Implementierung einer FPGA-basierten Latenzmessung von hardwarebeschleunigten Algorithmen für Hochgeschwindigkeitstrigger

David Mika Thein

Bachelorarbeit in Physik angefertigt im

Physikalischen Institut

vorgelegt der Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

März 2026

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate kenntlich gemacht habe.

Bonn,	....	....
	Datum	Unterschrift

1. Gutachter: Prof. Dr. Klaus Desch
2. Gutachter: Dr. Matthias Hamer

Danksagung

Ich bedanke mich herzlich bei Prof. Klaus Desch und Jochen Kaminski dafür, dass ich meine Bachelorarbeit in der GasDet-Gruppe anfertigen durfte. Ich möchte mich riesig bei Patrick bedanken, der mich nicht nur in die Entwicklung am Versal eingewiesen hat, sondern mir auch tatkräftig zur Seite stand, um tagelang anhaltende Probleme zu lösen, Fehler zu finden und meine zahlreichen Fragen zu klären.

Einen großen Dank möchte ich auch Felix und Luis widmen, die meine Bachelorarbeit gegengelesen und mir unverständliche Stellen rückgemeldet haben, sowie meiner Familie für die liebe Unterstützung.

Ich bedanke mich bei der gesamten GasDet-Gruppe, für die hilfreichen Tipps, die interessanten Gruppenmeetings und die lustigen Konversationen.

Inhaltsverzeichnis

1. Einleitung 6

2. Physikalischer Hintergrund 7

2.1. Lohengrin . . . 7
2.2. Anforderungen an das Lohengrin-Triggersystem . . . 8
2.3. Hardware . . . 9
2.3.1. FPGA . . . 9
2.3.2. Digitaler Signalprozessor (DSP) . . . 10
2.3.3. AXI4-Streams . . . 11
2.3.4. AI Engine (AIE) . . . 12
2.3.5. AMD Versal ACAP . . . 14

3. Software-Entwicklung zur Latenzmessung 17

3.1. Konzept . . . 17
3.2. Entwicklung eines Prototyps . . . 18
3.3. Wechsel auf den AMD Versal . . . 18
3.4. Programmierung von AI Engine & Datengenerator . . . 19
3.4.1. Passthrough-Verhalten . . . 19
3.4.2. Variierende AI Engine-Operationen . . . 20
3.5. Anpassung des Messmoduls für variierende AI Engine-Operationen . 22
3.6. Computerseitige Datenverarbeitung . . . 23

4. Abschätzung der Unsicherheiten 24

4.1. Taktzyklen . . . 24
4.2. Taktfrequenzen . . . 24

5. Durchführung der Latenzmessungen 27

5.1. Latenzzeit für Passthrough-Verhalten ($d = 0$) . . . 27
5.1.1. Taktfrequenz $f_{PL} = 200$ MHz . . . 27
5.1.2. Taktfrequenz $f_{PL} = 625$ MHz . . . 30
5.2. Latenzzeit bei Operationen auf der AI Engine . . . 32
5.3. Bedeutung für Lohengrin . . . 36

6. Fazit 37

A. Messdaten zur Abschätzung der Taktunsicherheiten 38

B. Quelltext **39**

Literaturverzeichnis **48**

1. Einleitung

Die dunkle Materie (DM) gehört zu den größten Rätseln der modernen Elementarteilchenphysik. Es gibt eindeutige Indizien für ihre Existenz, jedoch erweist sich ein direkter Nachweis und ihre Einordnung in das Standardmodell (SM) als schwierig [1][2]. Das LOHENGRIN-Experiment bemüht sich um die Suche nach einem dunklen Photon. Dieses könnte als Pendant zum „normalen“ Photon eine Brücke zwischen dem SM und der DM öffnen. Dazu wird ein Elektronenstrahl auf ein Target gerichtet. In seltenen Fällen könnte dunkle Bremsstrahlung entstehen, die in Form eines scheinbaren Verlusts der Gesamtenergie gemessen wird. [3]

Um hohe Elektronenraten zu erhalten, bedarf das Experiment eines schnellen Triggersystems, das uninteressante Messungen frühzeitig ausschließt. Dafür soll ein neuronales Netz eingesetzt werden, das mit hohem Rechenaufwand verbunden ist. Für LOHENGRIN wird erwogen, das neuronale Netz auf einer FPGA-basierten AMD Versal ACAP zu implementieren. Diese ist mit sogenannten „AI Engines“ ausgestattet, die auf Vektoroperationen und neuronale Netze spezialisiert sind. Allerdings bedeutet der Einsatz von AI Engines zusätzliche Latenzzeiten.

Das ErUM-Data-Projekt *Data Efficiency in Embedded Processors* (DEEP) betrifft ein ähnliches Problem. Die modernen Forschungsmethoden erzeugen große Datenmengen, die herkömmliche Systeme nicht verarbeiten können. DEEP forscht daran, durch maschinelles Lernen (ML) und in die Ausleseelektronik integrierte Prozessoren die Datenmengen zur Laufzeit zu reduzieren. [4] Dafür soll ein Compiler-Framework entwickelt werden, das entsprechende ML-Modelle direkt für die AMD Versal ACAP kompiliert. Damit das Framework möglichst effiziente Modelle realisieren kann, sind Kenntnisse über die Latenzen wichtig. Das Framework soll auch bei dem LOHENGRIN-Experiment eingesetzt werden, um ML-Modelle auf die AMD Versal-Hardware zu übertragen.

Das Ziel dieser Arbeit ist es, ein Firmware-Modul für die AMD Versal ACAP zu entwickeln, das eine einfache Messung der Latenz ermöglicht. Darüber hinaus werden mehrere Messungen vorgenommen, um vorläufig abschätzen zu können, welche FPGA- und AI Engine-seitige Konfiguration eine möglichst geringe Latenz ermöglichen und für LOHENGRIN praktikabel sein könnte.

2. Physikalischer Hintergrund

2.1. Lohengrin

Es gibt verschiedene Ansätze, um die Bestandteile von dunkler Materie einzuordnen. Eine vielversprechende Lösung bieten schwere, schwach-wechselwirkende Teilchen (WIMPs), die über direkte SM-DM-Streuung gemessen werden könnten [5, S. 6 ff., S. 15]. Alternativ wird nach einem Feld aus sehr leichten Axionen gesucht. Für den Nachweis von Axionen wird versucht, ein Axion durch ein starkes Magnetfeld in ein Photon zu konvertieren [6, S. 2090]. Diese Messstrategien greifen jedoch nicht, falls die dunkle Materie aus mittelschweren Teilchen im GeV-Bereich besteht. Einen weiteren Ansatz zur Untersuchung von dunkler Materie erhofft man sich durch das „dunkle" Photon. Bislang ist allein die gravitative Wechselwirkung der dunklen Materie bekannt [2, S. 8]. Bei dem dunklen Photon handelt es sich um ein potenzielles Spin-1-Eichboson und Austauschteilchen, über das die dunkle Materie untereinander wechselwirkt. Es wird vermutet, dass das dunkle Photon schwach an das bekannte elektromagnetische Photon koppelt (etwa über kinetische Mischung), weil es andernfalls zu einer Überproduktion der dunklen Materie im frühen Entwicklungsstadium des Universums gekommen wäre [7, S. 1][3, S. 4].

Durch das LOHENGRIN-Experiment wird versucht, die Existenz von dunklen Photonen nachzuweisen. Am ELSA-Beschleuniger der Universität Bonn soll dafür ein Elektronenstrahl auf ein stationäres Wolfram-Target gerichtet werden. Die Elektronen wechselwirken mit den Atomkernen $\mathcal{H}$ und emittieren Photonen (Bremsstrahlung) γ, oder in seltenen Fällen potenziell ein dunkles Photon („dunkle" Bremsstrahlung) A' [3, S. 6]:

$$e^- + \mathcal{H} \to e^- + \mathcal{H} + \gamma/A'.$$

Während das Photon durch etablierte Messtechniken nachgewiesen werden kann, bleibt eine Messung des dunklen Photons aufgrund dessen schwacher Kopplung aus. Der Aufbau des Experiments wird in Abbildung 2.1 dargestellt. Ein von ELSA erzeugter Strahl aus einzelnen Elektronen wird durch den *Tag Tracker* gelenkt. Dieser besteht aus dünnen Silizium-Pixeldetektoren und prüft die Anzahl und die Position der einfallenden Elektronen. Anschließend treffen die Elektronen senkrecht auf ein Target und erzeugen (potenziell dunkle) Bremsstrahlung. Hinter dem Target misst der *Recoil Tracker* die Trajektorien der Elektronen. Die Tracker und das Target befinden sich in einem externen Magnetfeld, das die Elektronen aus dem weiteren Messaufbau hinauslenkt. Wenn ein Elektron stärker mit dem Target wechselwirkt

und höhere energetische Strahlung freisetzt, wird es stärker durch das Magnetfeld abgelenkt. Dadurch lässt sich aus der Trajektorie auf den Energieverlust des Elektrons und somit auf die Energie der (dunklen) Bremsstrahlung schließen. Die Energie der klassischen Bremsstrahlung wird durch ein elektromagnetisches Kalorimeter (ECAL) gemessen. In seltenen Fällen könnten Hadronen entstehen, die von einem hadronischen Kalorimeter (HCAL) gemessen werden. In diesen Fällen wird das Messereignis verworfen. Falls jedoch dunkle Bremsstrahlung emittiert wurde, misst der Tracker zwar ein einfliegendes Elektron, das Energie verliert, aber die Kalorimeter können das dunkle Photon und somit die fehlende Energie nicht nachweisen. [3, S. 11 ff.]

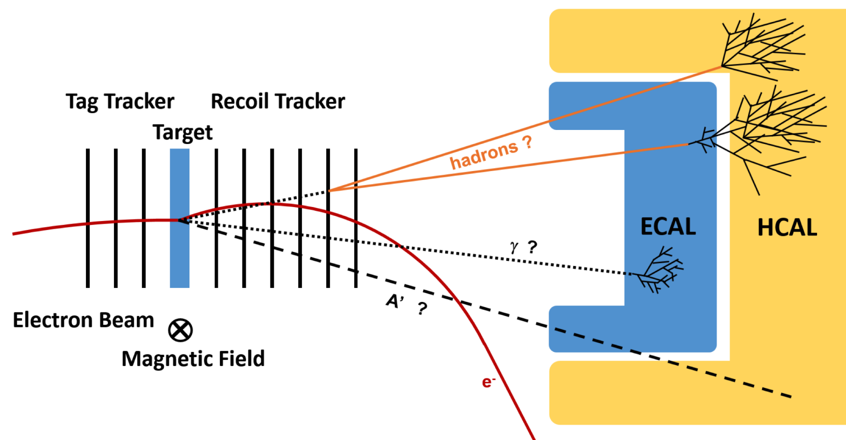

Abbildung 2.1.: Schematische Skizze des Lohengrin-Aufbaus, entnommen aus [3]. ECAL und HCAL bezeichnen ein elektromagnetisches beziehungsweise ein hadronisches Kalorimeter.

2.2. Anforderungen an das Lohengrin-Triggersystem

Lohengrin strebt eine Elektronenrate von 100 MHz an, um in einer angemessenen Zeit ausreichend viele Ereignisse zu messen. Um die Messrate im Aufbau zu reduzieren, wird ein zweistufiges Triggersystem eingesetzt. Das Triggersystem soll auf Elektronen reagieren, die einen bestimmten Schwellenwert an Energie am Target verloren haben. [3, S. 14 f.]

Die erste Triggerstufe – der „L0“-Trigger – nutzt aus, dass niederenergetische Elektronen stärker im Magnetfeld abgelenkt werden als hochenergetische, kaum mit dem Target wechselwirkende Elektronen. Dadurch werden die niederenergetischen Elektronen am *Recoil Tracker* weiter außen gemessen. Der L0-Trigger soll auf Elektronen reagieren, die mit der Trackerebene zunehmende Distanzen zur Ebenenmitte erreichen. Ereignisse mit Elektronen, die näher zur Ebenenmitte gemessen werden,

werden verworfen. Dadurch soll die Ereignisrate um einen Faktor von etwa 10 bis 20 reduziert werden. [3, S. 14 f.]

Um die Ereignisrate weiter einzugrenzen, wird eine zweite Triggerstufe erprobt. Diese basiert auf einer Mustererkennung durch ein neuronales Netz, das die Elektronentrajektorie rekonstruiert, um Parameter wie den Elektronenimpuls zu bestimmen [3, S. 15]. Durch die Vorauswahl des L0-Triggers bleibt der zweiten Triggerstufe eine Zeit von etwa 100 ns bis 200 ns, um eine Entscheidung zu treffen.

Dies könnte durch eine AMD Versal ACAP erreicht werden, die in Abschnitt 2.3.5 beschrieben wird.

2.3. Hardware

2.3.1. FPGA

Ein *Field Programmable Gate Array* (FPGA) ist eine integrierte Schaltung, auf die Hardware-nah logische Gatter und Flipflops konfiguriert werden können [8, Kap. 1, Kap. 3, Kap. 4].

Im Gegensatz zu herkömmlichen Prozessoren und Mikrocontrollern können dadurch effiziente, auf den Anwendungszweck spezialisierte Programme flexibel realisiert werden. Computer werden durch verschiedene Einheiten in der CPU gesteuert. Diese nehmen seriell Instruktionen entgegen und arbeiten mit festen Datenwortgrößen (z. B. 64 Bit). Bei FPGAs hingegen werden durch die direkte Konfiguration logischer Gatter parallele Operationen für eine „beliebige“ Datenwortgröße ermöglicht. Dadurch sind größere Bandbreiten bei geringeren Taktraten möglich [9, S. 167].

Die logischen Gatter werden durch Lookup-Tabellen (LUTs) realisiert. Zur Speicherung von binären Zuständen (Bits) dienen taktgesteuerte D-Flipflops. LUTs und D-Flipflops bilden *Configurable Logic Blocks*, aus denen sich die programmierbare Logik (PL) von FPGAs zusammensetzt. [8, Kap. 4][10]

Die Programmierung von FPGAs geschieht in Hardwarebeschreibungssprachen, wie zum Beispiel Verilog oder VHDL. Bis das Programm auf dem FPGA ausgeführt wird, durchläuft es mehrere Schritte. In der Synthese wird die Hardwarebeschreibung in einen optimierten logischen Schaltkreis beziehungsweise in die primitiven Komponenten des FPGAs (d. h. LUTs und Flipflops) überführt. Anschließend wird das resultierende Schaltbild über ein *Place-and-Route*-Verfahren auf das tatsächliche Hardware-Layout übertragen. Daraus ergibt sich in der Regel eine Datei, die auf den FPGA geladen werden kann und vorgibt, welche LUTs und welche Flipflops wie konfiguriert und verbunden werden müssen. [8, Kap. 2]

Lookup-Tabelle (LUT)

Die Lookup-Tabellen eines FPGAs realisieren logische Gatter. Sie akzeptieren dafür mehrere Eingangssignale (üblicherweise zwei bis sechs) und definieren für jede Kombination von Eingangssignalen ein Ausgangssignal (siehe Abbildung 2.2 links). [8, Kap. 3] So kann beispielsweise ein zweistelliges ODER-Gatter durch eine LUT-2 realisiert werden, wie in Abbildung 2.2 rechts dargestellt.

