Standard. International Electrotechnical Commission, Aug. 2010.

[11] A. Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. `http://www.agner.org/optimize/instruction_tables.pdf`. [Zugegriffen am 22.01.2018]. 2017.

[12] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers*. `http://agner.org/optimize/microarchitecture.pdf`. [Zugegriffen am 20.11.2017]. 2017.

[13] G. H. Golub und C. F. Van Loan. *Matrix computations*. Bd. 3. JHU Press, 2012.

[14] E. Homsirikamol und K. Gaj. „Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study“. In: *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE. 2014, S. 1–8.

[15] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown und J. Anderson. „The effect of compiler optimizations on high-level synthesis for FPGAs“. In: *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE. 2013, S. 89–96.

[16] IEEE Standards Association and others. „Standard for Floating-Point Arithmetic“. In: *IEEE 754-2008* (2008).

[17] *Information technology – Programming languages – C++*. Standard. Geneva, CH: International Organization for Standardization, Dez. 2014.

[18] Intel. *Intel Atom® Processor E3800 Product Family: Datasheet*. `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-e3800-family-datasheet.pdf`. [Zugegriffen am 27.01.2018].

[19] Intel. *Intel Intrinsics Guide*. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. [Zugegriffen am 26.7.2017].

[20] Intel. *Intel Silvermont Microarchitecture*. https://software.intel.com/sites/default/files/managed/bb/2c/02_Intel_Silvermont_Microarchitecture.pdf. [Zugegriffen am 25.7.2017].

[21] D. H. Jones, A. Powell, C.-S. Bouganis und P. Y. Cheung. „GPU versus FPGA for high productivity computing“. In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE. 2010, S. 119–124.

[22] T. Jäckle. *Leistungsmessung und deren theoretischer Hintergrund*. http://www.zes.com/download/applikationsberichte/zes_applicat_105_leistungsmessung_d.pdf. [Zugegriffen am 14.10.2017].

[23] E. Meijering. „A chronology of interpolation: From ancient astronomy to modern signal and image processing“. In: *Proceedings of the IEEE* 90.3 (2002), S. 319–342.

[24] A. V. Oppenheim. *Discrete-time signal processing*. Pearson Education India, 1999.

[25] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss und E. S. Chung. „Accelerating deep convolutional neural networks using specialized hardware“. In: *Microsoft Research Whitepaper* 2.11 (2015).

[26] P. G. Paulin und J. P. Knight. „Force-directed scheduling for the behavioral synthesis of ASICs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.6 (1989), S. 661–679.

[27] M. Pelcat, C. Bourrasset, L. Maggiani und F. Berry. „Design Productivity of a High Level Synthesis Compiler versus HDL“. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2016)*. 2016.

[28] C. Pilato und F. Ferrandi. „Bambu: A modular framework for the high level synthesis of memory-intensive applications“. In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, S. 1–4.

[29] R. Martinho Fernandes et al. *Nonius*. https://github.com/libnonius/nonius/. [Zugegriffen am 4.9.2017].

[30] J. Sérot, F. Berry und S. Ahmed. „CAPH: a language for implementing stream-processing applications on FPGAs“. In: *Embedded Systems Design with FPGAs*. Springer, 2013, S. 201–224.

[31] C. E. Shannon. „Communication in the presence of noise“. In: *Proceedings of the IRE* 37.1 (1949), S. 10–21.

[32] The Scipy community. *scipy.stats.ttest_ind*. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html`. [Zugegriffen am 21.12.2017].

[33] M. Vestias und H. Neto. „Trends of CPU, GPU and FPGA for high-performance computing“. In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE. 2014, S. 1–6.

[34] J. Villarreal, A. Park, W. Najjar und R. Halstead. „Designing modular hardware accelerators in C with ROCCC 2.0“. In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE. 2010, S. 127–134.

[35] Xilinx. *Spartan-3 FPGA Family*. `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`. [Zugegriffen am 29.01.2018].

[36] Xilinx. *AXI BRAM Controller*. `https://www.xilinx.com/products/intellectual-property/axi_bram_if_ctlr.html`. [Zugegriffen am 29.10.2017].

[37] Xilinx. *AXI General Purpose IO*. `https://www.xilinx.com/products/intellectual-property/axi_gpio.html`. [Zugegriffen am 29.10.2017].

[38] Xilinx. *AXI Interconnect*. `https://www.xilinx.com/products/intellectual-property/axi_interconnect.html`. [Zugegriffen am 29.10.2017].

[39] Xilinx. *Block Memory Generator*. `https://www.xilinx.com/products/intellectual-property/block_memory_generator.html`. [Zugegriffen am 29.10.2017].

[40] Xilinx. *SDSoC Environment User Guide*. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1027-sdsoc-user-guide.pdf`. [Zugegriffen am 24.11.2017].

[41] Xilinx. *Spartan-6 Family Overview*. `https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`. [Zugegriffen am 29.01.2018].

[42] Xilinx. *Vivado Design Suite User Guide, Designing IP Subsystems using IP Integrator*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug994-vivado-ip-subsystems.pdf. [Zugegriffen am 21.10.2017].

[43] Xilinx. *Vivado Design Suite User Guide, High-Level Synthesis*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug902-vivado-high-level-synthesis.pdf. [Zugegriffen am 21.10.2017].

[44] Xilinx. *Xilinx TRD*. http://www.wiki.xilinx.com/Zynq+Base+TRD+2013.3. [Zugegriffen am 10.9.2017].

[45] Xilinx. *Zynq-7000 All Programmable SoC Data Sheet*. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Zugegriffen am 11.10.2017].

[46] Xilinx. *Zynq-7000 All Programmable SoC, Technical Reference Manual*. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Zugegriffen am 23.7.2017].

[47] Xilinx. *Zynq-7000 Processing System IP*. https://www.xilinx.com/products/intellectual-property/processing_system7.html. [Zugegriffen am 21.10.2017].

[48] ZES ZIMMER Electronic Systems GmbH. *User Manual Instrument Family LMG600*. Englisch. Version V1.030. ZES ZIMMER Electronic Systems GmbH. Feb. 2016.

[49] M. D. Zwagerman. *High level synthesis, a use case comparison with hardware description language*. 2015.