FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Tensor Operations on Edge Devices

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
im Studiengang Informatik

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Till Billerbeck
geb. am 26.08.1999 in Dresden
Betreuer: Prof. Dr. A. Breuer

Jena, 30.07.2025

Kurzfassung

Effiziente Berechnung von Matrixmultiplikation ist seit langer Zeit ein stark erforschtes Feld. Die Forschung hat sich zumeist auf x86-64, Aarch64, GPUs und spezialisierte Hardware konzentriert. Für die neue Helium-Erweiterung (auch M-Profile Vector Extension / MVE genannt) der Cortex-M-Prozessorreihe existiert zum derzeitigen Stand nur eine Referenzbibliothek von Arm zur Matrixmultiplikation. In dieser Arbeit werde ich die Helium-Erweiterung und den Cortex-M55 detailliert beschreiben, um mithilfe der daraus gewonnenen Erkenntnisse eine Bibliothek zur Matrixmultiplikation zu implementieren. Dazu wird ein Just-in-Time-Compiler entwickelt, der Kernel zur Matrixmultiplikation erzeugt. Im Vergleich zur Referenzbibliothek von Arm wird ein Geschwindigkeitszuwachs von durchschnittlich 94% erzielt und 99% der theoretisch erreichbaren maximalen Leistungsfähigkeit des Prozessors erreicht.

Inhaltsverzeichnis

Vorwort

An dieser Stelle möchte ich mich für jegliche Hilfe beim Verfassen dieser Arbeit bedanken. Zuerst bei Prof. Dr. Alexander Breuer und Stefan Remke für die Betreuung der Arbeit und zahlreiche Hilfestellungen.

Zudem gilt mein Dank den weiteren Mitgliedern des Scalable Analyses Lab für weitere Anregungen, sowie Daniel Schicker für Motivation und die Beseitigung mehrerer Fehler im vorliegenden Text.

1. Einführung

Die effiziente Multiplikation von Matrizen ist seit langer Zeit ein stark erforschtes Thema. Während früher der Fokus auf wissenschaftlichen Berechnungen lag, ist Matrixmultiplikation heutzutage zudem integral für das Training und die Inferenz von Machine Learning-Algorithmen (ML) und Tensoroperationen. Das für Large Language Models (LLMs) unerlässliche Attention-Layer etwa nutzt Matrixmultiplikation [38] und in den Frameworks Tensor Processing Primitives (TPP) [19] und Tensor Operator Set Architecture (TOSA) [13] ist Matrixmultiplikation als unerlässliche Primitive zur Berechnung von ML-Modellen integriert.

Zugleich werden neue Anwendungsgebiete erschlossen, in denen der Einsatz von leistungsstarken Systemen nicht möglich ist, aber trotzdem ML-Anwendungen mit schneller Reaktionszeit und teilweise ohne Internetzugriff ausgeführt werden. Somit ist es meist nicht möglich, die anfallenden Anfragen an einen leistungsfähigeren Server zu übermitteln [39]. Auf diesen „Edge Devices“ ist dann häufig eine energiesparende CPU verbaut, der die Fähigkeiten zum Ausführen eines kompletten Betriebssystems fehlen, die aber trotzdem die Anfragen bearbeiten muss und leistungsstark genug sein muss, die Inferenz für einige ML-Anwendungen auszuführen. Für den Anwendungsfall der besonders ressourcenbeschränkten Systeme hat Arm die Cortex-M-Prozessorfamilie entwickelt.

Da SIMD-Instruktionen, welche ansonsten auf anderen CPUs mittels Erweiterungen wie Neon oder AVX-512 bereitgestellt werden, auf Cortex-M Prozessoren bisher auf eine Registergröße von 32 Bit beschränkt waren und nur mit Integerwerten funktionierten, war die erreichbare Leistung für ML-Anwendungen aufgrund der häufig niedrigen Taktfrequenzen stark beschränkt. Um auf den Cortex-M Prozessoren eine höhere Leistung für genannte Anwendungen zu erzielen, hat Arm die „Helium“-Erweiterung entwickelt und im Jahr 2019 zusammen mit der neuesten Version 8.1 des Arm M-Profils vorgestellt. Die Helium-Erweiterung ist eine Vektorerweiterung nach Vorbild der Neon-Erweiterung und teilt sich mit ihr einige Features und Prinzipien, wenn auch teilweise mit größeren Unterschieden, die v.a. in einem Konzept zur Überlappung von Instruktionen bestehen, welches elementar für optimale Leistung ist.

Während sich ein Großteil der Forschung dabei bisher auf die populären Architekturen wie x86-64 und Aarch64 sowie GPUs und eigenständige Hardwarebeschleuniger wie TPUs konzentriert hat, gibt es für die Helium-Erweiterung nach derzeitigen Stand nur eine Bibliothek zur Matrixmultiplikation, die Helium unterstützt. Ebenfalls gibt es keine Literatur, die sich mit der Optimierung von Matrixmultiplikation mittels der Helium-Erweiterung beschäftigt.

Daher werde ich in dieser Arbeit die Cortex-M Prozessorreihe mitsamt der Helium-Erweiterung beschreiben und ihre Performancecharakteristiken analysieren. Aus den gewonnenen Erkenntnissen wird ein JIT-Compiler zur effizienten Matrixmultiplikation entwickelt. Die entwickelte Lösung erreicht einen durchschnittlichen Geschwindigkeitszuwachs von 1,9x gegenüber der vorhandenen Referenzimplementierung von Arm und bis zu 99% der theoretisch erreichbaren Leistung des Cortex-M55.

Kapitel 2 beschreibt die Armv8.1-Architektur und die Helium-Erweiterung, welche als Grundlage für die Implementierung der Matrixmultiplikation genommen wird. In Kapitel 3 werden charakteristische Eigenschaften von Embedded Hardware betrachtet und das Alif Ensemble E7 als typischer Vertreter eines Embedded Systems vorgestellt. Kapitel 4 zeigt An- und Herausforderungen der Just-In-Time-Kompilierung auf Embedded Hardware, während in Kapitel 5 die Umsetzung der Matrixmultiplikation auf Embedded Hardware mithilfe der Helium-Erweiterung und eines Just-In-Time-Compilers beschrieben wird. Ka-

pitel 6 zeigt Ergebnisse der Implementierung und vergleicht die vorgestellte Lösung mit der Referenzimplementierung.

Die Helium-Erweiterung bietet weitaus mehr Instruktionen und Features als für Matrixmultiplikation benötigt wird; bei ihrer Beschreibung wird daher besonderer Wert auf die für Matrixmultiplikation benötigten Features gelegt und andere Neuerungen weggelassen.

Jeglicher genutzter Programmcode ist verfügbar unter https://github.com/01ill/e7-appkit-lab.

2. Embedded Hardware I: Arm M-Profil

Grundlage für die Implementierung der Matrixmultiplikation wird das Armv8.1-M-Profil sein, das etwa vom Cortex-M55 implementiert wird. Dies unterscheidet sich in einigen Punkten von dem bekannten Arm A-Profil, welches von den Prozessoren der Cortex-A-Reihe unterstützt wird. Aus diesem Grund werden im Folgenden die Grundlagen und Besonderheiten der Arm M-Profil-Architektur und des Cortex-M55 erklärt. Dabei werden zuerst die M-Profil-Prozessorreihe, die Eigenschaften des Cortex-M55 und die Unterschiede zum A-Profil beschrieben. Zum Schluss erfolgt eine detaillierte Charakterisierung des vom Armv8.1-M-Profil unterstützten Thumb2-Instruktionssatzes (T32) und insbesondere der Helium-Erweiterung. Dabei wird jeweils vor allem auf den Einfluss auf die Performance für Matrixmultiplikation eingegangen.

2.1. Cortex M-Profil Prozessorreihe

Die von Arm entwickelten Prozessorarchitekturen werden von Arm grundsätzlich in drei Kategorien eingeteilt: Einerseits die Cortex-A Prozessoren, welche das A-Profil implementieren und auf die Ausführung von Betriebssystemen ausgerichtet sind und sich somit in zahlreichen Systemen von Smartphones bis Laptops finden. Cortex-R Prozessoren weisen größere Überschneidungen mit Cortex-A Prozessoren auf, sind aber für den Einsatz in Echtzeit-Systemen, wie Modems oder Festplattencontrollern, konzipiert. Zuletzt gibt es die Cortex-M-Prozessoren, welche auf die Bedürfnisse von eingebetteten Systemen zugeschnitten sind. Außerdem gibt es die Arm Neoverse Prozessoren für den Servereinsatz, welche ebenfalls das A-Profil implementieren.

Im Unterschied zu den A-Prozessoren haben die M-Prozessoren kein Speicherverwaltungssystem und können somit keine modernen Betriebssysteme ausführen. Der Fokus wird auf stark ressourcenbeschränkte Anwendungen in eingebetteten Systemen gelegt. Statt 64-Bit wird auf eine 32-Bit-Architektur gesetzt, da es nicht nötig ist, mehr Speicher zu adressieren. Außerdem wird nicht auf eine mehrstufige Cache-Architektur und großen DRAM gesetzt, sondern typischerweise auf eine zweigeteilte Speicherarchitektur (siehe 3.2) bestehend aus einem mit Tightly Coupled Memory (TCM) realisierten Echtzeitspeicher, aus dem Speicheranfragen im Normalfall innerhalb eines Zyklus abgearbeitet werden, sowie einem mit L1-Cache versehenen latenzbehafteten Speichersystem. TCM und L1-Cache sind nicht auf allen Modellen der M-Reihe verfügbar und sind vom CPU-Hersteller in der Größe konfigurierbar.