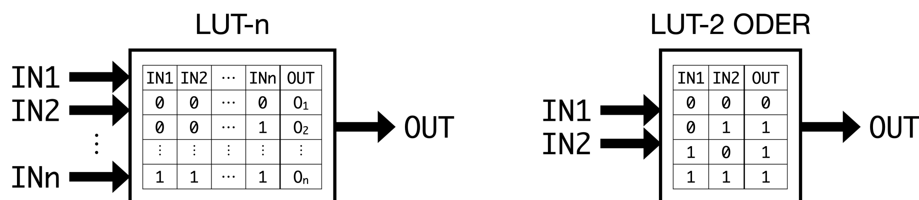

Abbildung 2.2.: Visualisierung einer allgemeinen LUT mit n Eingängen (links) sowie eine beispielhafte LUT-2 zur Realisierung des logischen Oders (rechts). Die Programmierung der LUT wird in dieser Abbildung durch Wahrheitstabellen modelliert.

D-Flipflop

Taktgesteuerte D-Flipflops dienen zur Speicherung von binären Zuständen. Sie verfügen über mindestens drei Eingänge: ein Eingangssignal `D` („Data"), ein Freigabesignal `CE` („Clock Enable") und ein Taktsignal `CLK` („Clock"). Zusätzlich werden oft ein oder mehrere Überschreibungssignale `S/R` („Set/Reset"; anstelle von „Reset" auch „Clear") akzeptiert. Wenn das Freigabesignal auf einer logischen 1 steht, übernimmt das Flipflop das Eingangssignal bei der nächsten (je nach Konfiguration auf- oder absteigenden) Taktsignalflanke. Falls das Freigabesignal auf einer logischen 0 steht, ignoriert das Flipflop das Eingangssignal und behält seinen letzten Wert bei. Über Überschreibungssignale kann der gespeicherte Flipflop-Zustand auf einen bauteilspezifischen Standardwert zurückgesetzt werden. Dies kann asynchron (d. h. sofort) oder synchron (bei der nächsten Taktflanke) passieren. Der Flipflop-Zustand wird durch ein Ausgangssignal `Q` (sowie oft auch invertiert $\bar{Q}$) ausgegeben. [11, Kap. 1.8] Abbildung 2.3 zeigt das Schaltbild eines D-Flipflops und das Zusammenspiel der wesentlichen Signale.

2.3.2. Digitaler Signalprozessor (DSP)

Ein digitaler Signalprozessor ist ein Prozessor, der auf arithmetische Operationen spezialisiert ist. Im Vergleich zu logischen Operationen ist Arithmetik auf FPGAs rechenintensiv, insbesondere die Multiplikation. Für diesen Zweck dienen DSPs, die bestimmte Operationen durch ihre physische Verkabelung implementiert haben. Üblicherweise sind DSPs auf *Multiply-Accumulate*-Operationen (MAC) ausgelegt, bei

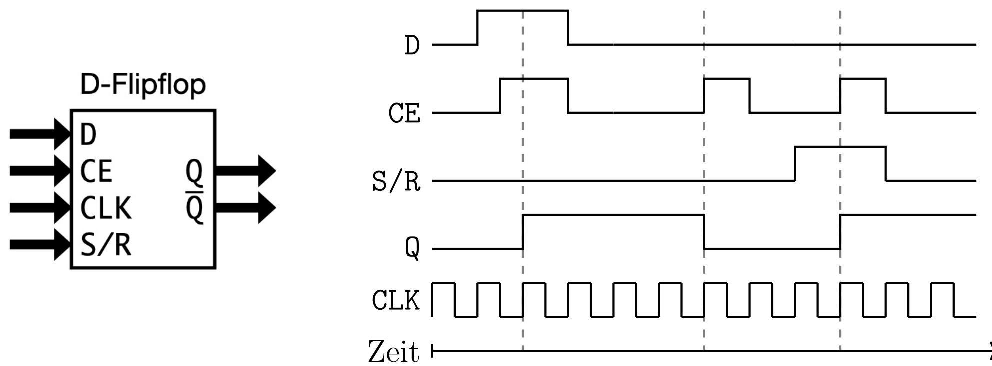

Abbildung 2.3.: Das Schaltbild eines D-Flipflops (links). Rechts wird die Funktionsweise eines D-Flipflops durch dessen Ein- und Ausgangssignale veranschaulicht. Das dargestellte D-Flipflop reagiert auf aufsteigende Taktflanken und setzt durch `S/R` synchron auf die logische `1` zurück.

denen ein Produkt fortlaufend aufsummiert wird [12, S. 173 f.]. MAC-Operationen werden beispielsweise für Filter in der Signalverarbeitung eingesetzt [13, Kap. 1.1]. Durch die parallele Arbeitsweise des AMD Versal können DSPs dort mehrere Filterdurchläufe in einem Taktzyklus zusammenfassen [14].

2.3.3. AXI4-Streams

Das *Advanced eXtensible Interface 4* (AXI4) ist ein Busprotokoll, das unter anderem für die Übertragung der Daten zwischen der PL und der AI Engines eingesetzt wird [15]. Ein AXI4-Stream wird im Wesentlichen durch vier Signale gesteuert [16, Kap. 2][17]:

- `TREADY`: Der Empfänger ist bereit, Daten zu empfangen.
- `TVALID`: Der Sender möchte, dass die auf `TDATA` liegenden Daten empfangen werden.
- `TDATA`: Das Datenpaket des Senders, das von dem Empfänger ausgelesen werden soll.
- `TLAST`: Signalisiert, dass `TDATA` das letzte Datenpaket einer zusammenhängenden Datenkette ist.

Das PL-seitige `TVALID`-Signal startet die Übertragung eines Datenpakets an die AI Engines und eignet sich somit als Startsignal für eine Latenzzeitmessung. Das `TVALID` der AI Engine gibt die Rückkehr der Daten vor und kann somit als Stoppsignal verwendet werden.

2.3.4. AI Engine (AIE)

Die AMD Versal ACAP ist mit AI Engines[1] ausgestattet, die auf anspruchsvolle Vektor-Operationen spezialisiert sind und auch komplexe skalare Operationen lösen können. [19]

Abbildung 2.4 zeigt den Aufbau einer einzelnen AI Engine. Über die *Instruction Fetch & Decode Unit* bezieht die AI Engine sogenannte *Very Long Instruction Words* aus dem Programmspeicher. Diese beinhalten bis zu sieben Instruktionen, die parallel ausgeführt werden können. Die *Scalar Unit* steuert den Programmfluss und führt skalare Rechnungen durch. Die Hauptrecheneinheit befindet sich in der *Vector Unit*, die auf MAC-Operationen für Datenvektoren ausgelegt ist. Das heißt, sie kann eine Instruktion parallel auf mehrere Datenwerte anwenden (*Single Instruction Multiple Data*). Ihre Rechenergebnisse werden in Akkumulatorregistern gespeichert. [20]

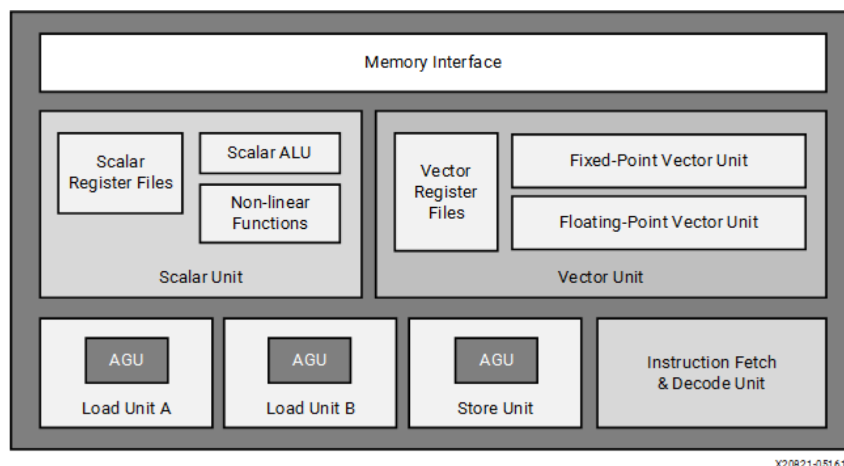

Abbildung 2.4.: Die Architektur einer AI Engine. Die Abbildung stammt von [20].

Jede AI Engine ist mit einem 32-kB-Speichermodul ausgestattet, das mit drei benachbarten AI Engines geteilt wird. (Einer AI Engine unterliegen somit bis zu 128 kB.) Hierdurch und durch die Möglichkeit, die Inhalte der Akkumulatorregister an die nächste AI Engine zu teilen (via *Cascade Streams*), können rechenintensive Operationen auf mehrere AI Engines aufgeteilt werden. [21] Die AI Engines sind an zwei 32-Bit-In- und zwei 32-Bit-Output-AXI4-Streams angeschlossen, die eine Datenübertragung zu der PL ermöglichen [15] (siehe Abbildung 2.5). Die Taktfrequenz der AI Engines (1250 MHz) ist unabhängig von der Taktfrequenz der programmier-

[1] Das „AI“ steht laut AMD nicht zwangsläufig für *Artificial Intelligence*, weil der Anwendungsbereich der AI Engines darüber hinaus gehe [18].

baren Logik des FPGAs (in Verbindung mit den AI Engines maximal 625 MHz).[2] Die AI Engines lassen sich in den Sprachen C und C++ programmieren [19].

Abbildung 2.5.: Die Schnittstellen einer AI Engine. Die AXI4-Streams zur PL werden in Blau dargestellt, die Schnittstellen zu den Datenspeichern in Rot und die Akkumulator-„Cascade“-Streams in Grün. Die Abbildung stammt von [21].

Abbildung 2.6 veranschaulicht den Aufbau der AXI4-Stream-Schnittstelle zwischen der AI Engine und der PL. Die einzelnen Komponenten werden in der zugehörigen Dokumentation nicht ausführlich beschrieben [26]. Bei den BLI-Registern (*Boundary Logic Interface*) handelt es sich um Register, die zwischengeschaltet werden können, um Signale zeitlich besser aufeinander abzustimmen [27]. Im Rahmen dieser Arbeit wurden die BLI-Register nicht explizit eingeschaltet. Die *Register Slicings* sind erforderlich, um hohe Taktfrequenzen zu ermöglichen [28]. Das CDC-Modul (*Clock Domain Crossing*) ermöglicht die Kommunikation zwischen PL und AI Engine, obwohl diese auf verschiedenen Taktfrequenzen arbeiten [29]. Über die Umsetzung konnten keine Informationen gefunden werden. Der *AXI-Stream-Switch* ist vermutlich dafür verantwortlich, den AXI4-Stream mit einer bestimmten AI Engine zu verbinden.

[2] In der Dokumentation von AMD wird die Frequenz des AIE-Takts meistens als 1 GHz und des PL-Takts standardmäßig als 500 MHz angegeben [22]. Bei dem hier verwendeten VCK190-Board lag die *Speed Grade* bei −2 (lt. Modellbezeichnung [23]) und die Spannungsversorgung der AIEs „VCCINT“ bei 0,8 V [24]. Hieraus folgen 1250 MHz (AIE) und 625 MHz (PL) [25].

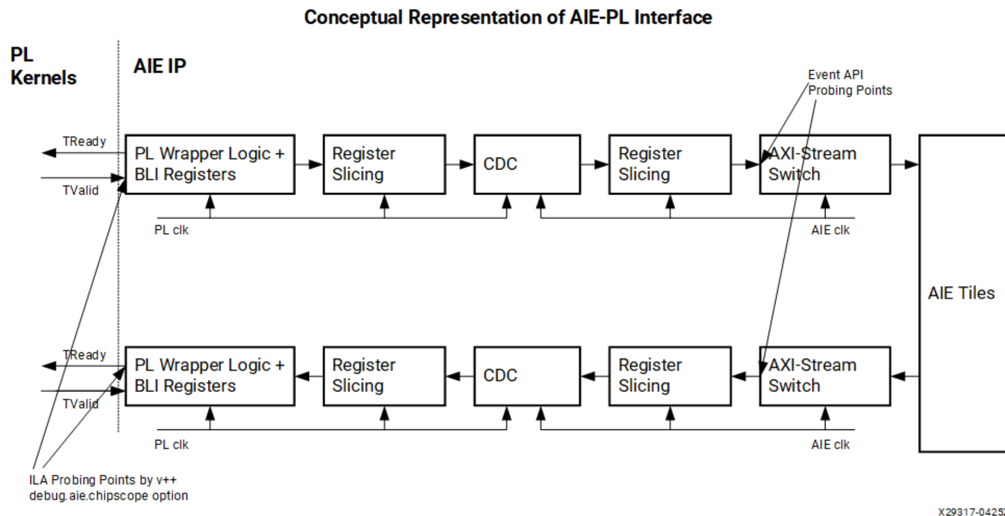

Abbildung 2.6.: Eine konzeptionelle Darstellung der AXI4-Stream-Schnittstelle zwischen den AI Engines und der PL. Die Abbildung stammt von [26].

2.3.5. AMD Versal ACAP

Im Rahmen dieser Arbeit wurde mit dem „AMD VCK190 Evaluation Board“ gearbeitet (Abbildung 2.7), das mit einer AMD Versal AI Core VC1902 ACAP[3] ausgestattet ist und für das LOHENGRIN-Experiment infrage kommt.

Dem AMD Versal liegt ein FPGA zugrunde, der eine hohe Bandbreite aufweist und über DSPs und AI Engines verfügt, um arithmetische Operationen zu beschleunigen. Im Vergleich zu den AI Engines sind die DSPs mit einer geringeren Latenz verbunden und eignen sich für einfache, skalare und schnelle Arithmetik. Die geringe Latenz und schnelle Arithmetik sind prinzipiell auch für LOHENGRIN gewünscht. Allerdings haben die DSPs durch ihre Architektur und ihre begrenzte Anzahl eine niedrigere Bandbreite. Die AI Engines können trotz höherer Latenz mehrere Instruktionen parallel ausführen. Sie verfügen zudem über eine höhere Taktfrequenz und sind auf Vektorarithmetik spezialisiert [19], die das neuronale Netz des LOHENGRIN-Triggersystems beschleunigen könnte.

Die AMD Versal ACAP verfügt über 899 840 Lookup-Tabellen, 1968 DSPs und 400 AI Engines [30]. Abbildung 2.8 zeigt den Aufbau des FPGA-Chips.

[3] ACAP = *Adaptive Compute Acceleration Platform*.

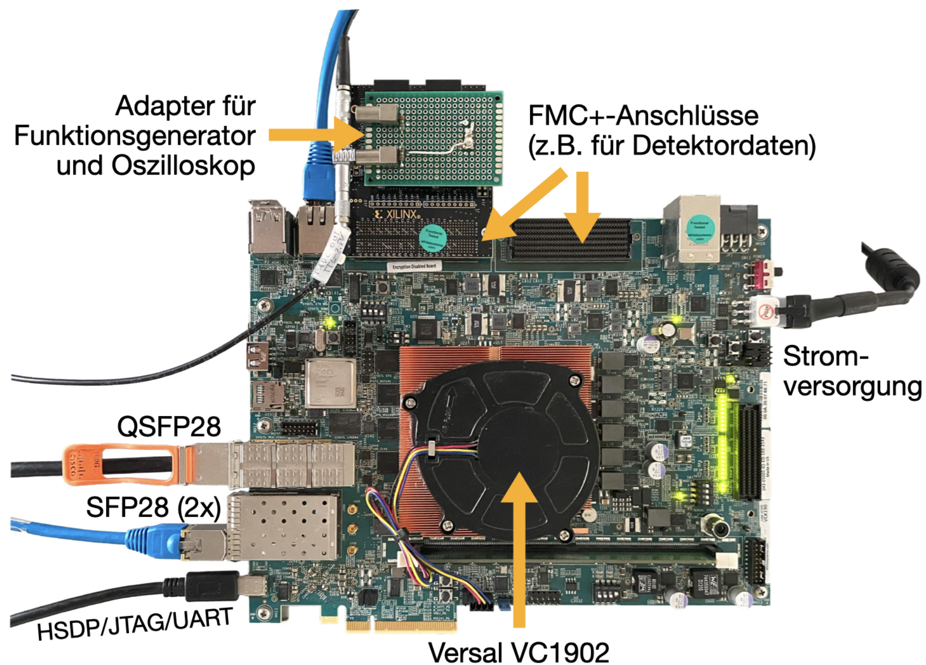

Abbildung 2.7.: Foto des AMD-VCK190-Boards, ausgestattet mit einem AMD Versal AI Core VC1902, zwei FMC+-Anschlüssen, zwei 25-Gbit-SFP28-Ethernet-Anschlüssen, einem 100-Gbit-QSFP28-Ethernet-Anschluss und einem USB-C-Anschluss zur Programmierung und Fehlerbehebung (via HSDP/JTAG/UART) [23]. An einem FMC+-Anschluss wurde ein Adapter angeschlossen, durch den Spannungssignale eingespeist und gelesen werden konnten.