2.1.1. Cortex-M55

Von Arm im Februar 2020 vorgestellt, war der Cortex-M55 der erste Prozessor, der auf dem Armv8.1-M Profil aufbaute und die Helium-Erweiterung unterstützte [42]. Somit war es auch das erste Modell, von dem kommerziell erhältliche Entwicklerboards wie das Alif

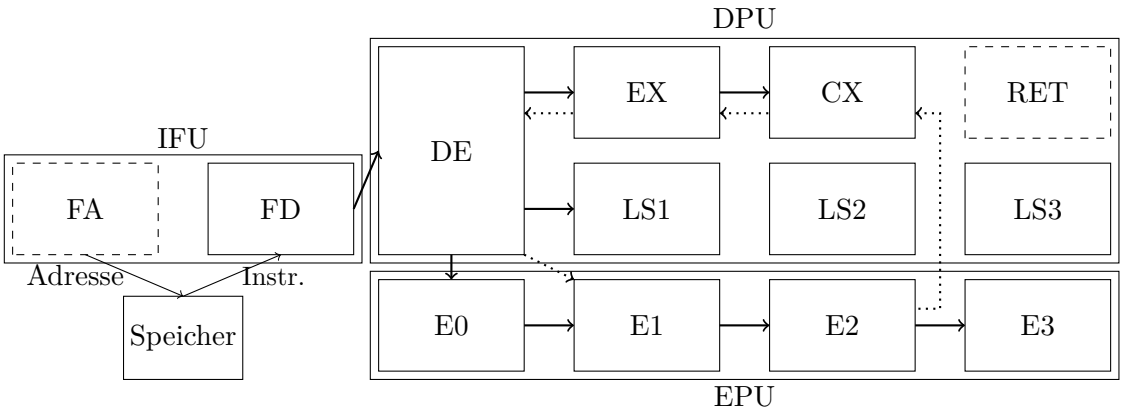


Abbildung 1: Pipeline des Cortex-M55. Grafik übernommen von [5]

E7 Ensemble erschienen, welches für die Umsetzung dieser Arbeit genutzt wurde. Kommerzielle Anwendung findet der Cortex-M55 u.a. in Smartwatches [31, 32] oder Kopfhörern [14].

Der Cortex-M55 siedelt sich mit seiner Leistungsfähigkeit in der Mitte der M-Profil-Prozessoren an, bietet mehr Leistung als M23 und M52, aber weniger Leistung als sein Nachfolger Cortex-M85 [11]. Der Grund für den Performanceunterschied liegt vor allem im Design der Pipeline. Daher erfolgt hier zuerst eine detaillierte Beschreibung der Pipeline gefolgt von einer Beschreibung der daraus implizierten Dual-Issue-Fähigkeiten sowie einer kurzen Beschreibung der typischen Speicherarchitektur.

Pipeline Beim Cortex-M55 wird zwischen den standardmäßigen skalaren Instruktionen sowie den Vektorinstruktionen der Helium-Erweiterung unterschieden. Für die Skalarinstruktionen wird eine vierstufige Pipeline genutzt, während für die Helium-Instruktionen eine fünfstufige Pipeline genutzt wird.[1] Die Pipeline ist dabei komplett In-Order, das heißt, dass alle Instruktionen in der Reihenfolge ausgeführt werden, in der sie abgesetzt werden und keine automatische optimierte Anordnung der Instruktionen durch die CPU vorgenommen wird. Für die Entwicklung von performanten Algorithmen bedeutet dies, dass jede Instruktion optimal angeordnet werden muss, um die theoretisch erreichbare Performance zu erreichen.

Die Pipeline ist aus der Instruction Fetch Unit (IFU) sowie der Data Processing Unit (DPU) und Extension Processing Unit (EPU) zusammengesetzt. Dabei ist die EPU nur für die Floating-Point- (FP) und Helium-Instruktionen zuständig und ist nur vorhanden, wenn auch die FP- oder Helium-Erweiterung implementiert wird.

Von der aus „Fetch Adress“-Stufe[2] (FA) und „Fetch Data“-Stufe (FD) aufgebauten IFU werden die Instruktionen aus dem Speicher geladen. Dabei werden in jedem Zyklus 32 Bit geladen, d.h. entweder eine 32-Bit- oder zwei 16-Bit-Instruktionen. Instruktionen können entweder aus dem TCM oder dem Instruktions-Cache geladen werden. Zudem wird die Erkennung des Endes einer Low-Overhead-Schleife (siehe 2.2.3) von der FA übernommen. Von der FD werden die geladenen Instruktionen an die Decode-Stufe (DE) der DPU gegeben.

[1] Vgl. Arm Ltd., Cortex-M55 Software Optimization Guide, S. 11ff. Ebenso wie die nachfolgende Beschreibung der Pipeline.

[2] Die englischen Bezeichnungen der Pipeline werden im Folgenden so wiedergegeben, wie sie im Software Optimization Guide des Cortex-M55 stehen.

Von der DE werden die Instruktionen dekodiert und Operanden-Register gelesen. Außerdem enthält sie die Logik für strukturelle Gefahren, d.h. Instruktionen, bei denen es zu Lese-/Schreib-Konflikten kommen kann. Helium- und FP-Instruktionen werden an die Decode-Stufe der EPU weitergeleitet; alle anderen Instruktionen können direkt dekodiert und von dort entweder an die EX oder LS1 weitergegeben werden. Es existieren drei Register-Lese-Ports wovon einer für Dual-Issue-Fälle reserviert ist. Zudem können Daten aus der EX und CX an die DE weitergeleitet werden.

Skalare Instruktionen werden von der DE an die „Simple EXecute"-Stufe (EX) gegeben, wo ein Großteil der arithmetischen Integer-Instruktionen abgearbeitet werden. Daten können entweder aus der Register-Bank oder der CX gelesen werden. Können Instruktionen nicht komplett bearbeitet werden wie z.B. bei Integer-Multiplikation, werden sie an die CX weitergegeben. Ansonsten werden die Ergebnisse bereits hier zurück in die Register geschrieben.

In der „Complex eXecute"-Stufe (CX) werden Instruktionen abgearbeitet, die nicht von der EX abgearbeitet werden können. Zum Schreiben von Ergebnissen stehen hier zwei Schreib-Ports zur Verfügung, die z.B. bei Long-Multiply-Instruktionen aber auch beim Verschieben von Werten aus den Vektorregistern benötigt werden. Müssen Daten zurück in den Speicher geschrieben werden, werden sie mit den Daten aus EX kombiniert und an RET gegeben. Die „Retire"-Stufe (RET) gibt Daten aus dem Registerspeicher an den TCM oder Datencache.

Lade- und Speicherinstruktionen werden von der DE an die „Load-Store Address"-Stufe (LS1) gegeben. Dort wird zuerst ermittelt, an welche Speicherschnittstelle die Anfrage gegeben werden muss. Zudem werden hier unausgerichtete Zugriffe ausgerichtet, wobei währenddessen die Pipeline gestalled wird, weshalb immer auf die korrekte Ausrichtung von Daten im Speicher geachtet werden muss. In der „Load-Store Read Data"-Stufe (LS2) werden die Daten aus dem Cache bzw. TCM gelesen. Die Pipeline wird solange gestalled, bis die Daten verfügbar sind. Werden Daten geschrieben, kommt die „Load-Store Write Transfer"-Stufe (LS3) zum Einsatz, welche Daten an die entsprechenden Schnittstellen zum Speichern weitergibt.

Beim Ausführen von FP- oder Helium-Instruktionen wird die Instruktion von der DE direkt an die Decode-Stufe der EPU (E0) gegeben, wo sie dekodiert wird. Außerdem werden hier Probleme für die Überlagerung von Helium-Instruktionen ermittelt und wenn nötig die Pipeline gestalled.

Danach kommt die Instruktion in die „Operand Register Read"-Stufe (E1), wo die Operanden-Register aus dem Registerspeicher oder späteren Stufen gelesen werden. Skalare Register werden von der DPU übertragen. Nachdem die Instruktion dekodiert und Operanden gelesen wurden, wird die Instruktion in der „EPU Arithmetic"-Stufe (E2) abgearbeitet. Mit wenigen Ausnahmen werden hier Floating-Point-Instruktionen innerhalb von einem Zyklus und Helium-Instruktionen innerhalb von zwei Zyklen ausgeführt. Ausnahmen beinhalten v.a. Operationen auf FP64-Werten und Division. Zum Schluss kommt die Instruktion in die Writeback-Stufe (E3) wo die Ergebnisse in den Vektor-Registerspeicher geschrieben werden.

In der EPU existieren die vier Ausführungspipelines System-Register (SY), Load/Store (LS), Integer Vector (I) und Floating Point (F), von denen jede für verschiedene Instruktionsarten zuständig ist. Dies ermöglicht die Überlagung von Helium-Instruktionen, indem verschiedene Pfade zur Ausführung verschiedener Instruktionstypen bereitgestellt werden.

Aus diesem Pipeline-Design folgt, dass es zwar möglich ist, in jedem Zyklus eine Instruktion auszuführen, wenn die Bedingungen dafür gegeben sind, aber darauf geachtet werden muss, dass die Instruktionen optimal angeordnet sind und Daten entweder im TCM oder

im Cache vorhanden sein müssen sowie korrekt ausgerichtet, da sonst die gesamte Pipeline aufgehalten wird.[3]

Dual Issue Der Cortex-M55 kann laut Arm entweder eine 32-Bit- oder in bestimmten Situationen auch zwei 16-Bit-Instruktionen zugleich ausführen, was von Arm als „Limited dual-issue of common 16-bit instruction pairs“[4] bezeichnet wird.

Für Dual Issue stehen dafür zwei Slots zur Verfügung, wobei 16-Bit-Instruktionen in drei Kategorien eingeteilt werden: Entweder sie sind gar nicht dual-issue-fähig, können nur aus Slot 0 oder zuletzt aus beiden Slots ausgeführt werden.[5] Meist können Instruktionen mit Immediates aus beiden Slots und die jeweilige Variante mit Register-Parametern nur aus dem ersten Slot ausgeführt werden, da die DE-Stage der Pipeline nur drei Register-Read-Ports besitzt und es somit nicht möglich ist, vier Register in einem Zyklus zu lesen.

In Tests war es mir aber nicht möglich, einen Performancezuwachs bei dual-issued-Instruktionen auszumachen. Dazu wurden verschiedene Instruktionsanordnungen der MOV-Instruktion in Register- und Immediate-Variante getestet. Außerdem wurde getestet, ob eine 16-Bit-CMN-Instruktion ohne dual-issue-Fähigkeit die gleiche Leistung erbringt. Wie in Tabelle 1 zu sehen ist, ist die Variante mit den aus beiden Slots dual-issue-fähigen MOV-Instruktionen so schnell, wie die nicht dual-issue-fähige CMN-Instruktion. Es ist allerdings ebenso ein Vorteil gegenüber der Nutzung von 32-Bit-Instruktionen sichtbar.