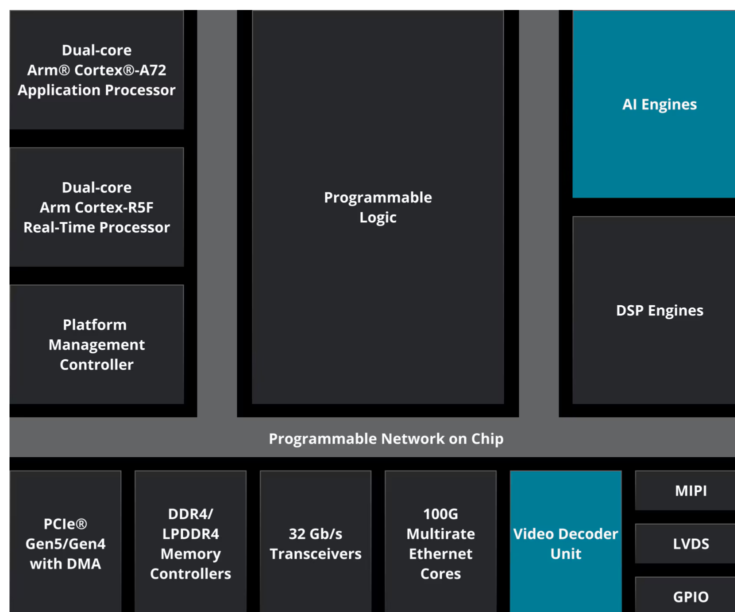

Abbildung 2.8.: Schematische Skizze von dem Aufbau des AMD Versal. Die DSPs sind – anders als das Bild suggeriert – Teil der PL und erhalten ihre Taktfrequenz von ebenda. Die Abbildung stammt von [30].

3. Software-Entwicklung zur Latenzmessung

3.1. Konzept

Das erste Ziel dieser Bachelorarbeit ist die Entwicklung eines einfach auf dem FPGA zu implementierenden Moduls zur Zeitmessung. Dieses akzeptiert drei digitale Signale: eines, um die Zeitmessung zu starten, eines, um sie zu beenden, und eines, um das Ergebnis auszugeben.

Das Modul operiert zunächst in drei Zuständen: `IDLE`, `COUNTING` und `SEND_COUNT` (siehe Abbildung 3.1). Standardmäßig befindet es sich im `IDLE`-Zustand. Mit jeder aufsteigenden Taktflanke wird überprüft, ob das Startsignal von der logischen `0` auf `1` gewechselt ist. In diesem Fall wechselt das Modul in den `COUNTING`-Zustand, in dem ein Zähler mit jedem Taktzyklus um eins erhöht wird. Außerdem prüft das Modul, ob nun das Stoppsignal von `0` auf `1` wechselt. Falls dem so ist, wird die Zeitmessung gestoppt und das Modul wechselt in den `SEND_COUNT`-Zustand. In diesem Zustand wird der Zählerstand N über das Ausgangssignal ausgegeben. Mithilfe der Taktfrequenz f_{PL} kann hieraus die Zeitdifferenz zwischen den Signalen berechnet werden: $t_{\mathrm{L}} = N/f_{\mathrm{PL}}$. Zuletzt kehrt das Modul wieder in den anfänglichen `IDLE`-Zustand zurück.

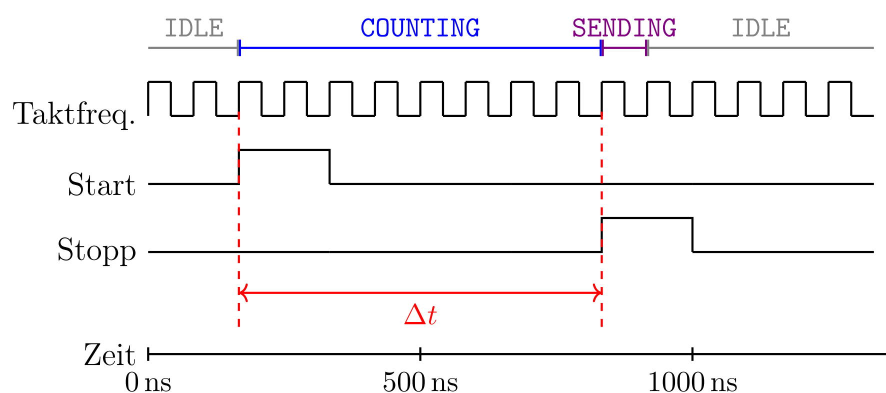

Abbildung 3.1.: Schematische Darstellung der Zeitmessung zwischen einem Start- und einem Stoppsignal mit einem 12-MHz-Takt. Zwischen den Signalen misst ein Zähler im `COUNTING`-Zustand $N = 8$ Taktzyklen und folglich eine Zeit von $t_{\mathrm{L}} \approx 0{,}67\,\mu\mathrm{s}$.

3.2. Entwicklung eines Prototyps

Um die grundlegende Funktionsweise des Moduls zu testen, wurde zunächst ein Prototyp für einen leistungsschwächeren Lattice ICE40HX1K-TQ144-FPGA entwickelt. Auf diesem erfolgt der Syntheseprozess wesentlich schneller als auf dem vorgesehenen AMD Versal.

Das Modul operiert wie in Abschnitt 3.1 vorgesehen. Es akzeptiert zwei Input-Signale `i_inp_start` und `i_inp_stop`, die die Taktzählung starten und stoppen. Über die Ausgangssignale `o_data_output_port` und `o_output_port_ready` wird das Zählergebnis seriell über eine UART-Schnittstelle an einen Computer übertragen. Außerdem muss ein Taktsignal `itm_clk` vorgegeben werden.

Um den Prototyp zu testen, wurden das Start- und das Stoppsignal auf dasselbe Rechtecksignal gesetzt, das von einem externen Funktionsgenerator eingespeist wurde. Das bedeutet, dass jede zweite aufsteigende Signalflanke die Taktzählung startete und die jeweils nächste aufsteigende Signalflanke die Taktzählung stoppte. Das FPGA-Board wurde über ein USB-Kabel mit einem Computer verbunden. Die gemessenen Taktzyklen wurden von einem Python-Skript auf dem Computer empfangen und mithilfe der Taktfrequenz von 12 MHz in die Periodendauer des Rechtecksignals umgerechnet. Daraus folgte eine Periodendauer von $\tau_{\mathrm{FPGA}} = (2{,}627 \pm 0{,}001)\,\mathrm{ms}$. Die Unsicherheit entspricht der Wurzel der Varianz, die aus etwa 12 200 Datenpunkten gemäß [31, S. 89 ff.] berechnet wurde. Mithilfe eines Oszilloskops wurde die Periodendauer des Rechtecksignals auf $\tau_{\mathrm{Osz}} = (2{,}626 \pm 0{,}001)\,\mathrm{ms}$ gemessen. Beide Werte liegen im $1\,\sigma$-Bereich des jeweils anderen und sind somit statistisch miteinander vereinbar.

3.3. Wechsel auf den AMD Versal

Anschließend sollte das prototypische Modul auf den leistungsstärkeren AMD Versal übertragen werden. Dafür wurde das Modul in ein bestehendes Projekt „MiniLORRAINE“ [32] eingefügt. Dieses stellt grundlegende Funktionen zur Verfügung, wie beispielsweise einen Netzwerkstack, und basiert auf der FPGA-Software des LOHENGRIN-Testbeams aus dem November 2025. Nach einigen Problembehebungen konnte das Modul zur Zeitmessung auch an diesem FPGA-Board ein externes Signal als Start- und Stoppsignal empfangen und daraus das Zeitintervall zwischen den aufsteigenden Signalflanken bestimmen. Das Start- und das Stoppsignal wurden erst von einem externen Funktionsgenerator und schließlich von einem FPGA-Modul gesetzt. Die Funktionalität wurde durch Simulationen und die Implementierung eines *Integrated Logical Analyzers* geprüft. Letzterer ermöglicht es, die Signale des AMD Versal am Computer zur Laufzeit zu beobachten. Abbildung 3.2 zeigt eine Bildschirmaufnahme einer solchen Messung. Die Beobachtung deckt sich mit den Messungen in Abschnitt 5.1.1.

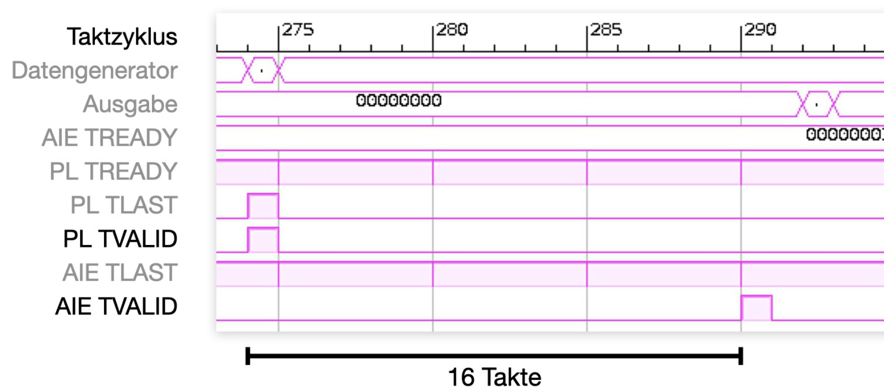

Abbildung 3.2.: Bearbeitete Bildschirmaufnahme aus der AMD-Software *Vivado*. Zu sehen sind verschiedene Signale auf dem FPGA, insbesondere des AXI4-Streams zwischen der AIE und der PL. Ein Datengenerator generiert ein 32-Bit-Wort, das an die AI Engine geschickt werden soll, und startet die Zeitmessung, indem er das PL-seitige `TVALID` auf eine logische `1` setzt. Die AI Engine sendet das Signal zurück und stoppt die Zeitmessung mit dem AIE-seitigen `TVALID`. Zuletzt wird das Messergebnis über den Netzwerk-Stack an einen Computer ausgegeben. (Aufgenommen für eine Taktfrequenz von $f_{\mathrm{PL}} = 200\,\mathrm{MHz}$ im Passthrough-Verhalten, siehe Abschnitt 3.4.1.)

3.4. Programmierung von AI Engine & Datengenerator

Im Rahmen dieser Arbeit wurden die AI Engines im Wesentlichen auf zwei Verhaltensmuster programmiert. Ein separates PL-seitiges Datengenerator-Modul, das die Zeitmessungen startete und stoppte, wurde verwendet, um das Verhalten der AI Engines zu beeinflussen.

3.4.1. Passthrough-Verhalten

Um die Latenzzeit durch die Kommunikation mit einer AI Engine zu messen, schickte ein PL-seitiges Datengenerator-Modul ein Datenpaket von der PL über einen AXI4-Stream an eine AI Engine. Das `TVALID`-Signal des AXI4-Streams startete das Zeitmessmodul. Die AI Engine schickte das Signal unverändert an die PL zurück (*Passthrough*). Sobald die PL das Signal mit dem AI Engine-seitigen `TVALID`-Signal empfing, wurde die Taktzählung des Zeitmessmoduls gestoppt. Die dadurch gemessene Zeit entspricht also der Zeit, die das Signal benötigt, um von der PL zur AI Engine und zurück zu laufen. Das von dem Datengenerator erzeugte Datenpaket

wurde mit zunächst willkürlichem Dateninhalt erzeugt und mit niedriger Frequenz ausgesendet.[1]

Der Quelltext des AI Engine-Programms zur Übertragung von 64-Bit-Wörtern wird unter Quelltext 3.1 aufgeführt. Um 32-Bit-Wörter zu übertragen, kann der Datentyp `int64` durch `int32` ausgetauscht werden. Da es keinen primitiven 128-Bit-Datentyp gibt, müssen die zugehörigen Daten als 32-Bit-Vektor mit vier Elementen durch `readincr_v` eingelesen werden (siehe [33]). Die Datenwortgröße wird im Rahmen dieser Arbeit durch das Formelzeichen s bezeichnet.

```
1 void inoutping(input_stream<int64>* __restrict in,
     output_stream<int64>* __restrict out) {
2      writeincr(out, readincr(in), true);
3 }
```

Quelltext 3.1: AI Engine-seitiges C++-Programm (vereinfacht), das 64-Bit-Eingangsdaten über `readincr` einliest und über `writeincr` wieder zurückschickt (siehe Gitlab). Das `true` im `writeincr`-Aufruf setzt das `TLAST`-Signal im AXI4-Stream und gibt vor, dass das zurückgesendete Paket vollständig ist.

3.4.2. Variierende AI Engine-Operationen

Um zu untersuchen, wie die Latenzzeit mit der Anzahl an Operationen auf der AI Engine skaliert, wurde der Datengenerator so angepasst, dass er eine aufsteigende Zahl d (*Delay*) in einem Bereich von 0–255 an die AI Engine überträgt. Die AI Engine schickte dieses Signal nicht länger direkt zurück, sondern sollte zuvor d Taktzyklen abwarten. Anschließend gab sie d über einen AXI4-Stream an die PL zurück. Die Methodik, über die der Programmfluss pausiert wurde, führte dazu, dass die tatsächlich auf der AI Engine abgewartete Anzahl an Zyklen von d abwich (siehe weiter unten in diesem Abschnitt). Die tatsächliche Anzahl an abgewarteten Zyklen wird im Folgenden als D bezeichnet. D wurde über einen parallelen AXI4-Stream an die PL versendet. Dem Zeitmessmodul auf der PL stehen dann folgende Messgrößen zur Verfügung:

- d: Die Anzahl an AI Engine-seitigen Taktzyklen, die die AI Engine abwarten soll, bevor sie ein eingehendes Datenpaket zurücksendet.
- D: Die Anzahl an AI Engine-seitigen Taktzyklen, die die AI Engine tatsächlich abgewartet hat, bevor sie ein eingehendes Datenpaket zurücksendete.

[1] Die Zeitintervalle zwischen dem Aussenden der Datenpakete variierten zwischen verschiedenen Tests und wurden groß genug gewählt, sodass sie keinen Einfluss auf die Messung hatten.

- N: Die Anzahl an PL-seitigen Taktzyklen, die verstrichen sind, während das Datenpaket auf der PL losgesendet, auf der AI Engine gehalten und anschließend an die PL zurückgesendet wurde.