Test	CPU-Zyklen
2x 16-Bit-MOV Dual Issue aus beiden Slots	6492
2x 32-Bit-MOV	8592
2x 16-Bit-CMN Kein Dual Issue	6492
2x 32-Bit-CMN	8592

Tabelle 1: Vergleich von Dual-Issue, 16-Bit- und 32-Bit-Instruktionen. Jeweils 200 Iterationen unterbrochen von zwei NOP-Instruktionen.

Speicherschnittstellen Der Cortex-M55 unterstützt wie auch die anderen Prozessoren des M-Profils verschiedenste Schnittstellen, um auf Speicher zurückgreifen. Als Hauptschnittstelle wird ein 64-Bit-AXI-Bus genutzt über den der Hauptspeicher angebunden ist. Es werden pro Zyklus ein Instruktionszugriff von 32 Bit sowie ein Datenzugriff von zweimal 32 Bit also 64 Bit unterstützt. Der AXI-Bus ist, wenn vorhanden, mit dem Instruktions- und Datencache verbunden. Außerdem kann optional ein Instruktions- und Daten-TCM genutzt werden, wobei der Instruktions-TCM wieder mit einem 32-Bit-Pfad und der Daten-TCM mit vier unabhängigen 32-Bit-Pfaden angebunden ist. Damit wird ermöglicht, dass in jedem Zyklus die unterstützten 64 Bit Daten und 32 Bit Instruktionen geladen werden können. Prinzipiell ist es erlaubt, Instruktionen in den Daten-TCM zu speichern und Daten in den Instruktions-TCM; durch die unterschiedliche Anzahl an Pfaden für Instruktions- und Daten-TCM ist es aber empfehlenswert, Instruktionen in den Instruktions- und Daten in den Daten-TCM zu speichern (siehe 4.1).

Wie in Abbildung 2 zu sehen, ist dabei der TCM nah an der CPU angebunden und übergeht die Cache-Architektur völlig. Der über AXI angebundene Speicher verfügt über

[3] Vgl. Arm Ltd., Cortex-M55 Technical Reference Manual, S. 186.
[4] Arm Ltd., Cortex-M55 Technical Reference Manual, S. 32.
[5] Vgl. Arm Ltd., Cortex-M55 Software Optimization Guide, S. 16.

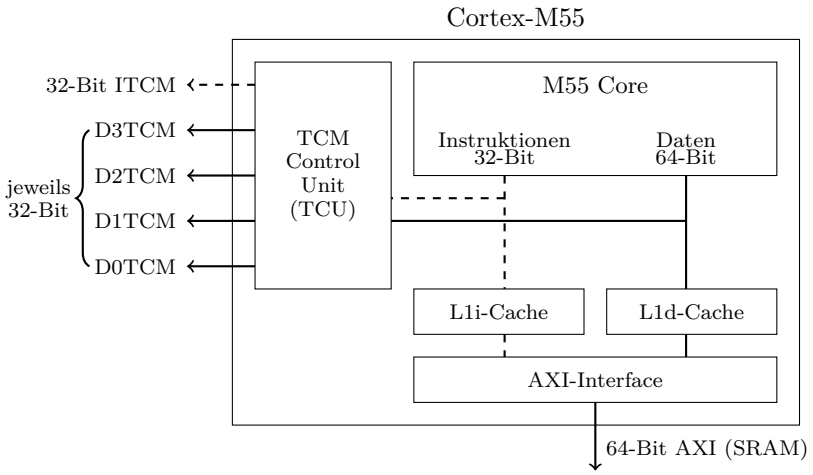


Abbildung 2: Speicheranbindung des Cortex-M55.[6]Abbildung übernommen von [40].

einen L1-Cache für Daten und Instruktionen, so dass auch hier bei optimiertem Caching 64 Bit an Daten pro Zyklus geladen werden können. Wie bereits aus der Pipelinebeschreibung hervorging, wird hier nochmal deutlich, wie wichtig es ist, dass die performance-kritischen Daten entweder im TCM abgespeichert werden oder mittels Caching dafür gesorgt wird, dass ein Großteil der Daten immer im Cache vorhanden ist, da es sonst zu Pipeline-Stalls kommen kann.

2.1.2. Weitere Prozessoren mit M-Profil

Neben dem Cortex-M55 existieren aktuell elf weitere Prozessoren der M-Familie im Angebot von Arm, wobei v.a. der Cortex-M85 als leistungsfähigerer Nachfolger des Cortex-M55 erwähnenswert ist. Im Gegensatz zum M55 ist hier ein Großteil der Instruktion dual-issue-fähig. Außerdem wird eine längere sieben-stufige In-Order-Pipeline für skalare Instruktionen und eine neun-stufige Pipeline für Helium-Instruktionen genutzt [7]. Zugleich führt er ebenso wie der M55 pro Zyklus zwei Beats einer Helium-Instruktion aus. Auch die Speicherarchitektur ist beim Cortex-M85 auf die erweiterte Performance ausgelegt, indem statt einer 32-Bit-Schnittstelle für das Laden von Instruktionen auf eine 64-Bit-Schnittstelle für das Laden von zwei Instruktionen pro Zyklus gesetzt wird.

2.2. Armv8.1-M-Profil

Im Februar 2019 von Arm angekündigt [18], ist das Armv8.1-M-Profil der Nachfolger des Armv8-M-Profils und bringt Änderungen wie die Helium-Erweiterung, Low-Overhead-Loops, Instruktionsmaskierung sowie Unterstützung für die Datentypen FP16 und INT8. Ansonsten ist Armv8.1-M rückwärts kompatibel zu Armv8-M.

Wie bereits seine Vorgänger stellt Armv8.1-M insgesamt 16 General-Purpose-Register (GP-Register) bereit, von denen drei für Link Register, Stack Pointer und Program Counter reserviert sind. Jedes dieser Register kann dabei 32 Bit speichern. Ebenso werden die Instruktionen des Thumb2-Instruktionssatz (T32) unterstützt.

Der T32-Instruktionssatz beinhaltet viele Instruktionen, die auch im A64-Instruktionssatz der Aarch64-Architektur vorhanden sind, wie Arithmetik-, Lade- und Speicher- sowie

[6]Vgl. Arm Ltd., Cortex-M55 Technical Reference Manual, S. 29ff

Branchinginstruktionen. Besonderheit von T32 ist die Verfügbarkeit von bestimmten Instruktionen in 16-Bit- und 32-Bit-Enkodierung, wobei die 16-Bit-Enkodierungen meist starke Einschränkungen, wie kleine Offsets und eingeschränkte Registerauswahl besitzen. In Armv6-M werden dabei von T32 nur 56 Instruktionen implementiert, von denen ein Großteil nur in 16-Bit-Enkodierung unterstützt wird [41]. Armv7-M unterstützt eine deutlich größere Anzahl an Instruktionen und fügt für diese eine 32-Bit-Enkodierung hinzu. Außerdem können durch Implementierung von Erweiterungen weitere Instruktionen hinzugefügt werden. Für Armv7-M und aufwärts gibt es die DSP- und Floating-Point-Erweiterung (FP).

Von der DSP-Erweiterung werden Instruktionen bereitgestellt, die nach dem Single-Instruction-Multiple-Data-Prinzip (SIMD) arbeiten, d.h. mit einer Instruktion können mehrere Werte bearbeitet werden. Bei der DSP-Erweiterung wird dabei immer noch auf einem GP-Register gearbeitet. Die Instruktion UADD16 etwa teilt ein GP-Register in zwei Teile auf und rechnet dann separat auf jeder Hälfte. Wie später beschrieben, ist die Helium-Erweiterung eine deutlich mächtigere Erweiterung, um Instruktionen nach dem SIMD-Prinzip zu implementieren.

Die FP-Erweiterung stellt verschiedene Instruktionen bereit, um mit Fließkommazahlen zu rechnen. Im Unterschied zu Helium-Instruktionen werden dabei allerdings nur skalare Werte berechnet, auch wenn die Instruktionen teilweise die gleichen Namen besitzen. Außerdem bringt die FP-Erweiterung 32 FP-Register, welche später von der Helium-Erweiterung wiedergenutzt werden. In Armv8-M werden FP32 und FP64 unterstützt, in Armv8.1-M zusätzlich FP16.

Für Armv8-M gibt es außerdem die Main-Erweiterung, die implementiert werden muss, um rückwärtskompatibel mit Armv7-M zu sein. Wird die Main-Erweiterung nicht implementiert, besteht nur Kompatibilität mit Armv6-M und die DSP- und FP-Erweiterungen können nicht implementiert werden.

Im Folgenden werden die wichtigsten Neuerungen des Armv8.1-Profils beschrieben.

2.2.1. Helium-Erweiterung

Da die bisherigen SIMD-Fähigkeiten der M-Reihe stark beschränkt waren, wird in Armv8.1-M die optionale M-Profile Vector Extension (MVE) eingeführt, die in Anlehnung an die bestehende Neon-Vektorerweiterung des A-Profils auch Helium-Erweiterung genannt wird. Dabei bestehen zwischen Neon und Helium sowohl einige Ähnlichkeiten, aber auch mehrere Unterschiede, so dass bei der Entwicklung von Algorithmen mit Helium bestehender Neon-Code meist nicht übernommen werden kann. Daher erfolgt hier zuerst ein Vergleich der Funktionalität von Helium- und Neon-Erweiterung gefolgt von einer Beschreibung der wichtigsten Konzepte der Helium-Erweiterung und deren Auswirkung auf die Performance. Zum Schluss werden einige Instruktionen vorgestellt, die für die Umsetzung von Matrixmultiplikation unerlässlich sind.

Sowohl Helium als auch Neon nutzen Vektorregister mit einer Länge von 128 Bit, die sich mit der Floating-Point-Einheit geteilt werden. Jedes Register wird in Lanes unterteilt, wovon jede ein Element beinhaltet, so dass bei einer Lanegröße von 32 Bit die Register in vier Lanes aufgeteilt werden und bei einer Instruktion vier Werte bearbeitet werden. Es werden Lanegrößen von 8, 16, 32 und 64 Bit unterstützt. Auch in der Semantik vieler Instruktionen bestehen Gemeinsamkeiten. So funktioniert beispielsweise das Addieren zweier Vektoren oder die gleichzeitige Multiplikation zweier Vektoren mit nachfolgender Addition auf einen Ergebnisvektor genau gleich, wenn auch mit unterschiedlicher Syntax. In den Instruktionen liegt der Unterschied vor allem darin, dass bei Helium keine Kombination von Vektoren mit einzelnen Lanes möglich ist, so dass keine Lane mit allen Elementen eines

Vektors addiert werden kann. Als Ersatz dafür stellt Helium bei vielen Instruktionen die Möglichkeit bereit, statt einer einzelnen Lane ein GP-Register anzugeben, welches dann als Lane interpretiert wird.

Anders als bei Neon stehen bei Helium statt 32 nur 8 Vektorregister zur Verfügung und es besteht nicht die Möglichkeit auf verschiedene Sichten zuzugreifen, so dass nur die 128-Bit-Q-Sicht unterstützt wird. Bei Helium besteht die Möglichkeit, Instruktionen zu maskieren (siehe 2.2.2), welche auf dem A-Profil erst mit der Scalable Vector Extension (SVE) eingeführt wurde [12]. Außerdem stehen Low-Overhead-Loops (siehe 2.2.3) und Tail-Predication (siehe 2.2.4) auf Neon nicht zur Verfügung.

Der große Unterschied zu Neon besteht aber in der unterliegenden Architektur, die es ermöglichen soll, Vektorinstruktionen auf ressourcenbeschränkten Cortex-M-CPUs performant auszuführen. Das grundlegende Problem der Cortex-M-CPUs besteht darin, dass aufgrund von Platz- und Kostengründen die Speicher- und Rechenbandbreite nicht ausreicht, um 128 Bit in einem Zyklus zu laden bzw. zu bearbeiten. So hat etwa der Cortex-M55 eine interne Bandbreite von 64-Bit und unterstützt zwei 32-Bit Speicherzugriffe pro Zyklus, womit die Berechnung eines 128-Bit-Vektors zwei Zyklen benötigt und es zwangsläufig zum Stocken der Pipeline kommt. Auch der leistungsfähigere Cortex-M85 besitzt nur eine Bandbreite von 64 Bit während der Cortex-M52 nur eine Bandbreite von 32 Bit besitzt, wodurch die Abarbeitung eines Vektor ganze vier Zyklen benötigt. Um dieses Problem zu lösen, ermöglicht es Helium, Instruktionen zu überlagern und gleichzeitig auszuführen, so dass die vorhandene Bandbreite zu allen Zeiten ausgenutzt werden kann. Damit dies zuverlässig funktioniert, wird die Ausführung von Vektorinstruktionen in sogenannte „Beats“ eingeteilt. Jeder Beat bearbeitet 32 Bit an Daten, so dass eine Instruktion in vier Beats unterteilt wird. Dabei ist es egal, welcher Datentyp ausgewählt, also auf wie vielen Lanes die Instruktion ausgeführt wird.[7] Je nach Prozessor können dabei einer (Single-Beat), zwei (Dual-Beat) oder vier (Quad-Beat) Beats einer Instruktion pro Zyklus ausgeführt werden. Der Cortex-M52 mit seiner 32-Bit-Bandbreite ist dabei Vertreter der Single-Beat-Systeme während Cortex-M55 und M85 zwei Beats pro Zyklus abarbeiten. Die folgende Darstellung der Überlagerungsmöglichkeiten konzentriert sich daher auf Dual-Beat-Systeme.

Im Cortex-M55 gibt es in der Vektorerweiterung je nach Instruktionstyp verschiedene Pfade (siehe 2.1.1), welche zeitgleich genutzt werden können. Dies erlaubt es, Instruktionen, welche unterschiedliche Pfade nutzen, gleichzeitig auszuführen.[8] Somit können in einem Zyklus insgesamt vier Beats und somit 128 Bit bearbeitet werden. Abbildung 3 zeigt, wie sich effektiv Lade-Instruktionen mit VFMA-Instruktionen überlagen lassen.

Bei Dual-Beat-Systemen werden beim Überlagern immer die letzten beiden Beats der ersten Instruktion[9] mit den ersten beiden Beats der zweiten Instruktion überlagert. Es ist nicht möglich, etwa die zweiten und dritten Beats zu überlagern. Dieses Design erlaubt es, auch Instruktionen gleichzeitig auszuführen, bei denen die erste Instruktion in ein Vektorregister schreibt und die zweite Instruktion aus diesem Register liest, da das Ergebnis der ersten beiden Beats bereits verfügbar ist, wenn die zweite Instruktion darauf zugreifen will. Im umgekehrten Falle - erst lesen, dann schreiben - gilt das gleiche: Wenn die zweite Instruktion schreibt, sind die Werte von der ersten Instruktion bereits gelesen, so dass es zu keinem Konflikt kommt.

Grundsätzlich lassen sich alle Vektorinstruktionen überlagen, wenn diese unterschiedliche Pfade in der Pipeline nutzen; aber auch Standard-Instruktionen können teilweise

[7] Vgl. ARM V8M Referenz, B5.1, S. 178.

[8] Im Cortex-M85 erlaubt der Einsatz von Instruktionsgruppen in der Pipeline sogar, etwa Lade- und Speicherinstruktionen zu überlagern.

[9] Bei Single-Beat kann nur der letzte Beat überlagert werden

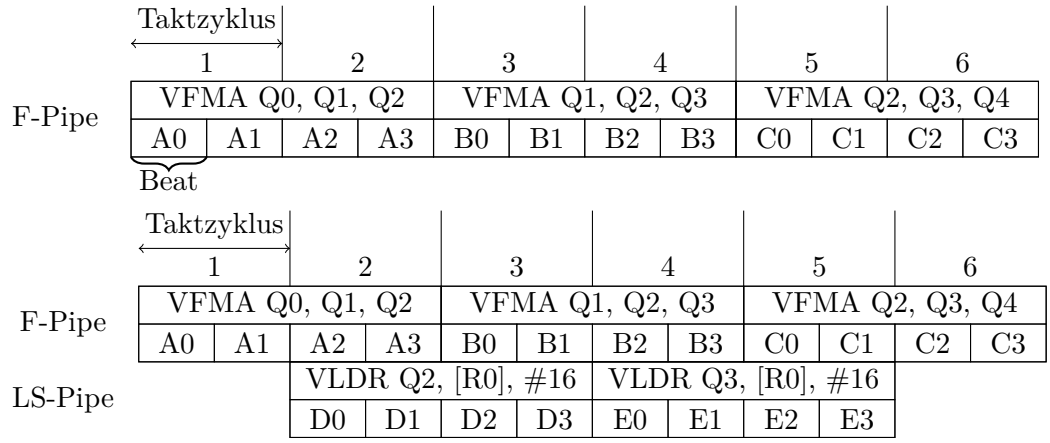


Abbildung 3: Überlappung von Instruktionen. VFMA nutzt Floating Point-Pipeline und VLDR die Load/Store-Pipeline. A0, A1, usw. bezeichnet die einzelnen Beats der Instruktionen.

überlagert werden, auch wenn diese nicht mithilfe von Beats ausgeführt werden. Stattdessen können die letzten beiden Beats einer Vektorinstruktion mit der gesamten nachfolgenden Instruktion überlagert werden, wie in Abbildung 4 gezeigt. Nicht möglich ist es, eine Instruktion zu überlagern, wenn diese vor der Vektorinstruktion ausgeführt wird, da dies in den meisten Fällen einem Dual-Issue der Instruktionen gleichkommen würde, was vom Cortex-M55 nicht unterstützt wird.[10] Hinzu kommen folgende Ausnahmen, in denen Instruktionen nicht überlagert werden können: (1) Arithmetische Skalarinstruktion steht hinter einer Lade- bzw. Speicher Vektorinstruktion mit Write-Back; (2) eine Skalarinstruktion steht hinter der letzten Instruktion in Low-Overhead-Loop und (3) wenn eine Abhängigkeit zwischen beiden Instruktionen besteht.

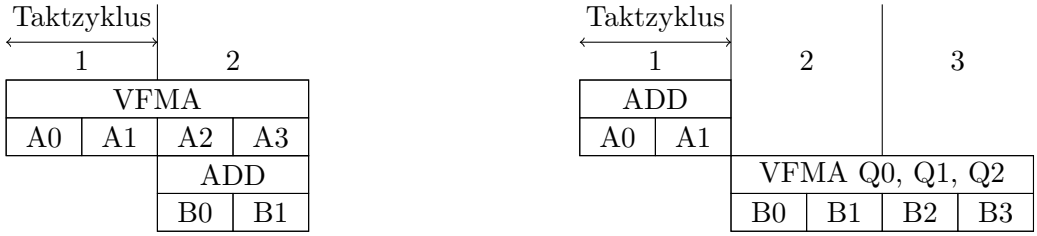


Abbildung 4: Überlappung von Vektor- mit Standard-Instruktionen.

Die Helium-Erweiterung erlaubt es somit, mit einem 64-Bit-Datenpfad pro Zyklus 128 Bit zu bearbeiten und pro Zyklus eine Instruktion auszuführen, selbst wenn die Helium-Instruktionen nur einen Durchsatz von einer halben Instruktion und eine Latenz von zwei Zyklen haben. Für die Entwicklung performanter Algorithmen bedeutet dies allerdings zusätzlichen Aufwand, da jede Instruktion optimal angeordnet werden muss, damit auch die gesamte Leistungsfähigkeit abgerufen werden kann. Außerdem muss die begrenzte Anzahl an Vektorregistern in Zusammenspiel mit der fehlenden Option, auf einzelne Lanes zuzugreifen, beachtet werden.

Für den Zweck dieser Arbeit folgt hier noch eine Beschreibung der wichtigsten Instruk-

[10] Vgl. Arm Ltd., Cortex-M55 Software Optimization Guide, S. 46. Der Cortex-M85 hingegen erlaubt es durch seine Dual-Issue-Pipeline auch, bestimmte Instruktionen zu überlagern, wenn diese vor der Vektorinstruktion ausgeführt werden. Vgl. Arm Ltd., Cortex-M85 Software Optimization Guide, S. 43.

tionen, die für die Umsetzung effizienter Matrixmultiplikation mit FP32-Werten nötig sind.