Die tatsächlich abgewarteten Zyklen D lassen sich im Quelltext durch die Funktion `get_cycles` berechnen [34]. Die Funktion gibt den Zählerstand eines unabhängigen Zählers zurück, der mit jedem Takt auf der AI Engine um eins erhöht wird. Die so gemessene Zahl lässt sich mit dem Assemblercode vergleichen, in den das C++-Programm kompiliert wird (Quelltext B.2). Der C++-Quelltext wird für eine Wortgröße von $s = 64$ Bit in Quelltext 3.2 dargestellt. Die Übertragung anderer Wortgrößen geschieht analog wie in Abschnitt 3.4.1 beschrieben. Um einen größeren Wartebereich von 0–2550 abzudecken, wurde d auf der AI Engine mit dem Faktor 10 multipliziert.

```
1  void inoutping(input_stream<int64>* __restrict in,
     output_stream<int64>* __restrict out, output_stream<int64>*
      __restrict out2) {
2      // Lies Delay ein und speichere Zyklen zur Beginn der
         Funktion
3      const unsigned long d = readincr(in);
4      unsigned long prev_c = get_cycles();
5
6      // Warte d-viele Taktzyklen ab
7      unsigned long stop_at = prev_c + (d);
8      volatile unsigned long current_c = prev_c;
9      do {
10         current_c = get_cycles();
11     } while (current_c < stop_at);
12
13     // Berechne benötigte Zyklen und schicke diese mit d in
         parallelen Streams zurück
14     uint64 c_diff = get_cycles() - prev_c;
15     writeincr<aie_stream_resource_out::a>(out2, d, true);
16     writeincr<aie_stream_resource_out::b>(out, c_diff, true);
17 }
```

Quelltext 3.2: AI Engine-seitiges C++-Programm (vereinfacht), das eine 64-Bit-Zahl d über `readincr` einliest, d-Zyklen abwartet und über zwei parallele Streams d und die tatsächlich abgewarteten Zyklen wieder zurückschickt (siehe Gitlab).

Eine Schwierigkeit bestand darin, den Programmfluss für eine dynamische Anzahl an Zyklen zu pausieren, weil die AI Engine keine Funktion zu diesem Zweck bereitstellte. Der Compiler entfernt scheinbar unnötige Prozesse aus dem Programmfluss, weshalb zum Beispiel eine einfache, leere `while`-Schleife unwirksam ist (trotz des

Versuchs, den Optimierungsprozess durch Compiler-Einstellungen explizit zu unterdrücken). Deshalb wurde auf das C-intrinsische Keyword `volatile` zurückgegriffen. `volatile` kennzeichnet Variablen, die beispielsweise extern zugänglich sein sollen. Für den Compiler ist es nicht möglich, die externe Verwendung dieser Variable zu erkennen, wodurch zugehörige Optimierungen verhindert werden [35][36, Kap. 6.7.3].

In Quelltext 3.2 wurde eine `while`-Schleife mit einer `volatile`-Variablen implementiert. Nach jedem Schleifendurchlauf wird geprüft, ob die gewünschte Verzögerung erreicht wurde. Der Durchlauf der Schleife benötigt eine feste Anzahl an Taktzyklen, die in jedem Schleifendurchlauf durchlaufen werden und die Auflösung einschränken (sogenannter „Overhead“). Das führte dazu, dass ähnlich große gewünschte Verzögerungen d eine gleich große messbare Verzögerung D bewirkten. Ferner wird vermutet, dass das `volatile`-Keyword zu einer Diskrepanz in der theoretischen Verzögerung (gemäß dem Assemblercode) und der tatsächlich gemessenen Verzögerung führte (siehe Abschnitt 5.2).

3.5. Anpassung des Messmoduls für variierende AI Engine-Operationen

In Abschnitt 3.4.2 wurde die Funktionalität der AI Engine so ausgebaut, dass sie über zwei AXI4-Streams mit d und D parallel antwortete. Während einiger Tests wurde festgestellt, dass die Antworten für $f_{\text{PL}} = 625\,\text{MHz}$ und $s = 128\,\text{Bit}$ zwar gleichzeitig von der AI Engine abgeschickt wurden, jedoch nicht gleichzeitig bei der PL ankamen. Der Inhalt des verspäteten Streams wurde dadurch fehlerhaft ausgelesen. Daher wurde das Messmodul auf zwei Stoppsignale für beide AXI4-Streams erweitert, die gleichermaßen die Zeitmessung stoppen. Außerdem wurde das Modul um die zwei Zustände `WAIT_FOR_1` und `WAIT_FOR_2` ergänzt, in denen das Modul zwischen den Zuständen `COUNTING` und `SEND_COUNT` verharrt, bis beide Antworten von der AI Engine eingetroffen sind (vgl. mit Abschnitt 3.1). Der finale Quelltext des Zeitmessmoduls wird in Quelltext B.1 aufgeführt.

d und D wurden anschließend mit der PL-seitig gemessenen Taktzahl N in einen 56-Bit-String eingebettet und in einem UDP-Paket über den Netzwerk-Stack ausgegeben.

3.6. Computerseitige Datenverarbeitung

Die gemessene Anzahl an latenzbedingten Taktzyklen N (sowie ggf. auch die AIE-Takte d und D) wurden von dem FPGA über den Netzwerk-Stack in Echtzeit in UDP-Paketen an einen Computer entsandt. Um die Ergebnisse in eine verständliche Darstellung zu überführen und zu analysieren, wurden verschiedene computerseitige C++-Programme geschrieben. Zwei davon arbeiten wie folgt:

- `count_clock_cycles.cpp`: Das Programm zählt, wie oft die Taktzyklen N gemessen wurden. Der Zählprozess wird in Echtzeit dargestellt und läuft bis zur manuellen Unterbrechung. Die Ausgabe wird in Abbildung 3.3 dargestellt. Das Ergebnis der Zählung wird zusätzlich im Tabellen-Format in eine lokale Datei geschrieben.

- `storef_data.cpp`: Das Programm speichert die Anzahl der gemessenen Taktzyklen N mit den AI Engine-seitigen Taktzyklen d und D tabellarisch in einer lokalen Datei.

```
Listening for any on port 8020...
```

Clk cycles	Count	Percentage
16	838253794	99.9996 %
17	3235	0.00038592 %

```
Running for 68678 s (1144 m)...
```

Abbildung 3.3.: Eine Bildschirmaufnahme einer Latenzzeitmessung durch `count_clock_cycles.cpp`. Die Tabellenüberschrift gibt die eingestellte IP-Adresse (hier „any“) und den Port wieder, auf dem die Daten erwartet werden. In der ersten Spalte werden die Taktzyklen notiert, die das Dummy-Signal für den Weg zur AI Engine und zurück brauchte. In der mittleren Spalte wird gezählt, wie häufig die zugehörigen Taktzyklen gemessen wurde. Die rechte Spalte gibt den prozentualen Anteil der jeweiligen Taktzyklen von der Anzahl an Messungen insgesamt an. Unten wird in Sekunden und Minuten notiert, wie lange das Programm bereits misst.

4. Abschätzung der Unsicherheiten

Das Zeitmessmodul misst die Anzahl an latenzbedingten Taktzyklen N (siehe Abschnitt 3.1). Diese kann durch die PL-seitige Taktfrequenz f_{PL} in eine Latenzzeit überführt werden:

$$t_N = \frac{N}{f_{\mathrm{PL}}}, \quad \Delta t_N = \sqrt{\left(\frac{\Delta N}{f_{\mathrm{PL}}}\right)^2 + \left(\frac{\Delta f_{\mathrm{PL}} N}{f_{\mathrm{PL}}^2}\right)^2}. \tag{4.1}$$

Die Unsicherheit wurde mittels der Gaußschen Fehlerfortpflanzung bestimmt. Die Umrechnung der AIE-seitigen Takte D in t_D bezüglich der Taktfrequenz f_{AIE} erfolgt analog.

4.1. Taktzyklen

Eine Abschätzung der Unsicherheit der latenzbedingten Taktzyklen N ist nicht ohne Weiteres möglich. Die für diese Arbeit entwickelten Module und Programme wurden durch Simulationen und *Integrated Logical Analyzer* getestet und arbeiteten wie beabsichtigt. Mögliche Variationen von N werden folglich der Hardware zugeschrieben, deren Verhalten nicht vollständig nachvollzogen werden konnte. In Abschnitt 5.1 wurden die Verteilungen von N gemessen. Die Messwerte wichen nur selten (i. d. R. in unter 0,001 % der Fälle) von dem häufigsten N-Wert ab. Die Verteilungen sind stark asymmetrisch und gleichen keiner Normalverteilung. In Abschnitt 5.1.1 wurde eine Periodizität in den Messungen von N festgestellt, sodass die Messergebnisse nicht unabhängig voneinander zu sein scheinen. Aus diesen Gründen und weil die Taktzahlen N digital gezählt und in großer Anzahl betrachtet werden, wird die Messunsicherheit vernachlässigt und $\Delta N = 0$ angenommen.

In Abschnitt 5.1 wird gezählt, wie häufig N Taktzyklen gezählt werden. Die Unsicherheit hierauf wird aus denselben Gründen als 0 approximiert.

4.2. Taktfrequenzen

Die Unsicherheiten der Taktfrequenzen sind ebenfalls nicht direkt ersichtlich. Für den PL-Takt bestimmt AMDs Software *Vivado* eine Taktunsicherheit von 0,039 ns. Diese entspricht einer halben Zitterperiode (*Jitter*), die beschreibt, wie groß das Fenster ist, in dem die Periodendauer einzelner PL-Takte von dem Durchschnitt

abweichen kann. Im Vergleich zu den Taktperioden von 1–5 ns ist die Zitterperiode klein. Unter der Annahme eines normalverteilten Zitterns sinkt der Einfluss dieser Unsicherheit mit der Anzahl der betrachteten Taktzyklen und wird daher nicht beachtet.

Der Referenztakt (33,333 MHz) des AMD Versal VK190-Boards wird durch Silicon Labs' „SI570JAC000900DG“ vorgegeben. AMD notiert eine Taktstabilität von 61,5 ppm (*parts per million*). [37] Das bedeutet, dass der Referenztakt höchstens um $61{,}5 \cdot 10^{-6} \cdot 33{,}333\,\mathrm{MHz} \approx 2{,}05\,\mathrm{kHz}$ von seinem idealen Wert abweicht. Allerdings ist unklar, inwiefern die *Phase-Locked Loops* (PPLs), die daraus die höheren Taktfrequenzen erzeugen, die Taktstabilität beeinflussen. Denkbar ist, dass die Abweichung durch zusätzliche Schaltungen vergrößert wird. Unter der Annahme, dass die PPLs die Unsicherheit im Verhältnis nicht verkleinern, lassen sich hieraus untere Schranken $\Delta f_{-} = 61{,}5 \cdot 10^{-6} \cdot f$ für die Taktunsicherheiten bestimmen, die in Tabelle 4.1 gelistet werden.

Um die Taktunsicherheit von oben abschätzen zu können, wurde das PL-Taktsignal über einen FMC+-Anschluss ausgegeben und am Oszilloskop betrachtet. Eine Ausgabe des AI Engine-Taktes war nicht möglich. Allerdings beruht dieser auf demselben Referenztakt [38]. Daher wurden die Periodendauern τ mit ihren Unsicherheiten $\Delta\tau$ von verschiedenen PL-Taktfrequenzen gemessen (siehe Tabelle A.1, Daten von P. Schwäbig), um durch eine Extrapolation die Unsicherheiten weiterer Taktfrequenzen abzuschätzen. Die gemessenen Periodendauern gleichen für die meisten Frequenzen f ihren theoretischen Werten $1/f$. Die Periodendauern für $f = 100\,\mathrm{MHz}$ und $f = 259\,\mathrm{MHz}$ weichen lediglich um 0,1 % bzw. 0,003 % davon ab. Die Unsicherheiten der Periodendauern können über die Gaußsche Fehlerfortpflanzung in die zugehörigen Frequenzen umgerechnet werden:

$$f = \frac{1}{\tau}, \quad \Delta f = \frac{\Delta\tau}{\tau^2}. \tag{4.2}$$

Die gemessenen Periodendauern werden in Abbildung 4.1 dargestellt. Obwohl kein eindeutiger funktionaler Zusammenhang entnehmbar ist, zeigt sich eine Tendenz, die approximativ durch eine Gerade beschrieben werden kann:

$$\Delta\tau = 0{,}103\,\mathrm{ps} \cdot \frac{\tau}{\mathrm{ns}} + 3{,}216\,\mathrm{ps}. \tag{4.3}$$

Unter der Annahme, dass der Verlauf für größere Frequenzen, d. h. kleinere Periodendauern, extrapoliert werden kann, kann daraus die Unsicherheit des AI Engine-Taktes (0,8 ns) auf 3,3 ps abgeschätzt werden. Die durch Gleichung (4.2) resultierenden Taktunsicherheiten werden in Tabelle 4.1 aufgeführt. In den folgenden Abschnitten werden die oberen Unsicherheiten angenommen, das heißt $f_{\mathrm{PL}} = (200{,}00 \pm 0{,}14)\,\mathrm{MHz}$, $f_{\mathrm{PL}} = (625{,}0 \pm 1{,}4)\,\mathrm{MHz}$ und $f_{\mathrm{AIE}} = (1250{,}0 \pm 5{,}2)\,\mathrm{MHz}$. Insbesondere die Bestimmung der Unsicherheit von f_{AIE} verlief ungenau und unter stark vereinfachten Annahmen. In der Praxis sind die Taktunsicherheiten auf der ACAP vermutlich allesamt wesentlich kleiner.

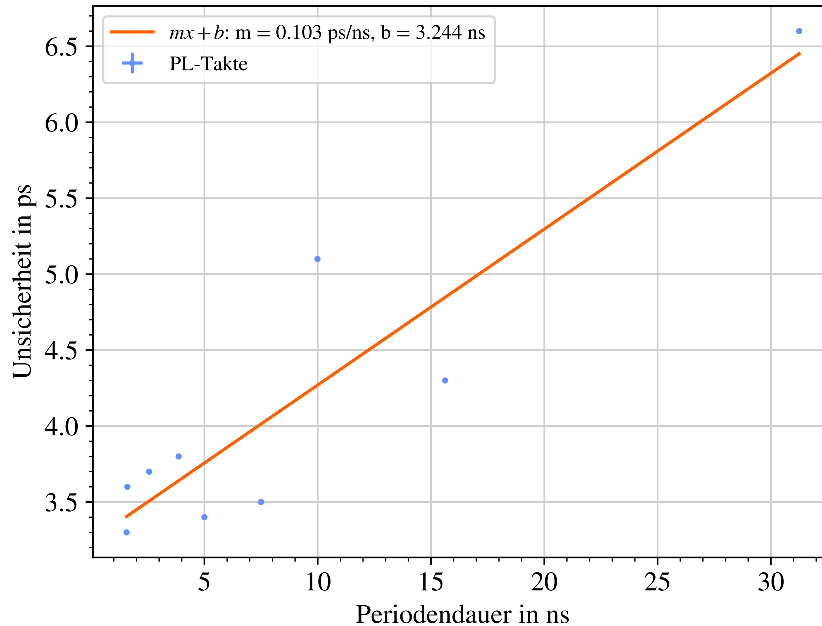

Abbildung 4.1.: Die Unsicherheiten der Periodendauern $\Delta\tau$ von verschiedenen PL-Taktfrequenzen aufgetragen gegen die Periodendauer τ, angepasst durch eine Gerade. Die Daten stammen aus Tabelle A.1, aufgenommen von P. Schwäbig. Die Unsicherheiten der aufgetragenen Unsicherheiten wurden vernachlässigt.

f in MHz	Δf_- in MHz	Δf_+ in MHz	τ in ns	$\Delta\tau$ in ns
200,00	0,02	0,14	5,0000	0,0034
625,00	0,04	1,38	1,6000	0,0036
1250,00	0,08	5,16	0,8000	0,0033

Tabelle 4.1.: Die Unsicherheiten ausgewählter Taktfrequenzen f, aufgeteilt in eine untere Abschätzung Δf_- und eine obere Abschätzung Δf_+. Δf_- wurde mithilfe der Unsicherheit des Referenztaktes berechnet. Δf_+ wurde durch eine externe Messung der Periodendauer τ verschiedener PL-Takte (Tabelle A.1) und eine Umrechnung mit Gleichung (4.2) bestimmt. Die obere Unsicherheit der 1250 MHz folgt aus einer Extrapolation durch eine Anpassungsgerade.

5. Durchführung der Latenzmessungen

Für die Latenzzeitmessungen wurden drei Parameter variiert: die Taktfrequenz f_{PL} der PL des FPGAs (200 MHz oder 625 MHz), die Wortgröße s des AXI4-Streams (32, 64 oder 128 Bit) und die Verzögerung d auf der AI Engine (0–2550). Bei den 625 MHz handelt es sich um die maximale Taktfrequenz, die von AMD für die PL im Zusammenhang mit den AI Engines empfohlen wird (siehe S. 13). Die Taktfrequenz von 200 MHz wurde zu Testzwecken gewählt, weil die Implementierung eines *Integrated Logical Analyzers* (vgl. Abschnitt 3.5) bei 625 MHz zu Zeitfehlern führte.

5.1. Latenzzeit für Passthrough-Verhalten (d = 0)

5.1.1. Taktfrequenz f_{PL} = 200 MHz

Als Erstes wurde die latenzbedingte Taktzahl N für $f_{\mathrm{PL}} = 200\,\mathrm{MHz}$ und $d = 0$ Operationen auf der AI Engine für die verschiedenen Wortgrößen gemessen. Die Messdaten werden in Tabelle 5.1 gelistet. Die Anzahl der Messdaten variierte je nach Messreihe. Abbildung 5.1 demonstriert beispielhaft, dass die prozentualen Anteile der gemessenen Taktzyklen jedoch schnell konvergieren und eine Genauigkeit auf die vierte Dezimalstelle der Prozentsätze gerechtfertigt ist.

Für $s = 32\,\mathrm{Bit}$ entsprach die Latenz in etwa 99,9996 % der Fälle 16 Taktzyklen. Für $s = 64\,\mathrm{Bit}$ waren es zu 99,9994 % ebenfalls 16 Taktzyklen. Für $s = 128\,\mathrm{Bit}$ wurden zu 99,9996 % stattdessen 17 Taktzyklen gemessen.

In den verbleibenden Fällen traf die Antwort der AI Engine jeweils einen Taktzyklus später ein. Dies könnte darauf zurückzuführen sein, dass die AI Engine auf einer anderen Taktfrequenz als die PL arbeitet. Der AMD Versal verfügt über einen nicht einsehbaren Mechanismus, der die Taktfrequenzen aufeinander abstimmt und die Kommunikation stabilisiert (CDC, siehe Abschnitt 2.3.4). Allerdings könnte zum Beispiel das Taktzittern zu einer Variation führen.

Die Variation der Messwerte wurde für $s = 128\,\mathrm{Bit}$ näher untersucht, indem die Zeitdifferenzen zwischen zwei Latenzmessungen an dem Computer gespeichert wurden, bei denen 18 anstelle von 17 Taktzyklen gemessen wurden. Hieraus ergab sich das Histogramm aus Abbildung 5.2. Das Histogramm zeigt, dass die Zeitdifferenzen zwischen den Abweichungen etwa 160 ms, 500 ms oder ungefähr 660 ms betragen.

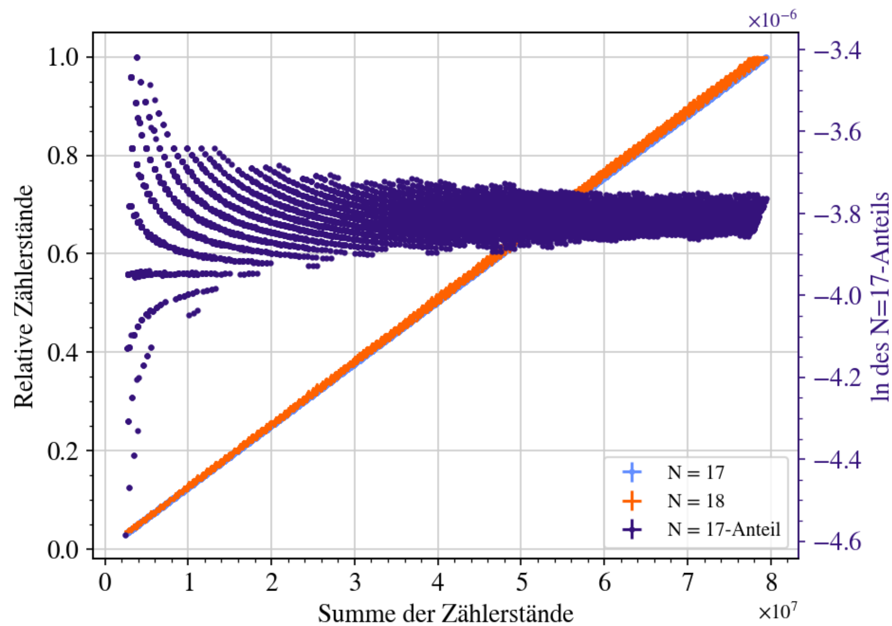

Abbildung 5.1.: Die Entwicklung der Latenztaktzählungen aufgetragen gegen die Gesamtanzahl der Latenztaktmessungen. Die linke Ordinatenachse gibt vor, wie oft die Latenzen $N = 17$ und $N = 18$ jeweils relativ zu ihrer maximalen Anzahl gemessen wurden. Beide Zählungen steigen linear. Die rechte Ordinatenachse beschreibt den $N = 17$-Anteil von der Gesamtanzahl der Latenzmessungen. Der Anteil wurde logarithmiert, damit die Achsenbeschriftung besser lesbar ist. Die Skala reicht also von 99,999 54 % bis 99,999 66 %. Nach etwa 50 Million Datenpunkten liegt der Prozentsatz in dem Bereich zwischen 99,999 61 % und 99,999 63 %. Der Graph wurde beispielhaft für $f_{\mathrm{PL}} = 200\,\mathrm{MHz}$, $s = 128\,\mathrm{Bit}$ und $d = 0$ aufgenommen und enthält 68 Datensätze mit je 300 Datenpunkten von etwa 2 bis 80 Millionen.

Wortgröße	Taktzahl	Latenzzeit	Anzahl	Anteil in %
32 Bit	16	$(80{,}000 \pm 0{,}056)$ ns	854 808 670	99,9996 %
	17	$(85{,}000 \pm 0{,}060)$ ns	3299	0,0004 %
64 Bit	16	$(80{,}000 \pm 0{,}056)$ ns	123 365 846	99,9994 %
	17	$(85{,}000 \pm 0{,}060)$ ns	723	0,0006 %
128 Bit	17	$(85{,}000 \pm 0{,}060)$ ns	113 733 492	99,9996 %
	18	$(90{,}000 \pm 0{,}063)$ ns	435	0,0004 %

Tabelle 5.1.: Die Ergebnisse der Messungen der latenzbedingten Taktzahl N für verschiedene Wortgrößen s bei einer Taktfrequenz von $f_{\mathrm{PL}} = (200{,}00 \pm 0{,}14)$ MHz und ohne künstliche Verzögerung ($d = 0$) auf der AI Engine. Die Unsicherheiten wurden anhand von Abschnitt 4 berechnet. Die prozentualen Anteile wurden auf die vierte Dezimalstelle gerundet.

Eine Autokorrelation auf einen Datensatz mit 14 072 295 Aufnahmen ergab, dass eine Verschiebung der Messdaten um etwa $9 \cdot 10^5$ Datenpunkte (das entspricht rund 50 s) zu einem Pearson-Korrelationskoeffizient von ungefähr 0,7 führte. Die Messdaten scheinen daher einem periodischen Muster zu folgen. Ein ähnliches Verhalten (jedoch mit anderen Zeitdifferenzen) wurde auch für die anderen Wortgrößen und $f_{\mathrm{PL}} = 625$ MHz festgestellt. Die Ursache liegt vermutlich auch hier an AMD Versal-internen CDC-Prozessen, die nicht ohne Weiteres einsehbar sind.

Die Übertragung von 128 Bit beansprucht einen Taktzyklus mehr im Vergleich zu den anderen Wortgrößen. Dies liegt daran, dass die PL und die AI Engines über je zwei ein- und ausgehende 32-Bit-AXI4-Streams (siehe Abschnitt 2.3.4) kommunzieren. Je größer das Datenwort ist, in desto mehr 32-Bit-Pakete muss es aufgeteilt und hinterher wieder zusammengesetzt werden.

Die durch die Prozentsätze gewichteten Mittel der Latenzen werden in Tabelle 5.3 aufgeführt. Zusammenfassend bedeutet die Übertragung von 32-Bit-Signalen gemäß den Messdaten keinen bedeutsamen Vorteil gegenüber der 64-Bit-Übertragung. Der Unterschied in der Latenzzeit ist deutlich kleiner als die zugehörige Unsicherheit. Gleichzeitig wird bei der 64-Bit-Übertragung ein doppelter Durchsatz (transportierte Datenmenge pro Zeit) erreicht:

$$\delta = \frac{s}{\overline{t_N}}, \quad \Delta\delta = \Delta\overline{t_N} \cdot \frac{s}{\overline{t_N}^2}. \tag{5.1}$$

Im Vergleich zu der 64-Bit-Übertragung wird bei 128 Bit durch einen zusätzlichen Taktzyklus die doppelte Datenmenge übertragen und somit der größte Durchsatz ermöglicht.

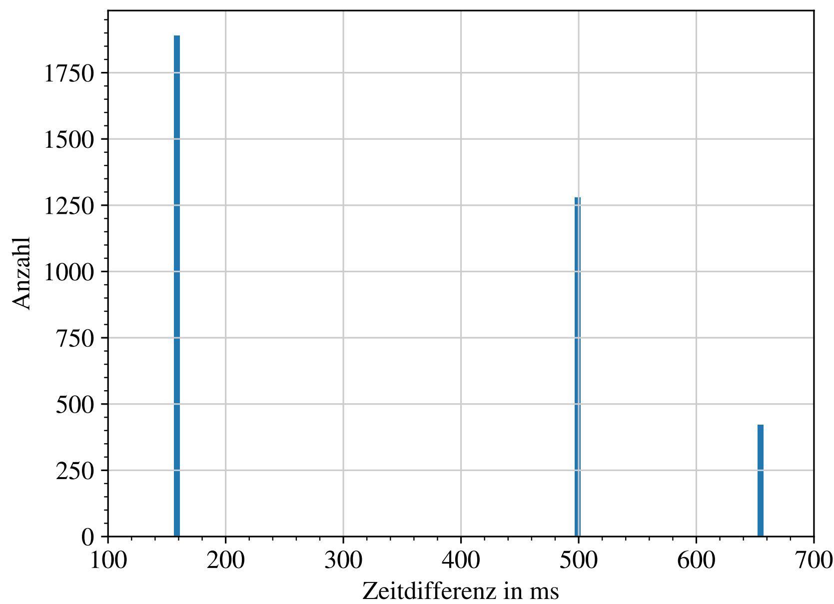

Abbildung 5.2.: Histogramm mit den Zeitdifferenzen zwischen zwei Latenzzeitmessungen ($d = 0$, $s = 128$ Bit, $f_{\mathrm{PL}} = 200$ MHz), bei denen 18 Taktzyklen anstelle der üblichen 17 Taktzyklen gemessen wurden. Insgesamt 3592 Daten auf 100 Bins mit einer Breite von je 6 ms.

5.1.2. Taktfrequenz f_{PL} = 625 MHz

Die Messung der latenzbedingten Taktzahl N bei $d = 0$ wurde anschließend für die Taktfrequenz $f_{\mathrm{PL}} = 625$ MHz wiederholt. Die Messdaten werden in Tabelle 5.1 gelistet. Die Übertragung von 32 Bits wurde nicht länger untersucht, weil sie den geringsten Durchsatz ermöglicht und sich in der Latenz nicht wesentlich von den anderen Wortgrößen unterscheidet.

Während die Taktfrequenz von 200 MHz auf 625 MHz um 312,5 % gestiegen ist, sind die Latenzzeiten im Durchschnitt auf 56,1 % (64 Bit) beziehungsweise 52,7 % (128 Bit) gesunken. Dieses Verhalten deckt sich mit der Dokumentation von AMD: „Generally, the higher the frequency of the PL, the lower the latency in absolute time“ [26]. Durch die erhöhte Taktfrequenz können die Daten vermutlich eher von der PL auf die AI Engine und zurück übertragen werden, ohne dass die schneller getaktete AI Engine auf die langsamer getaktete PL warten muss.

Die Streuung auf mehrere Taktzahlen tritt stärker auf als bei der 200-MHz-Messung. Die Taktzyklen nehmen nun fünf anstelle von zwei Werten an. Die prozentualen Anteile nehmen zudem für höhere Taktzahlen nicht länger monoton ab, sondern wachsen zwischenzeitlich wieder an. Die durch den Prozentsatz gewichteten Mittel der Latenzzeiten werden in Tabelle 5.3 aufgelistet.

Auffällig ist, dass die Übertragung eines 128-Bit-Signals mit $(44{,}800 \pm 0{,}101)$ ns im Durchschnitt eine geringere Latenz aufweist als die Übertragung eines 64-Bit-

Wortgröße	Taktzahl	Latenzzeit	Anzahl	Anteil in %
64 Bit	28	$(44{,}80 \pm 0{,}11)$ ns	1 298 892 885	93,8680 %
	29	$(46{,}40 \pm 0{,}11)$ ns	84 842 064	6,1313 %
	30	$(48{,}00 \pm 0{,}11)$ ns	2688	0,0002 %
	31	$(49{,}60 \pm 0{,}12)$ ns	2697	0,0002 %
	32	$(51{,}20 \pm 0{,}12)$ ns	4450	0,0003 %
128 Bit	28	$(44{,}80 \pm 0{,}11)$ ns	210 103 503	99,9993 %
	29	$(46{,}40 \pm 0{,}11)$ ns	409	0,0002 %
	30	$(48{,}00 \pm 0{,}11)$ ns	406	0,0002 %
	31	$(49{,}60 \pm 0{,}12)$ ns	541	0,0003 %
	32	$(51{,}20 \pm 0{,}12)$ ns	164	0,0001 %

Tabelle 5.2.: Die Ergebnisse der Messungen der latenzbedingten Taktzahl N für verschiedene Wortgrößen s bei einer Taktfrequenz von $f_{\mathrm{PL}} = (625{,}0 \pm 1{,}4)$ MHz und $d = 0$ künstlichen Operationen auf der AI Engine. Die Unsicherheiten wurden anhand von Abschnitt 4 berechnet. Die prozentualen Anteile wurden auf die vierte Dezimalstelle gerundet.

Signals $(44{,}898 \pm 0{,}101)$ ns. Dies widerspricht der Messung aus Abschnitt 5.1.1 und der Erwartung, u. a. weil die 128 Bits in mehr Taktzyklen übertragen werden müssten [39]. In Abbildung 2.5 durchlaufen die 32-Bit-AXI4-Streams einen Stream-FIFO, der über 32-Bit-Streams und 128-Bit-Streams mit der AI Engine verbunden ist. Das bedeutet womöglich, dass die 64 Bits auf der AI Engine und die 128 Bits auf dem Stream-FIFO in 32 Bits zerlegt werden. Diese Aufteilung impliziert, dass die Schnittstelle für die Übertragung von 128 Bits optimiert wurde.