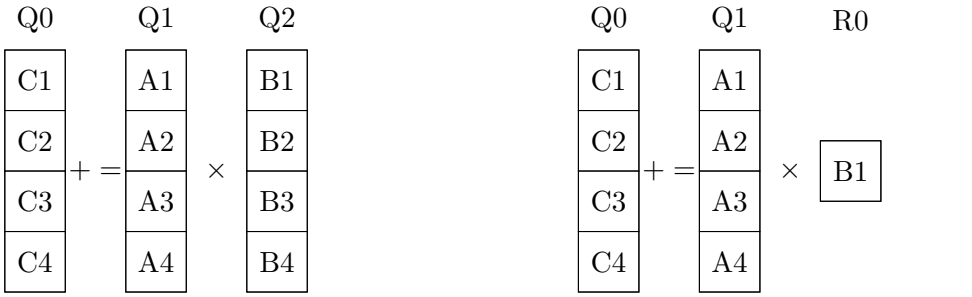
VFMA Die VFMA-Instruktion führt, wie in 5a gezeigt, eine Multiplikation auf den Elementen zweier Vektorregister durch und addiert die Ergebnisse auf die Elemente eines zusätzlichen Vektorregisters. Anders als bei Neon gibt es nicht die Möglichkeit, das erste Vektorregister mit einer einzelnen Lane des zweiten Vektorregisters zu multiplizieren. Dazu muss immer der Datentyp angegeben werden, sowie die drei benötigten Register:

```
VFMA.F32 Q0, Q1, Q2
```

Als Alternative wird von Arm angedacht, dafür ein GP-Register zu nutzen. In diesem Fall wird, wie in 5b gezeigt, jeder Wert im ersten Vektorregister mit dem Wert des GP-Registers multipliziert und die Ergebnisse auf die Werte des Vektor-Zielregisters addiert.

```
VFMA.F32 Q0, Q1, R0
```

Die VFMA-Instruktion unterstützt die Datentypen FP16 und FP32, wobei jeweils acht bzw. vier Elemente multipliziert und addiert werden.



(a) VFMA mit Vektorregistern (b) VFMA mit General Purpose Register

Abbildung 5: VFMA-Varianten (jeweils FP32)

VLDRW und VSTRW Mit den VLDR- und VSTRW-Instruktionen werden Daten aus dem Speicher in Vektorregister geladen bzw. zurückgeschrieben. Grundsätzlich muss nur das Zielvektorregister, sowie das GP-Register welches die Speicheradresse hält, angegeben werden. Die VLDR-Instruktion lädt dann 128 Bit von der angegebenen Adresse. Wie bei Neon besteht dabei auch die Möglichkeit, mittels eines Immediates den Abstand zur Speicheradresse anzugeben, wobei der Abstand vor bzw. nach dem Laden angewandt und ebenfalls zurückgeschrieben werden kann. Anders als bei der LDR-Instruktion besteht aber nicht die Möglichkeit, ein GP-Register anzugeben, welches den Abstand angibt; einzige Alternative ist die Angabe eines Vektorregisters, welches den Abstand für jedes Element angibt.

Es ist nur möglich, einen Abstand von 512 Bytes (auf vier Bytes ausgerichtet) anzugeben, was bei FP32 bedeutet, dass nur 127 Elemente übersprungen werden können. Für Anwendungsfälle mit größeren Matrizen bedeutet dies, dass zwangsläufig Skalarinstruktionen zum Modifizieren der Zeiger mit Vektorinstruktionen vermischt werden müssen, wenn der Abstand zu den nächsten Werten im Speicher zu groß ist, um diesen mit einem Immediate abzubilden.

Anders als bei NEON ist es nicht möglich, mehr als 128 Bit mit einer Instruktion gleichzeitig zu laden, einzig die Instruktionen VLD2/VST2 und VLD4/VST4 existieren, welche allerdings 128 Bit auf zwei bzw. vier Vektorregister verteilen, wie in Abbildung 6

für VLD2 gezeigt. Dies würde sonst mit dem beschriebenen Beat-Konzept zwangsläufig zu Stalls in der Pipeline führen, da die Lade-Instruktionen mit anderen Instruktionstypen überlagert werden müssen.

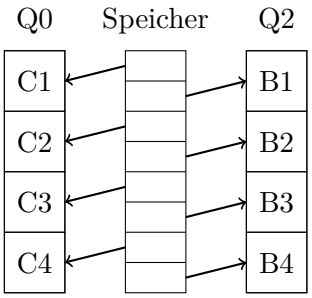


Abbildung 6: Funktionsweise der VLD2-Instruktion

2.2.2. Maskierung

Wenn es überschüssige Elemente gibt, diese aber ein Vektorregister nicht komplett füllen, müssen die verbleibenden Elemente maskiert werden, so dass keine Elemente aus dem Speicher geladen oder gespeichert werden, da dies zu unzulässigen Speicherzugriffen führen kann. Damit diese verbliebenen Elemente nicht mit skalaren Code bearbeitet werden müssen, stellt die Helium-Erweiterung die Möglichkeit bereit, Vektorinstruktionen zu maskieren, damit nur bestimmte Lanes bearbeitet werden. Dafür wird das Maskierungsregister VPR hinzugefügt, in dem gespeichert wird, wie viele Instruktionen und welche Lanes maskiert werden. Der Wert P0 im Register bestimmt, welche Lanes maskiert werden und die Werte MASK23 und MASK01 bestimmen, wie viele Instruktionen maskiert werden.[11] In jeder Vektorinstruktion wird geprüft, ob die Instruktion maskiert werden muss, indem das VPR-Register ausgelesen wird. Außerdem werden die MASK-Werte nach jeder Instruktion zurückgeschrieben, so dass nicht unendlich viele Instruktionen maskiert werden.

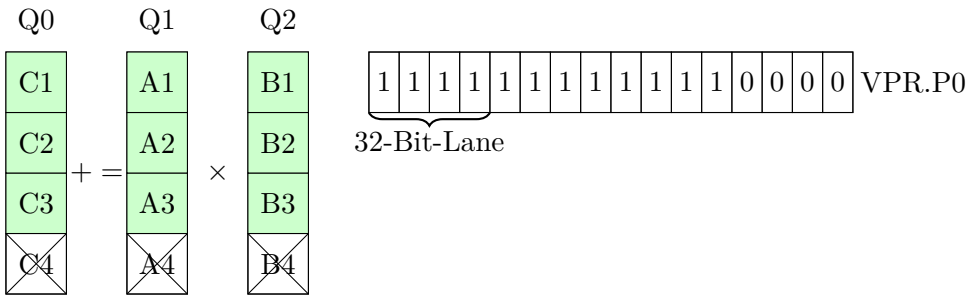


Abbildung 7: Maskierung einer VFMA-Instruktion. Lanes 1 bis 3 werden berechnet, während Lane 4 ignoriert wird

Mit der Instruktion VCTP wird angegeben, wie viele Lanes maskiert werden sollen, indem der Wert P0 gesetzt wird. Diese Instruktion muss nur einmal ausgeführt werden, da die Einstellung danach nicht zurückgesetzt wird. Außerdem muss die Instruktion VPST ausgeführt werden, mit der gesetzt wird, wieviele der folgenden Instruktionen maskiert werden. Abbildung 7 zeigt folgendes Beispiel einer einzelnen maskierten VFMA-Instruktion:

[11] Vgl. Arm Ltd., Armv8-M Reference Manual, S. 186f, 1900.

```
mov r0, #3
vctp.32 r0
vpst
vfmat.f32 q0, q1, q2
```

Da die kleinste Lanegröße 8 Bit beträgt, werden bei Angabe von 32-Bit-Lanes für jede Lane insgesamt vier Werte in P0 gesetzt.

2.2.3. Low Overhead Loops

Um Schleifen in Assembly umzusetzen, wird typischerweise eine Kombination aus Vergleichs-und Branch-Instruktionen genutzt. Dabei kann es zu einer Leerung der Pipeline und einem damit verbundenen Stocken der Programmausführung kommen, wenn einem Branch gefolgt wird, der nicht dem Standard-Programmfluss folgt, da an einen neuen Ort gesprungen wird, von dem noch keine Instruktionen geladen sind. Auf modernen CPUs wie der Cortex-A-Reihe sind dafür komplexe Branchvorhersageeinheiten verbaut, die ermitteln, welchem Branch gefolgt wird und von dort bereits die Pipeline füllen, womit der zusätzliche Aufwand minimiert wird. Aufgrund von Kosten- und Platzgründen sind aber nicht auf allen Cortex-M-CPUs Branchvorhersageeinheiten verbaut, weshalb durch Schleifen ein erheblicher Mehraufwand entstehen kann. Aus diesem Grund [21] wurden die sogenannten Low-Overhead-Loops bzw. Low-Overhead-Branches (LOB) als optimierte Alternative zu traditionellen Schleifenstrukturen zu Armv8.1-M hinzugefügt.[12]

Grundidee der Low-Overhead-Loops ist es, mittels einem einzelnen zusätzlichen Register und wenigen zusätzlichen Instruktionen Schleifenstrukturen bereitzustellen, die auch ohne Branchvorhersage ohne zusätzlichen Mehraufwand ausgeführt werden.

Umsetzung Wie erwähnt, wird nur ein einzelnes zusätzliches 67-bit-Register[13] zum Speichern des Loop-Caches benötigt. Der Loop-Cache besteht dabei aus Start- und Endadresse der jeweiligen Schleife sowie aus Flags um zu speichern, ob man eine unendliche Schleife eingerichtet hat bzw. ob der Loop-Cache noch gültig ist.[14] Der Loop-Zähler wird nicht im Cache gespeichert, sondern im Link Register.

In der ersten Iteration der Schleife wird dieser Cache sowie das Link Register mittels einer Instruktion zum Beenden und einer zum Starten der Schleife befüllt. In den folgenden Iterationen wird in der Fetch-Stufe der Pipeline geprüft, ob die Programmadresse mit der Endadresse im Loop-Cache übereinstimmt, wobei die Instruktion zum Beenden des Loops nicht mehr ausgeführt wird. Wenn dies der Fall ist, wird wieder an die Startadresse des Loops gesprungen, insofern der Zähler im Link Register noch nicht auf Null dekrementiert wurde.

Somit fällt einerseits ein Großteil des Branching-Mehraufwands weg; zusätzlich erlaubt diese Architektur bei Implementierung der Helium-Erweiterung eine Überlagerung der letzten und ersten Instruktion der Schleife, was bei passenden Instruktionen zu einem weiteren Performancegewinn führt (siehe 2.2.1).