Frequenz	Wortgröße	Taktzahl $\overline{N}$	Latenzzeit $\overline{t_N}$ in ns	Durchsatz δ in Bit/µs
200 MHz	32 Bit	16,000 004	$(80{,}000 \pm 0{,}057)$	$(400{,}00 \pm 0{,}28)$
	64 Bit	16,000 006	$(80{,}000 \pm 0{,}057)$	$(800{,}00 \pm 0{,}56)$
	128 Bit	17,000 004	$(85{,}000 \pm 0{,}060)$	$(1505{,}9 \pm 1{,}1)$
625 MHz	64 Bit	28,061 34	$(44{,}898 \pm 0{,}101)$	$(1425{,}4 \pm 3{,}2)$
	128 Bit	28,000 02	$(44{,}800 \pm 0{,}101)$	$(2857{,}1 \pm 6{,}4)$

Tabelle 5.3.: Die gewichteten arithmetischen Mittel der Latenzen N und t_N für verschiedene Wortgrößen s und Taktfrequenzen f_{PL} und ohne künstliche Verzögerung ($d = 0$). Die Daten stammen aus Tabelle 5.1 und Tabelle 5.2. t_N wurde aus Gleichung (4.1) berechnet, δ aus Gleichung (5.1).

5.2. Latenzzeit bei Operationen auf der AI Engine

Zuletzt wurde untersucht, wie die Latenzen für eine zunehmende Anzahl an Operationen auf der AI Engine skalieren. Wie in Abschnitt 3.4.2 beschrieben, wurde dafür eine Zahl d an die AI Engine übertragen. Die AI Engine wartete ungefähr d AI Engine-seitige Taktzyklen ab und gab d als ideale Verzögerung mit der tatsächlichen Verzögerung D an die PL zurück. Auf der PL wurde wie zuvor die währenddessen verstrichene PL-seitige Taktzahl N gemessen.

Die Messreihen beziehen sich auf die Wortgrößen $s = 64$ Bit und $s = 128$ Bit bei $f_{\text{PL}} = 625$ MHz in den Bereichen $d \in [0, 255]$ (in Einerschritten) und $d \in [0, 2550]$ (in Zehnerschritten). Die gezählten Taktzahlen wurden über die Gleichung (4.1) in die Latenzzeit umgerechnet, um besser verglichen werden zu können. In Abbildung 5.3 werden die gemessenen Latenzzeiten t_N und t_D gegen d aufgetragen.

Die Abbildungen bis $d = 255$ zeigen einen Stufenverlauf, der dem Overhead der `while`-Schleife geschuldet ist (siehe Abschnitt 3.4.2). Die Stufen des AIE-Verlaufs haben eine Höhe von 17 Taktzyklen (13,6 ns). Der Assemblercode der AI Engine (Quelltext B.2) zeigt, dass die `while`-Schleife lediglich 12 Takte einnimmt und die verbleibenden fünf Takte nicht planmäßig sind. In den Zeilen 11 und 12 schreiben `ST.SPIL` und `LDA.SPIL` den Inhalt eines Registers in den Speicher und anschließend wieder in ein Register. Dieses Verhalten könnte auf das `volatile`-Keyword zurückzuführen sein. Zugriffe auf den Speicher dauern üblicherweise länger als Zugriffe auf die Register. Womöglich wurde dadurch der Programmfluss hinausgezögert, ohne dass der unabhängige [34] Taktzähler D durch `get_cycles` betroffen ist. Für $d = 0$ wurden allerdings 24 Takte gezählt. Dies stimmt mit dem Assemblercode überein, obwohl hier die `while`-Schleife bereits einmal durchlaufen wurde. (Die $d = 0$-Messung in diesem Abschnitt ist nicht direkt mit der $d = 0$-Messung in den vorherigen Abschnitten vergleichbar, weil das hiesige Programm einen größeren Overhead hat.)

Für beide Wortgrößen steigt die gesamte Latenzzeit t_N mit der Anzahl an Operationen D linear an. Die Differenz zwischen den auf der PL gemessenen und den auf der AI Engine gemessenen Latenzzeiten liegt näherungsweise konstant bei 50 ns (64 Bit) bzw. 51 ns (128 Bit). Hierbei handelt es sich um die Zeit, die das Signal für den Weg zur AI Engine und zurück benötigt. Diese Zeiten liegen etwas über den Latenzzeiten von etwa 45 ns, die im Abschnitt 5.1 für das Passthrough-Verhalten gemessen wurden. Das liegt daran, dass sie in diesem Abschnitt auf der AI Engine einen zusätzlichen Overhead enthalten (siehe Quelltext 3.2, die Instruktionen vor und hinter dem Aufruf von `get_cycles`).

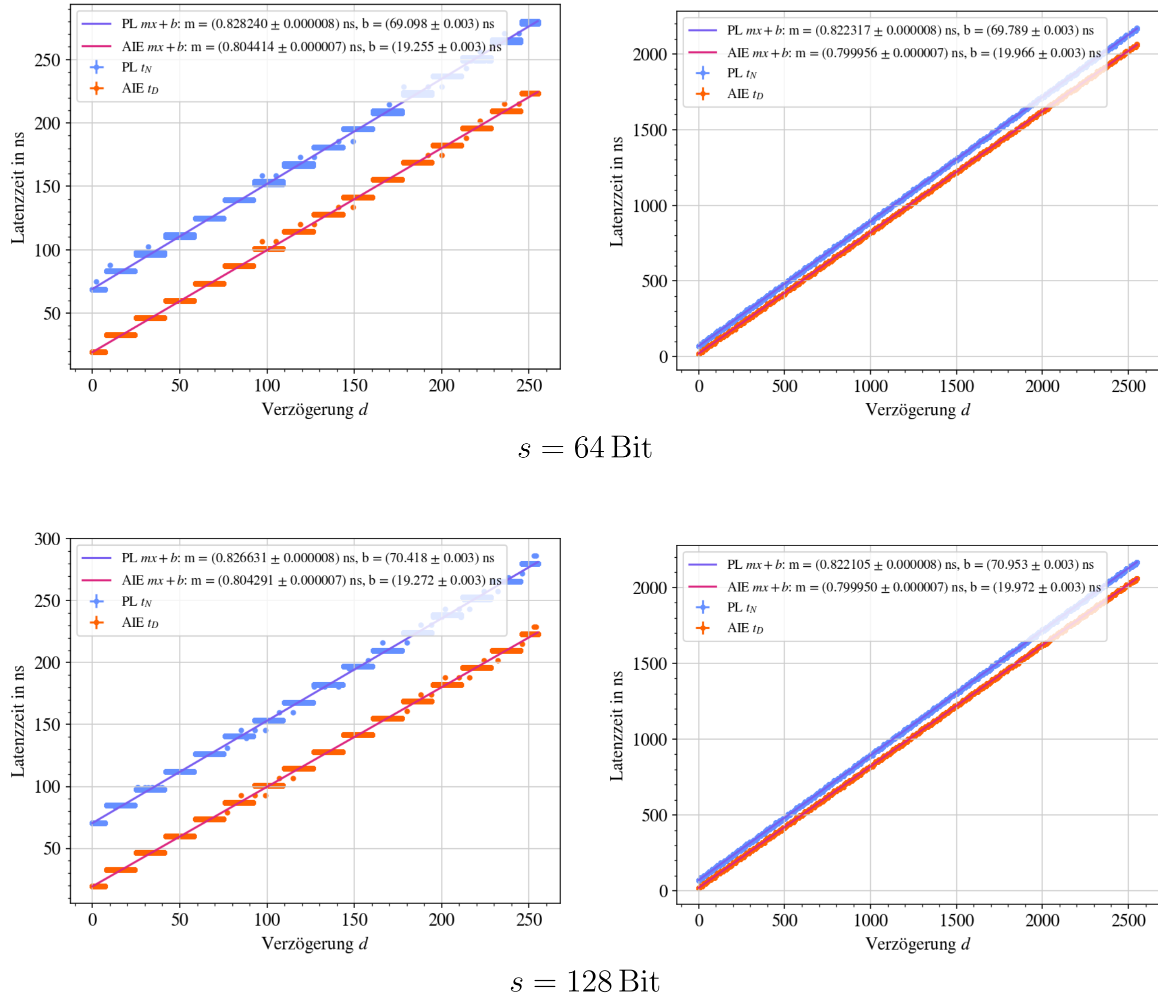

Abbildung 5.3.: Die PL- und AI Engine-seitigen Latenzmessungen für verschiedene Anzahlen d an AI Engine-Operationen und verschiedene Wortgrößen s. Jeweils 50 000 Messpunkte pro Graph gleichmäßig über d verteilt. Aufgenommen bei $f_{\mathrm{PL}} = (625{,}0 \pm 1{,}4)$ MHz und $f_{\mathrm{AIE}} = (1250{,}0 \pm 5{,}2)$ MHz in den Bereichen $d \in [0, 255]$ (in Einerschritten, links) und $d \in [0, 2550]$ (in Zehnerschritten, rechts). Die Unsicherheiten wurden gemäß Abschnitt 4 berechnet und sind aufgrund ihrer Größe nicht erkennbar. Die Messdaten wurden durch Geraden über die Methode der kleinsten Quadrate angepasst [40].

Wie anhand der Anpassungsgeraden und für den größeren Messbereich bis $d = 2550$ ersichtlich ist, scheinen die Latenzzeiten zu divergieren. Die Steigungen der PL-Latenzen weichen für die Graphen bis $d = 255$ um etwa 3,8 % bzw. für die Graphen bis $d = 2550$ um 2,5 % von den Steigungen der AIE-Latenzen ab. Dies könnte an einem systematischen Fehler bei den Taktfrequenzen liegen. In Abschnitt 4.2 wurde die Periodendauer von $f_{\mathrm{PL}} = 625\,\mathrm{MHz}$ auf $(1{,}6000 \pm 0{,}0036)\,\mathrm{ns}$ sehr nahe der erwarteten 1,6 ns gemessen. Folglich müsste der AIE-Takt für die Abweichung verantwortlich sein und um etwa 31 MHz bis 48 MHz von den angenommenen 1250 MHz abweichen. Unter [38] kann die Verteilung der Takte auf der AMD Versal ACAP ausgehend von dem allgemeinen Referenztakt „REF_CLK“ nachvollzogen werden. Über die AMD-Software *Vivado*[1] wurde festgestellt, dass der daraus bezogene Referenztakt für die AI Engines „HSM0_REF_CLK“ lediglich 32,4324 MHz, also etwa 97,3 % der geforderten 33,333 MHz vorgibt. Wenn dieser Prozentsatz auf die 1250 MHz übertragen wird, bedeutet das, dass die Taktfrequenz der AI Engine tatsächlich bei $f_{\mathrm{AIE}} = 1216{,}23\,\mathrm{MHz}$ lag. Dies ist einer falschen Konfiguration der Takte geschuldet, die auf einen Fehler in der AMD-Software zurückgeführt werden konnte.[2] Prinzipiell sollte eine Frequenz von 1250 MHz durch die geeignete Konfiguration aber möglich sein, zumal diese einen tiefen Eingriff in die Konfiguration der ACAP erfordert. Die korrigierten Graphen werden in Abbildung 5.4 abgebildet. Die Geraden weisen hier wesentlich ähnlichere Steigungen auf und weichen nur noch in der ps-Größenordnung voneinander ab. Diese Abweichung könnte an der ungleichen Stufenbreite am unteren und am oberen Ende der d-Skala liegen.

Im kleineren Messbereich bis $d = 255$ sind einzelne Abweichungen von den Stufen erkennbar, die größtenteils gleichermaßen auf der AI Engine und der PL und teilweise nur auf der PL gemessen werden. Die rein PL-seitigen Abweichungen sind bereits aus Abschnitt 5.1 bekannt. Die AI Engine-seitigen Abweichungen betragen in jedem gemessenen Fall 7 Zyklen nach oben oder 10 Zyklen nach unten. Eine Ursache hierfür konnte nicht gefunden werden. Die Abweichungen beeinflussen ebenfalls die Steigungen der Anpassungsgeraden im ps-Bereich, wie nach einer händischen Korrektur festgestellt wurde.

[1] AMD Vivado Design Suite 2025.1

[2] Der Referenztakt der AI Engines wird aus dem allgemeinen Referenztakt des AMD Versal bezogen. Die Taktfrequenz müsste den Angaben nach unverändert bleiben, jedoch sind mehrere PLLs (*Phase-Locked Loops*) zwischen den Takten geschaltet, die die Frequenz um den Faktor $72/2/37 \approx 0{,}97$ abwandeln. Würde der letzte Frequenzteiler die Frequenz durch 36 anstelle der 37 teilen, würde die Frequenz der Erwartung entsprechend unverändert bleiben.

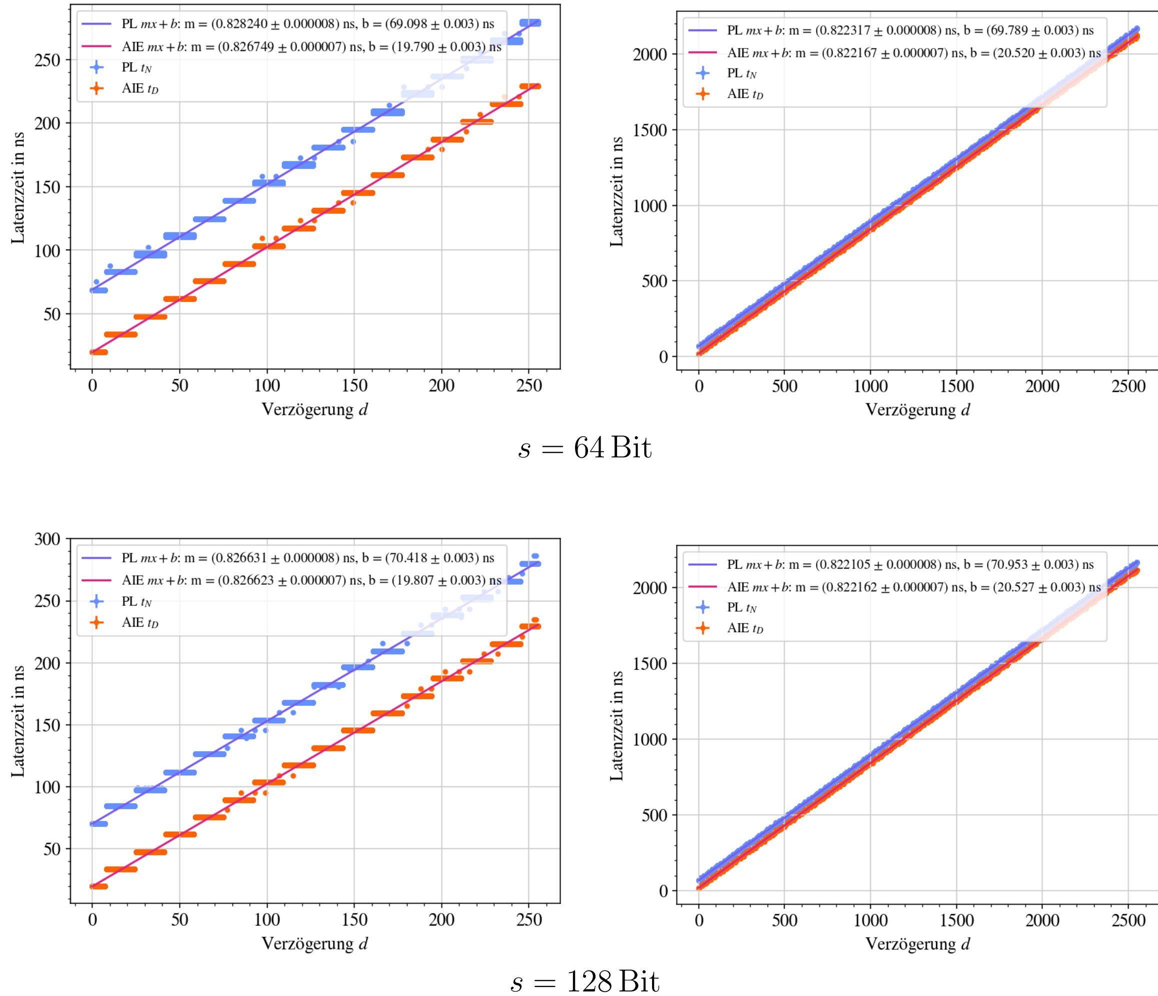

Abbildung 5.4.: Die korrigierten PL- und AI Engine-seitigen Latenzmessungen für verschiedene Anzahlen d an AI Engine-Operationen und verschiedene Wortgrößen s. Jeweils 50 000 Messpunkte pro Graph gleichmäßig über d verteilt. Aufgenommen bei $f_{\text{PL}} = (625{,}0 \pm 1{,}4)$ MHz und $f_{\text{AIE}} = (1216{,}23 \pm 4{,}89)$ MHz in den Bereichen $d \in [0, 255]$ (in Einerschritten, links) und $d \in [0, 2550]$ (in Zehnerschritten, rechts). Die Unsicherheiten wurden gemäß Abschnitt 4 berechnet und sind aufgrund ihrer Größe nicht erkennbar. Die Messdaten wurden durch Geraden über die Methode der kleinsten Quadrate angepasst [40].