Anwendung Es werden drei zusätzliche Instruktionen eingeführt: WLS (While Loop Start) und DLS (Do Loop Start) starten eine Schleife und kopieren die Anzahl der Iterationen (per Register angegeben) in das Link Register. WLS bietet dabei zusätzlich die Möglichkeit, direkt zum Ende der Schleife zu springen, wenn keine Iterationen ausgeführt

[12] Vgl. J. Marsh, Arm Helium Technology Reference Book, S. 32.
[13] Vgl. Arm Ltd., Armv8-M Reference Manual, D1.2.163, S. 1735.
[14] Vgl. Arm Ltd., Armv8-M Reference Manual, B3.28, S. 135f.

werden müssen. LE (Loop End) beendet eine Schleife und befüllt den Loop-Cache in der ersten Iteration, wobei die Startinstruktion des Loops angegeben werden muss, sowie das Link Register, um dieses in jeder Iteration zu dekrementieren.

Um zu prüfen, welchen Performancevorteil die Low-Overhead-Loops bieten, wurden verschiedene Schleifen-Varianten getestet, die in Abbildung 8 dargestellt sind. Einerseits eine Schleife mittels Backwards-Branch, bei dem sich die Branch- und CMP-Instruktionen am Ende der Schleife befinden und in jeder neuen Iteration an den Start der Schleife gesprungen wird. Zudem zwei Varianten von Forward-Branch, bei dem sich zusätzliche Branch-Instruktionen am Start der Schleife befinden und nach der letzten Iteration an das Ende der Schleife gesprungen wird, während in jeder Iteration bedingungslos an den Start gesprungen wird. Eine Variante davon nutzt die die CBZ-Instruktion, welche die CMP- und Branch-Instruktion kombiniert. Zuletzt wird eine einfache Schleife mit Low-Overhead-Loops getestet.

```
mov r1, #0
loopStartB1:
vfma.f32
vfma.f32
add r1, #1
vfma.f32
cmp r1, r0
vfma.f32

blt loopStartB1
```

```
mov r1, #0
loopStartB2:
cmp r1, r0
bge b2End
vfma.f32
vfma.f32
add r1, #1
vfma.f32
vfma.f32
b loopStartB2
```

```
dls lr, r0

loopLOB:
vfma.f32
vfma.f32
vfma.f32
vfma.f32

le lr, loopLOB
```

```
cbzStart:
cbz r0, cbzEnd

vfma.f32
vfma.f32
subs r0, #1
vfma.f32
vfma.f32
b cbzStart
```

Abbildung 8: Micro-Benchmark von Branch-Backward, Branch-Forward, Low-Overhead Loop und CBZ-Branch

Wie in Tabelle 2 zu sehen, sind Schleifen mittels Low-Overhead-Loops schneller, selbst wenn wie beim Cortex-M55 eine einfache Branchvorhersageeinheit verbaut ist. Gegenüber dem optimierten Backward-Branch ist der Low-Overhead-Loop um 12% schneller. Hingegen sind Forward-Branch und der modifizierte Forward-Branch mit CBZ-Instruktion etwa 11% langsamer als der Backward-Branch, da beim Cortex-M55 bei Backwards-Branch die Latenz nach der ersten Iteration Null ist.[15] Ein weiterer Vorteil des Low-Overhead-Loops ist die Einsparung eines zusätzlichen Registers, welches zum Speichern der Loop-Variable bei den anderen Methoden benötigt wird. Außerdem fällt der Programmieraufwand zum Prüfen der Schleifenbedingung weg, was die Implementierung einer Schleife einfacher gestaltet.

Test	Zeit abs. (ms)	Zeit rel.	Speedup
Backwards-Branch	2241	1.0	1.0
Forward-Branch	2489	1.11	0.9
Low Overhead Branch	1992	0.89	**1.12**
Forward-Branch CBZ	2489	1.11	0.9

Tabelle 2: Ergebnisse der verschiedenen Schleifen. Jede Variante wurde optimiert, so dass Instruktionen sich überlagern.

Grundsätzlich müssen sowohl 16- als auch 32-Bit-Instruktionen auf 2 Byte ausgerichtet werden. Für Helium-Instruktionen im Low-Overhead-Loop ist es zudem wichtig, dass

[15] Vgl. Arm Ltd., Cortex-M55 Software Optimization Guide, S. 17f

sie auf 4 Byte ausgerichtet werden, da es sonst pro Iteration zu einer Verzögerung von einem Zyklus kommen kann [43]. Dies liegt daran, dass von der IFU (siehe 2.1.1) der Pipeline in jedem Zyklus 32 Bit geladen werden. Wie Abbildung 9 zeigt, kann bei fehlender Ausrichtung der ersten VFMA-Instruktion die letzte VLDR-Instruktion nicht mit der VFMA-Instruktion überlagert werden, da insgesamt zwei Zyklen zum Laden der kompletten VFMA-Instruktion benötigt werden. Von der IFU wird das Schleifenende erkannt und die erste Instruktion am Schleifenanfang geladen. Der TCM führt aber jeglichen Lese- und Schreibzugriff immer auf 4 Byte ausgerichtet aus.[16] Somit wird nicht die gesamte erste 32-Bit-Instruktion geladen, sondern die letzten 16 Bit vor der Schleife und die ersten 16 Bit der ersten Instruktion. Um die Instruktion auszuführen, ist nun ein weiterer Zyklus nötig.

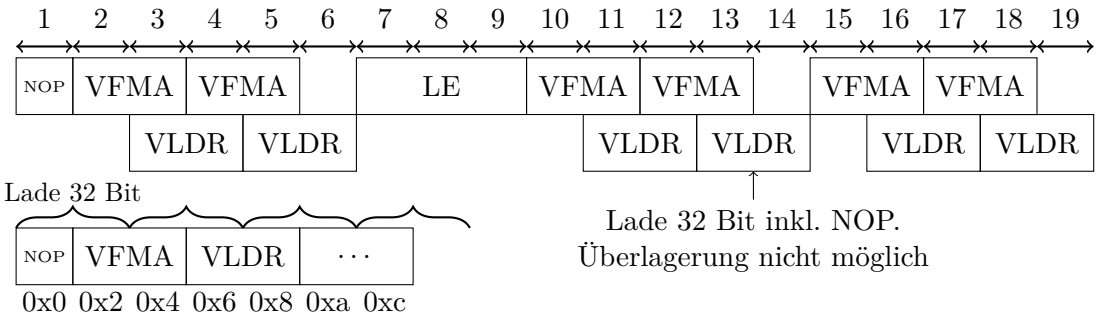


Abbildung 9: Darstellung Problematik Code-Alignment. Start des Loops ist die erste VFMA-Instruktion. NOP ist 16-Bit Variante.

Dies stellt nur ein Problem dar, wenn sich letzte und erste Instruktion überlagern können, da sonst durch den Ressourcenkonflikt auch ein Stall entstehen würde, der die zusätzliche Ladezeit überdeckt. Auch in der normalen Programmausführung stellt das kein Problem dar, da man höchstens bei der ersten Helium-Instruktion eine Instruktion länger warten muss, da dies aber in Low-Overhead-Loops in jeder Iteration auftritt, muss hier besonders drauf geachtet werden. Tabelle 3 zeigt, dass bei einer Schleife mit 10000 Iterationen auch knapp unter 10000 mehr Zyklen benötigt werden, wenn sich die Instruktionen überlagern können. Wenn hingegen nur VFMA-Instruktionen ausgeführt werden, hat die Ausrichtung der Instruktionen keinen Einfluss auf die Laufzeit, da sich das Stocken der Pipeline mit dem Nachladen der Instruktion überlagert.

Test	Zeit (μs)	Zyklen
Helium Aligned	100	40245
Helium Non-Aligned	125	50179
Helium Aligned VFMA	200	80174
Helium Non-Aligned VFMA	200	80018

Tabelle 3: Test von ausgerichteten und nicht ausgerichteten Funktionen mit 10000 Iterationen einer Schleife.

Prinzipiell sollte also mindestens die erste Instruktion des Loops auf 4-Byte ausgerichtet werden. Dies kann etwa durch Einfügen einer zusätzlichen 16-Bit-NOP-Instruktion außerhalb der Schleife passieren. Bei handgeschriebenen Assembly-Code besteht dafür die Direktive `.p2align`, die den Programmcode ausrichtet.

[16] Vgl. Arm Ltd., Cortex-M55 Technical Reference Manual, Table C-7, S. 369.

2.2.4. Tail-Predication

Beim Umwandeln von skalaren in vektorisierten Code muss immer berücksichtigt werden, dass die Anzahl der zu bearbeitenden Elemente häufig kein Vielfaches der Werte im Vektorregister ist. Dies führt dazu, dass man die übrigen Elemente entweder mit skalaren Instruktionen oder maskierten Vektorinstruktionen abarbeiten muss. Um dies zu vermeiden, wird in Armv8.1-M die sogenannte Tail-Predication eingeführt, welche die bereits beschriebenen Low-Overhead-Loops erweitert. Dabei speichert das Link Register nicht mehr die Anzahl an Iterationen sondern die Anzahl an zu bearbeitenden Elemente, wobei in jeder Iteration die Anzahl der bearbeiteten Elemente abgezogen wird.[17] In der letzten Iteration werden dann zusätzlich alle Vektorinstruktionen automatisch maskiert, so dass sie die übriggebliebenen Elemente bearbeiten.

Damit dies genutzt werden kann, werden die Instruktionen WLSTP, DLSTP und LETP eingeführt, die Äquivalente zu den Low-Overhead-Loop-Instruktionen sind. Dabei wird bei WLSTP und DLSTP nun zusätzlich angegeben und bei Ausführung gespeichert, welche Größe die Vektor-Elemente besitzen. Die Datentypgröße entscheidet, wie viele Elemente in jeder Iteration bearbeitet werden. Zum Schluss der Schleife muss statt LE nun LETP ausgeführt werden. Nach jeder Iteration wird die Anzahl der bearbeiteten Elemente vom Link Register abgezogen.

In Listing 10 ist beispielhaft gezeigt, wie durch Einsatz von Tail-Predication vektorisierter Code zum Addieren zweier Vektoren deutlich vereinfacht werden kann. Es wird eine Größe von 32 Bit angegeben, da als Datentyp FP32 genutzt wird. Somit werden durch die VADD-Instruktion in jeder Instruktion vier Elemente addiert. Ist die Anzahl der Elemente nicht durch vier teilbar, werden die Instruktionen in der letzten Iteration maskiert und nur die benötigten Elemente geladen und gespeichert.

```
dls lr, r0 // r0 = vec_len / 4
lStart:
  vldrw q1, [r1], #16
  vldrw q2, [r2], #16
  vadd.f32 q0, q1, q2
  le lr, lStart
lEnd:
  // vec_len % 4 übrig
  vctp // maskieren
  vpst
  vldrwt q1, [r1], #16
  vldrwt q2, [r2], #16
  vaddt.f32 q0, q1, q2
  vstrwt q0, [r3], #16
 ...
```

```
dlstp.32 lr, r0 // r0 = vec_len
lStart:
  vldrw q1, [r1], #16
  vldrw q2, [r2], #16
  vadd.f32 q0, q1, q2
  vstrw q0, [r3], #16
  letp lr, lStart
lEnd:
  ...
```

Abbildung 10: Vergleich von Standard-Schleife und Tail-Predication bei Vektor-Addition

2.3. Arm Microcontroller Software-Stack

Um die Entwicklung von Embedded Software für Entwickler zu vereinfachen, stellt Arm unter dem Namen „Cortex Microcontroller Software Interface Standard“ (CMSIS) verschiedene Softwarebibliotheken und Anwendungen bereit [10]. Für diese Arbeit sind dabei insbesondere die Tools zur Kompilation erwähnenswert.

[17] Vgl. J. Yiu, Blending DSP and ML features into a low-power general-purpose processor, S. 9.

CMSIS ist dabei grundlegend eine Schnittstelle für Hardware- und Software-Entwickler um eine Abstraktionsebene für die Hardware bereitzustellen, so dass etwa Treiber und Software schneller entwickelt werden können. Es werden die Bibliotheken CMSIS-NN [27] und CMSIS-DSP bereitgestellt, um ML- und DSP-Kernel auszuführen.

Über YAML-Dateien können Projekte konfiguriert werden, um diese dann mit verschiedenen Toolchains zu kompilieren. Von Arm wird eine eigene Version von clang bereitgestellt, sowie eine gcc-Version mit Unterstützung für das M-Profil, wobei die Unterstützung für Helium in armclang besser ausgebaut ist. Etwa Tail Predication wurde erst in der neuesten GCC15-Version hinzugefügt [17], die aber in CMSIS noch nicht verfügbar ist. Aufgrund der besseren Performance wird in dieser Arbeit immer armclang genutzt, was von Arm bei der DSP-Bibliothek auch explizit empfohlen wird.

Außerdem werden von Arm sogenannte Intrinsics bereitgestellt, d.h. Funktionen, mit denen direkt im C-Code Platzhalter eingefügt werden können, die dann beim Kompilierungsprozess zu den angegebenen Instruktionen kompiliert werden.

3. Embedded Hardware II: Alif E7 Board

Neben den bereits erwähnten kommerziellen Umsetzungen existieren es auch mehrere Implementierungen des Cortex-M55 in Entwicklerboards. Eine der ersten Umsetzungen waren die „Ensemble“-Entwicklerboards von Alif Semiconductors, die diese 2023 veröffentlichten [33]. Zu diesen gehört auch das Ensemble E7-Board, welches in dieser Arbeit genutzt wird. Das Ensemble E7 bietet eine typische Implementierung eines Cortex-M Prozessors mit L1-Cache und zwei Speicherbereichen bestehend aus SRAM und TCM. Es werden zwei Cortex-M55, die sich v.a. in der Taktfrequenz unterscheiden, verbaut. Auf beiden ist die Helium-Erweiterung mit Integer- und Floating-Point-Unterstützung verbaut.

Von Alif wird eine Integration in das CMSIS-Framework bereitgestellt, wodurch der benötigte Code zum Booten geladen werden kann. Außerdem werden automatisch Cache und Features wie Low-Overhead-Branches aktiviert. Zum Ausführen von Tests wird ein Timer mit 32768 Hz bereitgestellt.

Abschnitt 3.1 beschreibt das Rechensystem des Ensemble E7 mit seinen in zwei verschiedenen Bereichen verbauten Kernen während 3.2 die verbauten Speicherbereiche und deren Performance beschreibt.

3.1. Rechenleistung

Auf dem Ensemble E7 sind ein sogenannter High-Efficiency- (HE) und ein High-Performance-Bereich (HP) verbaut, sowie ein Application-Subsystem (APSS). Die HE und HP-Bereiche nutzen dabei jeweils eine Cortex-M55 CPU, welche im HP-Bereich mit 400 MHz und im HE-Bereich mit 160 MHz taktet. Im Application-Bereich kommt eine Cortex A32-CPU in Dual-Core Konfiguration zum Einsatz. Da für die Implementierung der Matrixmultiplikation mit der Helium-Erweiterung der Anwendungsbereich nicht nötig ist, wird die Beschreibung dieses Bereiches hier ausgespart.

Der Cortex-M55 Kern in HP- und HE-Bereich ist mit Helium-Erweiterung und Unterstützung für Integer- und Fließkomma-Operationen ausgestattet. Jeder Kern hat 32 KB an Daten- und Instruktionscache. An beiden Kernen ist jeweils ein Daten-TCM und ein Instruktions-TCM angebunden. Die Bereiche unterscheiden sich v.a. in der Taktung der CPU. So taktet der HP-Bereich mit 400 MHz während der HE-Bereich mit 160 MHz taktet. Außerdem ist der TCM-Speicher des HP-Bereiches größer und umfasst 256 KB für den Instruktions- und 1024 KB für den Daten-TCM. Der TCM-Speicher des HE-Bereich

Test	GFLOPS
Scalar FP32	0.752
Scalar FP64	0.033
Helium FP32	1.599

Tabelle 4: Peak Performance HP-Core

Test	GFLOPS
Scalar FP32	0.301
Scalar FP64	0.013
Helium FP32	0.639

Tabelle 5: Peak Performance HE-Core

umfasst nur jeweils 256 KB für Instruktions- und Daten-TCM (siehe 3.2).[18]

Da der Cortex-M55 ein Dual-Beat-System ist, benötigt er zwei Zyklen, um eine Vektorinstruktion, die auf einem 128-Bit-Vektorregister arbeitet, auszuführen. Eine einzelne VFMA-Instruktion mit Datentyp FP32 führt auf vier Elementen jeweils eine Multiplikation und eine Addition durch, d.h. insgesamt acht Fließkomma-Operationen (FLOP). Die theoretische maximale Leistung ergibt sich also folgendermaßen für HP- und HE-Bereich:

$$8 \text{ FLOP} * 400 \text{ MHz} * \frac{1}{2} = 1600 \text{ MFLOPS} = 1.6 \text{ GFLOPS} \quad \text{(HP)}$$

$$8 \text{ FLOP} * 160 \text{ MHz} * \frac{1}{2} = 640 \text{ MFLOPS} = 0.64 \text{ GFLOPS} \quad \text{(HE)}$$

Durch einen Microbenchmark von aneinandergereihten VFMA-Instruktionen kann die theoretisch erreichbare Leistung auch praktisch erreicht werden. Dabei ist es unerheblich, ob zwischen den einzelnen Instruktionen Abhängigkeiten bestehen; die Leistung bleibt gleich. Außerdem wird getestet, inwiefern die theoretisch erreichbare Leistung erhalten bleibt, wenn die VFMA-Instruktionen mit VLDR-Instruktionen vermischt werden. Auch hier bleibt die Performance erhalten, insofern die Daten im TCM liegen, weil VLDR und VFMA verschiedene Pipelines nutzen und sich so überlagern können, während bei einer Aneinanderreihung von VFMA-Instruktionen nur die FP-Pipeline genutzt würde und sich keine Instruktionen überlagern.

Wie in Tabelle 4 zu sehen ist, wird die Peak Performance annähernd erreicht. Im Vergleich zur skalaren Variante der früheren Floating-Point-Erweiterung ist ein zweifacher Speedup zu beobachten. Zudem ist zu sehen, dass die FP- und Helium-Erweiterung nicht auf das schnelle Rechnen mit doppelter Präzision ausgelegt sind, da FP64 nicht von Helium unterstützt wird und bei der FP-Erweiterung nur einen Durchsatz von $\frac{1}{23}$ hat.

Das gleiche Verhalten ist bei Ausführung auf dem HE-Kern zu sehen, wo ebenfalls die jeweilige Peak-Performance erreicht wird.

3.2. Speichersystem

Wie bereits im Abschnitt über die M-Profil-Prozessoren beschrieben, wird grundsätzlich auf zwei verschiedene Speichersysteme gesetzt: Den schnellen, direkt an die CPU angebundenen TCM-Speicher, sowie den langsameren, latenzbehafteten Hauptspeicher, der Caches nutzen kann. Auch auf dem Ensemble E7 hat man eine solche Speicherstruktur. Außerdem ist ein nichtflüchtiger MRAM-Speicher zum dauerhaften Speichern des Programmcodes vorhanden.

Insgesamt stehen 13.5 MB Speicher zur Verfügung, wovon 5.5 MB für den MRAM zur Verfügung stehen. Der SRAM-Bereich ist in einzelne Teile zerlegt, von denen einige für den TCM zur Verfügung stehen.

Mittels Linker-Skript werden die einzelnen Speicherbereiche konfiguriert. Standardmäßig werden alle Variablen im Daten-TCM abgelegt, sowie alle Instruktionen beim Bootprozess in den Instruktions-TCM kopiert. Es sind 8 KB Heap und 16 KB Stack vorkonfiguriert.

[18] Vgl. Alif Semiconductors, Datasheet Ensemble Family E7 Series v2.11, S. 12ff.

Der geringe dynamische Speicher und die verschiedenen Speichersegmente bedeuten, dass es sehr schnell zu Problemen kommen kann, wenn dynamische Speicherverwaltung genutzt wird. Um den Speicherort und damit das Speichersegment zu kontrollieren, werden Größe und Ort von Arrays hier immer bereits vor der Kompilierung festgelegt.

Über die Memory Protection Unit (MPU) kann der Cache konfiguriert werden. Standardmäßig werden beim SRAM0 Lesezugriffe gecached und Schreibzugriffe immer direkt zurückgeschrieben, so dass der Cache umgangen wird.