5.3. Bedeutung für Lohengrin

Das Lohengrin-Experiment strebt an, die AI Engines einzusetzen, um innerhalb von 100 ns bis 200 ns eine Trigger-Entscheidung zu treffen (siehe Abschnitt 2.2). Aus der Tabelle 5.3 geht hervor, dass der höchste Durchsatz für die Kommunikation mit der AI Engine über einen AXI4-Stream für $f_{\mathrm{PL}} = 625\,\mathrm{MHz}$ und $s = 128\,\mathrm{Bit}$ mit $\delta = (2857{,}1 \pm 6{,}4)\,\mathrm{Bit/\mu s}$ erzielt wird. In diesem Fall beträgt die Latenzzeit ungefähr $(44{,}8 \pm 0{,}1)\,\mathrm{ns}$. Diese beinhaltet lediglich die Zeit, die das zu übertragende Signal für den Weg zu einer AI Engine und zurück und die Schaltung der AXI4-Schnittstelle zur Übertragung von einem 128-Bit-Datensatz benötigt. Das bedeutet, dass die verbleibende Logik über $(55{,}2 \pm 0{,}1)\,\mathrm{ns}$ bis $(155{,}2 \pm 0{,}1)\,\mathrm{ns}$ verfügt, um potenziell größere Datenmengen für den AXI4-Stream aufzuspalten und das neuronale Netz zu betreiben. Unter der Voraussetzung, dass die AIE-Taktfrequenz erfolgreich auf 1250 MHz erhöht wird, entsprechen diese Latenzzeiten 34 bis 97 Taktzyklen auf der PL beziehungsweise 69 bis 194 Taktzyklen auf der AI Engine.

Es ist möglich, die am Trigger eintreffenden Ereignisse auf verschiedene AI Engines aufzuteilen und so die verfügbare Zeit zu erhöhen. Die AMD Versal ACAP verfügt über 400 AI Engines, die jedoch nicht alle gleich an das AXI4-Stream-System angebunden sind. Auf wie viele AI Engines die Daten verteilt werden können, hängt von der Implementierung des neuronalen Netzes ab. Außerdem ist zu beachten, dass die maximal zur Verfügung stehende Zeit durch andere Komponenten des Lohengrin-Aufbaus, wie beispielsweise der Buffergröße des Kalorimeters, eingeschränkt wird.

6. Fazit

Im Rahmen dieser Arbeit wurde ein FPGA-Modul zur Zeitmessung zwischen zwei digitalen Signalen entwickelt. Mit dem Modul wurden die Latenzzeiten gemessen, die entstehen, wenn die PL die von einem Datengenerator unterschiedlich groß erzeugten Datenmengen mit einer einzelnen AI Engine austauscht. Die resultierenden Messdaten wurden an einem Computer ausgelesen und in Laufzeit durch mehrere Programme ausgewertet. Die dafür entwickelten Module und Programme wurden durch Simulationen getestet.

Die Latenzen, die durch die Übertragung verschieden großer Datenwörter an eine AI Engine und zurück über einen AXI4-Stream entstehen, werden in Tabelle 5.3 aufgeführt. Hieraus wird ersichtlich, dass die auf der PL höchstmögliche Taktfrequenz $f_{\mathrm{PL}} = 625\,\mathrm{MHz}$ die geringste Latenz von etwa $(44{,}898 \pm 0{,}101)\,\mathrm{ns}$ für 64 Bit bzw. $(44{,}800 \pm 0{,}101)\,\mathrm{ns}$ für 128 Bit ermöglicht. Der höchste Durchsatz wurde mit $\delta = (2857{,}1 \pm 6{,}4)\,\mathrm{Bit/\mu s}$ für die Übertragung von 128 Bits erreicht.

Für die zweite Triggerstufe des LOHENGRIN-Experiments bedeutet diese Konfiguration, dass etwa $(55{,}2 \pm 0{,}1)\,\mathrm{ns}$ bis $(155{,}2 \pm 0{,}1)\,\mathrm{ns}$ für die Triggerentscheidung durch das neuronale Netz verbleiben, falls lediglich eine AI Engine auf der AMD Versal ACAP eingesetzt werden würde. Indem das neuronale Netz auf mehrere AI Engines ausgebreitet oder das gleiche neuronale Netz mehrfach parallel implementiert wird, lässt sich die effektive Latenz reduzieren. Allerdings ist zu beachten, dass die zur Verfügung stehende Zeit auch durch andere Komponenten des LOHENGRIN-Aufbaus eingeschränkt wird.

Hierfür wäre es interessant, die Latenzen unter Verwendung mehrerer AI Engines zu messen, die ggf. über den „Cascade"-Stream oder die geteilten Speicher Daten austauschen (siehe Abschnitt 2.3.4). Für genauere Messungen wäre zudem eine präzisere Abschätzung der Unsicherheiten von den Taktfrequenzen und den Taktzyklen sinnvoll.

Während der Auswertung der Messdaten fiel auf, dass die Taktfrequenz der AI Engines standardmäßig falsch konfiguriert war und 1216,23 MHz betrug, anstelle der vorgegebenen 1250 MHz. Es ist davon auszugehen, dass die Verwendung von 1250 MHz eine geringere Latenzzeit ermöglicht. Eine entsprechende Messung wäre womöglich hilfreich. Dafür müsste tiefgreifend in die Konfiguration der AI Engines eingegriffen werden.

A. Messdaten zur Abschätzung der Taktunsicherheiten

Frequenz	Periodendauer	Std.-Abweichung	Min.	Max.	Datenmenge
32 MHz	31,2500 ns	6,6 ps	31,23 ns	31,27 ns	5148
64 MHz	15,6250 ns	4,3 ps	15,61 ns	15,64 ns	6121
100 MHz	9,9898 ns	5,1 ps	9,97 ns	10,01 ns	5170
133,33 MHz	7,5001 ns	3,5 ps	7,49 ns	7,51 ns	5080
200 MHz	5,0000 ns	3,4 ps	4,99 ns	5,01 ns	5067
259 MHz	3,8611 ns	3,8 ps	3,85 ns	3,87 ns	5082
390,62 MHz	2,5600 ns	3,7 ps	2,55 ns	2,57 ns	5318
625 MHz	1,6000 ns	3,6 ps	1,59 ns	1,61 ns	5096
640 MHz	1,5625 ns	3,3 ps	1,49 ns	1,57 ns	5123

Tabelle A.1.: Die Messgrößen (gerundet) für verschiedene von der PL abgegriffene Taktfrequenzen, die an einem Oszilloskop (Tektronix 5 Series MSO) mit einem differentiellen Tastkopf (Tektronix TDP3500 Differential Probe) über einen FMC+-Anschluss aufgenommen wurden. Die erste Spalte beinhaltet die zugehörigen Frequenzen, die auf der PL konfiguriert wurden. Die Messdaten stammen von P. Schwäbig.

B. Quelltext

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity t_diff_measurement is
6    port(
7      itm_clk              : in  std_logic;
8      resetn               : in  std_logic;
9      i_inp_start          : in  std_logic;
10     i_inp_stop1          : in  std_logic;
11     i_inp_stop1_data     : in  std_logic_vector(15 downto 0);
12     i_inp_stop2          : in  std_logic;
13     i_inp_stop2_data     : in  std_logic_vector(7 downto 0);
14     o_data_output_port   : out std_logic_vector(55 downto 0);
15     o_output_port_ready  : out std_logic
16   );
17 end entity t_diff_measurement;
18
19 architecture timediff_behaviour of t_diff_measurement is
20   -- Merkt sich die vorherigen Zustände der Start- und
        Stoppsignale
21   signal r_inp_start_last_cycle : std_logic := '0';
22   signal r_inp_stop1_last_cycle : std_logic := '0';
23   signal r_inp_stop2_last_cycle : std_logic := '0';
24
25   -- State machine
26   type measurement_states is (IDLE, COUNTING, WAIT_FOR_1,
        WAIT_FOR_2, SEND_COUNT);
27   signal measurement_state : measurement_states := IDLE;
28
29   -- Kopien der ausgehenden Signale für ILA
30   signal r_data_output_port : std_logic_vector(55 downto 0);
31   signal r_inp_stop1_data : std_logic_vector(15 downto 0);
32   signal r_inp_stop2_data : std_logic_vector(7 downto 0);
33   signal r_output_port_ready : std_logic;
34
```

```
35     -- Taktzähler
36     signal cycles_since_last_event : unsigned(31 downto 0) := (
           others => '0');
37   begin
38
39     o_data_output_port <= r_data_output_port;
40     o_output_port_ready <= r_output_port_ready;
41
42     process (itm_clk) is
43     begin
44       if rising_edge(itm_clk) then
45         if resetn = '0' then
46           cycles_since_last_event <= (others => '0');
47           r_output_port_ready <= '0';
48           r_data_output_port <= (others => '0');
49           r_inp_stop1_data <= (others => '0');
50           r_inp_stop2_data <= (others => '0');
51           r_inp_start_last_cycle <= '0';
52           r_inp_stop1_last_cycle <= '0';
53           r_inp_stop2_last_cycle <= '0';
54           measurement_state <= IDLE;
55         else
56         -- Falls idle warte auf Startsignal, falls counting
              warte auf Stoppsignal, dann sende Zyklen and setze Zä
              hler zurück
57           case measurement_state is
58             when IDLE =>
59               cycles_since_last_event <= (others => '0');
60               r_output_port_ready <= '0';
61               r_data_output_port <= (others => '0');
62               r_inp_stop1_data <= (others => '0');
63               r_inp_stop2_data <= (others => '0');
64               if i_inp_start = '1' and r_inp_start_last_cycle =
                    '0' then
65                 measurement_state <= COUNTING;
66               else
67                 measurement_state <= IDLE;
68               end if;
69
70             when COUNTING =>
71               cycles_since_last_event <= cycles_since_last_event
                     + 1;
72               r_output_port_ready <= '0';
73               r_data_output_port <= (others => '0');
```

```
74            -- Stoppsignal 1 registriert
75            if i_inp_stop1 = '1' and r_inp_stop1_last_cycle =
                 '0' then
76              if i_inp_stop2 = '1' then
77                r_inp_stop1_data <= i_inp_stop1_data;
78                r_inp_stop2_data <= i_inp_stop2_data;
79                measurement_state <= SEND_COUNT;
80              else
81                r_inp_stop1_data <= i_inp_stop1_data;
82                r_inp_stop2_data <= (others => '0');
83                measurement_state <= WAIT_FOR_2;
84              end if;
85            -- Stoppsignal 2 (und nicht Stoppsignal 1)
                 registriert
86            elsif i_inp_stop2 = '1' and r_inp_stop2_last_cycle
                  = '0' then
87              r_inp_stop1_data <= (others => '0');
88              r_inp_stop2_data <= i_inp_stop2_data;
89              measurement_state <= WAIT_FOR_1;
90            -- Nichts registriert => weiterzählen
91            else
92              r_inp_stop1_data <= (others => '0');
93              r_inp_stop2_data <= (others => '0');
94              measurement_state <= COUNTING;
95            end if;
96
97          when WAIT_FOR_1 =>
98            cycles_since_last_event <= cycles_since_last_event
                 ; -- Useless
99            r_output_port_ready <= '0';
100           r_data_output_port <= (others => '0');
101           r_inp_stop2_data <= r_inp_stop2_data; -- Useless
102           if i_inp_stop1 = '1' then
103             r_inp_stop1_data <= i_inp_stop1_data;
104             measurement_state <= SEND_COUNT;
105           else
106             r_inp_stop1_data <= (others => '0');
107             measurement_state <= WAIT_FOR_1;
108           end if;
109
110         when WAIT_FOR_2 =>
111           cycles_since_last_event <= cycles_since_last_event
                 ; -- Useless
112           r_output_port_ready <= '0';
```

```
113             r_data_output_port <= (others => '0');
114             r_inp_stop1_data <= r_inp_stop1_data; -- Useless
115             if i_inp_stop2 = '1' then
116               r_inp_stop2_data <= i_inp_stop2_data;
117               measurement_state <= SEND_COUNT;
118             else
119               r_inp_stop2_data <= (others => '0');
120               measurement_state <= WAIT_FOR_2;
121             end if;
122 
123           when SEND_COUNT =>
124             r_data_output_port(55 downto 40) <=
                   r_inp_stop1_data;
125             r_data_output_port(39 downto 8) <=
                   std_logic_vector(cycles_since_last_event);
126             r_data_output_port(7 downto 0) <= r_inp_stop2_data
                   ;
127             r_inp_stop1_data <= (others => '0');
128             r_inp_stop2_data <= (others => '0');
129             r_output_port_ready <= '1';
130             cycles_since_last_event <= (others => '0');
131             measurement_state <= IDLE;
132 
133           when others =>
134             measurement_state <= IDLE;
135             cycles_since_last_event <= (others => '0');
136             r_output_port_ready <= '0';
137             r_data_output_port <= (others => '0');
138             r_inp_stop1_data <= (others => '0');
139             r_inp_stop2_data <= (others => '0');
140         end case;
141 
142       r_inp_start_last_cycle <= i_inp_start;
143       r_inp_stop1_last_cycle <= i_inp_stop1;
144       r_inp_stop2_last_cycle <= i_inp_stop2;
145      end if;
146     end if;
147   end process;
148 end architecture timediff_behaviour;
```

Quelltext B.1: Das finale VHDL-Zeitmessmodul (vereinfacht) gemäß Abschnitt 3.5 (siehe Gitlab).

```
1      658   MOV c0, cntr
2      660   NOP;                 MOV.s12 r12, #10;       MOV r8,
          cl0
3      668   MUL r12, r12, r9;  ST.SPIL r8, [sp, #-4];  NOP
4      676   NOP
5      678   NOP
6      680   NOP;                 ADD r12, r8, r12
7      684   NOP;                 NOP
8  .label
      TGT_F_Z9inoutpingP12input_streamIyEP13output_streamIyES4__80

9  .loop_nesting 1
10     688   MOV c1, cntr
11     690   NOP;                 ST.SPIL cl1, [sp, #-4]; NOP
12     698   NOP;                 LDA.SPIL r0, [sp, #-4]; NOP
13     706   NOP
14     708   NOP
15     710   NOP
16     712   NOP
17     714   NOP
18     716   NOP
19     718   NOP
20     720   NOP;                 LTU r1, r0, r12
21     724   NOP;                 BNEZ r1,#688;           NOP
22 .delay_slot
23 .swstall delay_slot
24     732   NOP
25 .delay_slot
26 .swstall delay_slot
27     734   NOP
28 .delay_slot
29 .swstall delay_slot
30     736   NOP
31 .delay_slot
32 .swstall delay_slot
33     738   NOP
34 .delay_slot
35 .swstall delay_slot
36     740   NOP
37 .loop_nesting 0
38     742   MOV c0, cntr
39     744   NOP;                 LDB r5, [p1]
40     748   LDA r6, [p2];       NOP;                    NOP
```

Quelltext B.2: Ausschnitt des Assemblercodes mit dem Programm, das eine künstliche Verzögerung auf der AI Engine erzeugt ($s = 64$ Bit, Quelltext 3.2). Die `while`-Schleife befindet sich unter `.loop_nesting 1`. Bei `cntr` handelt es sich um den freilaufenden Taktzähler, der über die C-Funktion `get_cycles` zurückgegeben wird. `NOP` bedeutet „No Operation" und zögert Instruktionen hinaus, falls eine vorherige Instruktion mehrere Takte benötigt. Die Register werden unter [41] erläutert, die Instruktionen teilweise unter [42]. Der Assemblercode wurde für eine verbesserte Lesbarkeit leicht umformatiert, sodass die Spalte der Instruktionen teilweise abgeändert wurde.

Abbildungsverzeichnis

2.1. Schematische Skizze des LOHENGRIN-Aufbaus. 8
2.2. Visualisierungen einer allgemeinen Lookup-Table und der Realisierung eines logischen Oders. 10
2.3. Das Schaltbild eines D-Flipflops und Veranschaulichung der zugehörigen Signale. 11
2.4. Die Architektur einer AI Engine. 12
2.5. Die Schnittstellen einer AI Engine. 13
2.6. Konzeptionelle Darstellung der AXI4-Stream-Schnittstelle zwischen der AIE und der PL. 14
2.7. Foto des AMD-VCK190-Boards. 15
2.8. Schematische Skizze des Versal-Aufbaus. 16

3.1. Veranschaulichung der Zeitmessung zwischen zwei Signalen. 17
3.2. Ein Foto der relevanten Signale während einer Zeitmessung. 19
3.3. Eine Bildschirmaufnahme des Computer-seitigen Auszählprogramms `count_clock_cycles.cpp`. 23

4.1. Die Unsicherheiten der Periodendauern von verschiedenen PL-Taktfrequenzen aufgetragen gegen die zugehörigen Periodendauern. 26

5.1. Die Konvergenz des prozentualen Anteils der Zählungen von $N = 17$. 28
5.2. Zeitdifferenzen zwischen zwei Latenzmessungen, bei denen $N = 18$ anstelle von $N = 17$ gemessen worden sind. 30
5.3. Die PL- und AIE-seitigen Latenzmessungen für verschiedene Anzahlen an AIE-Operationen, unter der Annahme, dass $f_{\text{AIE}} = 1250\,\text{MHz}$ gilt. 33
5.4. Die PL- und AIE-seitigen Latenzmessungen für verschiedene Anzahlen an AIE-Operationen, nach der Korrektur der AIE-Frequenz auf $f_{\text{AIE}} = 1216{,}23\,\text{MHz}$. 35

Tabellenverzeichnis

4.1. Die oberen und unteren Abschätzungen der Unsicherheiten von ausgewählten Taktfrequenzen. 26

5.1. Die Ergebnisse der Latenzmessungen für $f_{\text{PL}} = 200\,\text{MHz}$, $d = 0$. 29
5.2. Die Ergebnisse der Latenzmessungen für $f_{\text{PL}} = 625\,\text{MHz}$, $d = 0$. 31
5.3. Die zusammenfassend gemittelten Ergebnisse der Latenzmessungen $d = 0$ für $f_{\text{PL}} = 200\,\text{MHz}$ und $f_{\text{PL}} = 625\,\text{MHz}$. 31

A.1. Die Messungen verschiedener von der PL abgegriffene Taktfrequenzen zur Abschätzung der Unsicherheiten. 38

Quelltextverzeichnis

3.1. AI Engine-seitiges C++-Programm (vereinfacht), das 64-Bit-Eingangsdaten über `readincr` einliest und über `writeincr` wieder zurückschickt (siehe Gitlab). Das `true` im `writeincr`-Aufruf setzt das `TLAST`-Signal im AXI4-Stream und gibt vor, dass das zurückgesendete Paket vollständig ist. 20

3.2. AI Engine-seitiges C++-Programm (vereinfacht), das eine 64-Bit-Zahl d über `readincr` einliest, d-Zyklen abwartet und über zwei parallele Streams d und die tatsächlich abgewarteten Zyklen wieder zurückschickt (siehe Gitlab). 21

B.1. Das finale VHDL-Zeitmessmodul (vereinfacht) gemäß Abschnitt 3.5 (siehe Gitlab). 39

B.2. Ausschnitt des Assemblercodes mit dem Programm, das eine künstliche Verzögerung auf der AI Engine erzeugt ($s = 64$ Bit, Quelltext 3.2). Die `while`-Schleife befindet sich unter `.loop_nesting 1`. Bei `cntr` handelt es sich um den freilaufenden Taktzähler, der über die C-Funktion `get_cycles` zurückgegeben wird. `NOP` bedeutet „No Operation“ und zögert Instruktionen hinaus, falls eine vorherige Instruktion mehrere Takte benötigt. Die Register werden unter [41] erläutert, die Instruktionen teilweise unter [42]. Der Assemblercode wurde für eine verbesserte Lesbarkeit leicht umformatiert, sodass die Spalte der Instruktionen teilweise abgeändert wurde. 43

Literaturverzeichnis

[1] F. Zwicky, „Die Rotverschiebung von extragalaktischen Nebeln,“ *Helvetica Physica Acta*, Jg. 6, S. 110–127, 1933.

[2] M. Cirelli, A. Strumia und J. Zupan, „Dark matter,“ *SciPost Phys. Rev.*, S. 1, 2026. DOI: 10.21468/SciPostPhysRev.1. Adresse: https://scipost.org/10.21468/SciPostPhysRev.1.

[3] P. Bechtle u. a., *A Proposal for the Lohengrin Experiment to Search for Dark Sector Particles at the ELSA Accelerator*, 2025. arXiv: 2410.10956 [hep-ex]. Adresse: https://arxiv.org/abs/2410.10956.

[4] ErUM-Data DEEP. „ErUM-Data DEEP, About. “Adresse: https://www.erumdata-deep.de/about.html. Abruf: 20.03.2026.

[5] L. Roszkowski, E. M. Sessolo und S. Trojanowski, „WIMP dark matter candidates and searches – current status and future prospects,“ *Reports on Progress in Physics*, Jg. 81, Nr. 6, S. 066 201, 2018, ISSN: 1361-6633. DOI: 10.1088/1361-6633/aab913. Adresse: https://arxiv.org/abs/1707.06277.

[6] K. van Bibber, P. M. McIntyre, D. E. Morris und G. G. Raffelt, „Design for a practical laboratory detector for solar axions,“ *Phys. Rev. D*, Jg. 39, S. 2089–2099, 8 Apr. 1989. DOI: 10.1103/PhysRevD.39.2089. Adresse: https://link.aps.org/doi/10.1103/PhysRevD.39.2089.

[7] E. Izaguirre, G. Krnjaic, P. Schuster und N. Toro, „Testing GeV-scale dark matter with fixed-target missing momentum experiments,“ *Physical Review D*, Jg. 91, Nr. 9, Mai 2015, ISSN: 1550-2368. DOI: 10.1103/physrevd.91.094026. Adresse: http://dx.doi.org/10.1103/PhysRevD.91.094026.

[8] R. Merrick, *Getting Started with FPGAs, Digital Circuit Design, Verilog, and VHDL for Beginners*. 2024, ISBN: 978-1-7185-0294-9.

[9] Z. Guo, W. Najjar, F. Vahid und K. Vissers, „A Quantitative Analysis of the Speedup Factors of FPGAs over Processors,“ in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Ser. FPGA ’04, Monterey, California, USA: Association for Computing Machinery, 2004, S. 162–170, ISBN: 1581138296. DOI: 10.1145/968280.968304. Adresse: https://doi.org/10.1145/968280.968304.

[10] Advanced Micro Devices, Inc., *Versal Adaptive SoC Configurable Logic Block Architecture Manual (AM005), Overview*, Version 1.4 English, Mai 2025. Adresse: https://docs.amd.com/r/en-US/am005-versal-clb/Overview?tocId=QPnYK6uQEmnsc4awt3YeRw, Abruf: 20.03.2026.

[11] M. Balch, *Complete Digital Design, A Comprehensive Guide to Digital Electronics and Computer System Architecture*. The McGraw-Hill Companies, Inc., Juni 2003.

[12] M. E. Angoletta, „Digital signal processor fundamentals and system design," 2008. DOI: 10.5170/CERN-2008-003.167. Adresse: https://cds.cern.ch/record/1100536.

[13] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 4. Aufl. Springer, 2014, ISBN: 978-3-642-45308-3. DOI: 10.1007/978-3-642-45309-0.

[14] Advanced Micro Devices, Inc., *Versal ACAP DSP Engine Architecture Manual (AM004)*, Version 1.2.1 English, Sep. 2022. Adresse: https://docs.amd.com/r/en-US/am004-versal-dsp-engine, Abruf: 09.03.2026.

[15] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), AI Engine Array Interface Architecture*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/AI-Engine-Array-Interface-Architecture, Abruf: 19.03.2026.

[16] Arm Limited oder verbundene Unternehmen, *AMBA® AXI-Stream, Protocol Specification*, Version B, Apr. 2021. Adresse: https://developer.arm.com/documentation/ihi0051/latest/, Abruf: 19.03.2026.

[17] Advanced Micro Devices, Inc., *Vitis High-Level Synthesis User Guide (UG1399), How AXI4-Stream Works*, Version 2025.2 English, Jan. 2026. Adresse: https://docs.amd.com/r/en-US/ug1399-vitis-hls/How-AXI4-Stream-Works, Abruf: 19.03.2026.

[18] florentw. „Adaptive SoC & FPGA Support. "Adresse: https://adaptivesupport.amd.com/s/question/0D54U00008bZPyHSAW/hello-a-silly-question-but-i-have-not-been-able-to-find-the-answer-in-the-related-documentation-in-the-name-ai-engines-what-does-ai-stand-for-adaptive-intelligent-or-artificial-intelligence. Abruf: 09.03.2026.

[19] Advanced Micro Devices, Inc., *AI Engine Tools and Flows User Guide (UG1076), AI Engine Architecture Overview*, Version 2025.2 English, Nov. 2025. Adresse: https://docs.amd.com/r/en-US/ug1076-ai-engine-environment/AI-Engine-Architecture-Overview, Abruf: 19.03.2026.

[20] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), AI Engine Architecture*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/AI-Engine-Architecture, Abruf: 19.03.2026.

[21] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), AI Engine Tile Architecture*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/AI-Engine-Tile-Architecture, Abruf: 19.03.2026.

[22] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), Performance*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/Performance, Abruf: 23.03.2026.

[23] Advanced Micro Devices, Inc. „AMD Versal™ AI Core Series VCK190 Evaluation Kit, Product Information Specifications. “Adresse: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vck190.html#tabs-a75507a83a-item-2c1d8dac01-tab. Abruf: 21.03.2026.

[24] Advanced Micro Devices, Inc., *VCK190 Evaluation Board User Guide (UG1366), Board Power System*, Version 1.2 English, Sep. 2024. Adresse: https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd/Board-Power-System, Abruf: 23.03.2026.

[25] Advanced Micro Devices, Inc., *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics (DS957), AI Engine Switching Characteristics*, Version 1.10 English, Juli 2024. Adresse: https://docs.amd.com/r/en-US/ds957-versal-ai-core/AI-Engine-Switching-Characteristics, Abruf: 23.03.2026.

[26] Advanced Micro Devices, Inc., *AI Engine Kernel and Graph Programming Guide (UG1079), Design Considerations for Graphs Interacting with Programmable Logic*, Version 2025.2 English, Nov. 2025. Adresse: https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Design-Considerations-for-Graphs-Interacting-with-Programmable-Logic, Abruf: 24.03.2026.

[27] Advanced Micro Devices, Inc., *Vivado Design Suite Properties Reference Guide (UG912), BLI*, Version 2025.2 English, Nov. 2025. Adresse: https://docs.amd.com/r/en-US/ug912-vivado-properties/BLI, Abruf: 24.03.2026.

[28] Advanced Micro Devices, Inc. „AXI Register Slice. “Adresse: https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/axi-register-slice.html. Abruf: 24.03.2026.

[29] Advanced Micro Devices, Inc., *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide (UG1387), Clock Domain Crossing*, Version 2025.2 English, Dez. 2025. Adresse: https://docs.amd.com/r/en-US/ug1387-acap-hardware-ip-platform-dev-methodology/Clock-Domain-Crossing, Abruf: 24.03.2026.

[30] Advanced Micro Devices, Inc. „AMD Versal™ AI Core Series. “Adresse: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/ai-core-series.html. Abruf: 09.03.2026.

[31] N. Henze, *Stochastik: Eine Einführung mit Grundzügen der Maßtheorie*, ger, 1. Aufl. Springer-Verlag, 2019, ISBN: 9783662595633. DOI: 10.1007/978-3-662-59563-3.

[32] P. Schwäbig, *MiniLORRAINE*, Gitlab, „This is an work-in-progress port of the Timepix3 DAQ firmware to the AMD/Xilinx Versal VCK190 Evaluation Board.“, Jan. 2026. Adresse: https://gitlab.uni-bonn.de/pschwaeb/minilorraine, Abruf: 24.03.2026.

[33] Advanced Micro Devices, Inc., *AI Engine Kernel and Graph Programming Guide (UG1079), Reading and Advancing an Input Stream*, Version 2025.2 English, Nov. 2025. Adresse: https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Reading-and-Advancing-an-Input-Stream, Abruf: 19.03.2026.

[34] Advanced Micro Devices, Inc./XILINX, *AI Engine Intrinsics User Guide, Cycle Counter*, Version r2p22, 2022. Adresse: https://download.amd.com/docnav/aiengine/xilinx2022_2/aiengine_intrinsics/intrinsics/group__intr__counter.html#ga0307e400f008e04b1c77590c26aac8e8, Abruf: 13.03.2026.

[35] cppreference.com, *C++ Reference, CV (Const And Volatile) Type Qualifiers*. Adresse: https://en.cppreference.com/w/cpp/language/cv.html, Abruf: 19.03.2026.

[36] International Organization for Standardization, *ISO/IEC 9899:1999 – Programming languages – C*, 2. Aufl., Dez. 1999.

[37] Advanced Micro Devices, Inc., *VCK190 Evaluation Board User Guide (UG1366), Clock Generation*, Version 1.2 English, Sep. 2024. Adresse: https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd/Clock-Generation, Abruf: 14.03.2026.

[38] Advanced Micro Devices, Inc., *Versal Adaptive SoC Technical Reference Manual (AM011), Clock Distribution Diagram*, Version 1.9 English, März 2026. Adresse: https://docs.amd.com/r/en-US/am011-versal-acap-trm/Clock-Distribution-Diagram, Abruf: 25.03.2026.

[39] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), AI Engine Interfaces*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/AI-Engine-Interfaces, Abruf: 21.03.2026.

[40] SciPy, *scipy.optimize.curve_fit*, Version 1.17.0. Adresse: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html, Abruf: 25.03.2026.

[41] Advanced Micro Devices, Inc., *Versal Adaptive SoC AI Engine Architecture Manual (AM009), Register Files*, Version 1.4 English, Feb. 2026. Adresse: https://docs.amd.com/r/en-US/am009-versal-ai-engine/Register-Files, Abruf: 16.03.2026.

[42] Advanced Micro Devices, Inc., *Vitis Tutorials: AI Engine Development (XD100), AI Engine Algorithm Performance Optimization*, Version 2025.2 English, Dez. 2025. Adresse: https://docs.amd.com/r/en-US/Vitis-Tutorials-AI-Engine-Development/AI-Engine-Algorithm-Performance-Optimization, Abruf: 16.03.2026.