3.2.1. Dauerhafter Speicher – MRAM

Der MRAM auf dem Ensemble E7 umfasst 5.5 MB auf dem alle Daten untergebracht werden müssen, die auch nach Stromverlust noch vorhanden sein sollen. Dies umfasst in erster Linie den Programmcode. Um den Programmcode schnell auszuführen, wird er mittels Linkerskript beim Boot in den Instruktions-TCM kopiert. Der MRAM ist über den AXI Bus mit 64 Bit angebunden, taktet mit 33 MHz und gehört zur HP-Region.[19] Laut Alif sind Schreibraten von 2.28 MB/s und Leseraten von 232 MB/s erreichbar.[20] Da der Programmcode nachfolgend immer auf den TCM kopiert wird und der MRAM somit, außer zum nichtvolatilen Speichern unwichtig ist, werden diese Angaben hier nicht überprüft und der MRAM im Folgenden ignoriert.

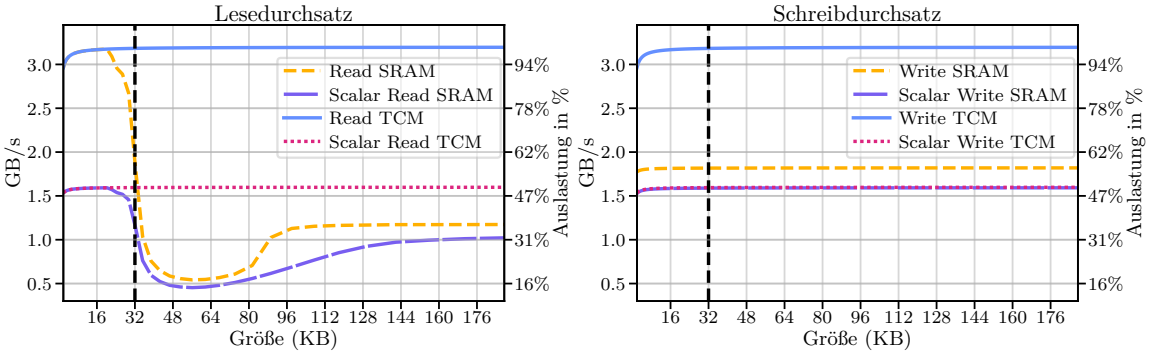
3.2.2. Langsamer Speicher - SRAM

Der SRAM ist auf dem Ensemble E7 als Hauptspeicher gedacht und in mehrere Teile unterteilt, die verschiedene Zwecke und Taktraten haben und auch unterschiedlich angebunden sind. Insgesamt sind 13.5 MB SRAM verfügbar. Es sind SRAM-Teile 0 bis 9 verfügbar, wobei SRAM2 bis SRAM5 als TCM-Speicher angebunden sind (siehe 3.2.3). SRAM0 und SRAM1 sind am HP-Bereich über einen 64-Bit-AXI-Bus mit 400 MHz angebunden, während SRAM6 bis SRAM9 am HE-Bereich angebunden sind. SRAM0 und SRAM1 haben eine Kapazität von 4 MB bzw. 2.5 MB und sollen als gemeinsamer Hauptspeicher für Anwendungen dienen.[21] Die anderen SRAM-Bereiche sind für nicht performance-kritische Anwendungen gedacht, werden aber hier nicht weiter betrachtet. Aufgrund der vergleichsweise großen Kapazität wird für die folgenden Tests immer der SRAM0-Bereich genutzt.

Alif zufolge sind Leseraten von 1350 MB/s (558 MB/s im HE-Bereich) auf SRAM0 möglich. Schreibraten werden mit 1824 MB/s (731 MB/s) angegeben. Außerdem werden bei SRAM0 die Daten- und Instruktionscaches genutzt, so dass bei Wiedernutzung von Daten die Lese- und Schreibraten erheblich höher sein können. Der verbaute Cortex-M55 unterstützt zwei 32-Bit-Zugriffe pro Zyklus, so dass eine maximale Bandbreite von 3.2 GB/s (1,28 GB/s) erreicht werden kann.

Um zu prüfen, welcher Durchsatz praktisch zu erreichen ist, wurden mittels Low-Overhead-Loop mehrere Lade- bzw. Schreibinstruktionen hintereinandergehangen, um ein Array aus FP32-Werten mehrfach zu laden und zusätzlich der Unterschied zwischen Vektor- und Skalar-Ladeinstruktionen getestet. Abbildung 11 zeigt das Verhalten bei verschiedenen Problemgrößen. In Abbildung 11b ist zu sehen, dass die Bandbreite beim Schreiben immer erreicht wird. Zudem existiert kein Unterschied in der Schreibrate bei skalaren Instruktionen für TCM und SRAM. Allerdings wird die Bandbreite beim Lesen nicht erreicht, wie Abbildung 11a zeigt. Vielmehr ist ein kompletter Einbruch der Leserate zu

[19] Vgl. Alif Semiconductors, Datasheet Ensemble Family E7 Series v2.11, S. 23.
[20] Vgl. Alif Semiconductors, Datasheet Ensemble Family E7 Series v2.11, Table 5-14, S. 137.
[21] Vgl. Alif Semiconductors, Datasheet Ensemble Family E7 Series v2.11, S. 36.



(a) Lesedurchsatz für TCM und SRAM0 (b) Schreibdurchsatz für TCM und SRAM0

Abbildung 11: Test des Lese- und Schreibdurchsatz auf TCM und SRAM0. L1-Cache als schwarze Linie eingezeichnet.

sehen, sobald die Problemgröße über den L1-Cache hinauswächst. Danach werden die angegebenen 1350 MB/s auch nicht erreicht, sondern nur 1176 MB/s. Weitere Tests wurden mit verschiedenen Cache-Attributen durchgeführt, welche allerdings keine Verbesserung bei den Leseraten gebracht haben. Außerdem ist zu sehen, dass das Laden mittels Helium-Instruktionen immer schneller ist, insbesondere aber in Fällen, wo ein hoher Durchsatz erreicht werden kann, da dort die volle Bandbreite von 64 Bit ausgeschöpft werden kann. Nicht abgebildet ist der Durchsatz auf dem HE-Kern, da dort das gleiche Verhalten auftritt.

3.2.3. Schneller Speicher - Tightly Coupled Memory

Der TCM-Speicher ist ein nah an die CPU angebundener Teil des SRAM-Speichers, der als besonders schneller Speicher fungiert. Dafür wird der TCM auf dem Ensemble E7 mit der Frequenz getaktet, mit der auch die angrenzende CPU getaktet ist. Für den HP-Kern bedeutet dies eine Frequenz von 400 MHz und für den HE-Kern eine Frequenz von 160 MHz. Dabei sind auf dem HP-Core der SRAM2-Bereich als Instruktions-TCM (ITCM) und SRAM3 als Daten-TCM (DTCM) eingebunden, wobei SRAM2 256 KB und SRAM3 1024 KB groß sind. Auf dem HE-Core sind beide SRAM-Bereiche 256 KB groß. Für den ITCM gibt es jeweils eine 32-Bit-Schnittstelle, die es unterstützt, pro Zyklus eine Instruktion zu laden, während es für den Zugriff auf den DTCM jeweils vier Schnittstellen D0TCM bis D3TCM gibt (siehe Abbildung 2 in 2.1.1).[22] Jede dieser vier Schnittstellen greift dabei mittels Adressfilterung nur auf bestimmte Speicherbereiche zu. Anhand von Bit 2 und 3 der Speicheradresse wird entschieden, welche Schnittstelle genutzt wird. Dabei wird für 0b00 die D0TCM-, für 0b01 die D1TCM-, für 0b10 die D2TCM- und für 0b11 die D3TCM-Schnittstelle genutzt. Diese Auswahl sorgt dafür, wie Abbildung 12 zeigt, dass bei FP32-Werten für jeden nachfolgenden Wert eine andere Schnittstelle genutzt wird. Dadurch kann die maximale Bandbreite von 64 Bit beim Nutzen von Vektor-Ladeinstruktionen, die 128 Bit innerhalb von zwei Zyklen laden, theoretisch erreicht werden.

Von Alif wird angegeben, dass eine Bandbreite von 3200 MB/s (1280 MB/s) erreicht werden kann.[23] Mit dem Microbenchmark aus 3.2.2 wird dies überprüft. Wie in Abbildung 11 zu sehen ist, wird die angegebene Bandbreite sowohl beim Lesen als auch beim Schreiben für HP- und HE-System erreicht.

[22] Arm Ltd., Cortex-M55 Technical Reference Manual, S. 193
[23] Vgl. Alif Semiconductors, Datasheet Ensemble Family E7 Series v2.11, Table 5-15, S. 137.

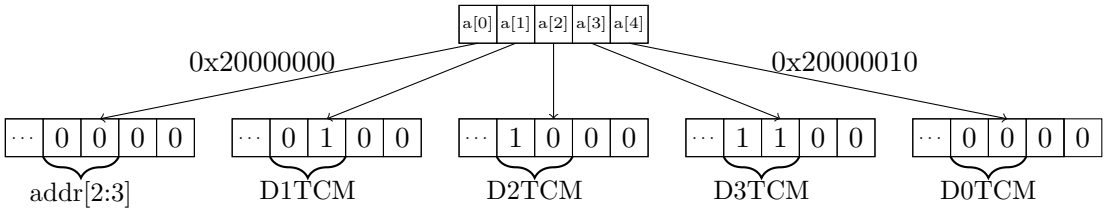


Abbildung 12: Nutzung verschiedener Schnittstellen im D-TCM beim Laden von FP32-Werten. Unten abgebildet sind die letzten vier Bit der zu den Werten gehörenden Speicheradressen.

Wie bereits geschrieben, wird bei Nutzung des TCMs der L1-Cache der CPU umgangen. Wie zu sehen ist, ist dieser auch gar nicht nötig, da der Durchsatz so hoch ist wie beim Cache. Der TCM ist somit optimal für performance-kritische Daten und Programmcode.

4. Just-In-Time-Kompilierung für Matrixmultiplikation

Um die Matrixmultiplikation für verschiedene Größen optimal auszuführen, soll der Matrixkernel programmatisch für eine angegebene Größe erstellt werden. Dazu ist es nötig, dass man ihn „Just in time“ (JIT) generiert. Dafür werden Assembler-Instruktionen in einen Speicherbereich abgelegt, ein Zeiger auf den Speicherbereich abgespeichert und der Zeiger in eine Funktion konvertiert. Will man die Matrixmultiplikation ausführen, ruft man die Funktion an diesem Zeiger auf. Dies ist mit verschiedenen Herausforderungen verbunden, denn einerseits braucht man einen Speicherbereich, in dem die Instruktionen abgelegt werden können. Dieser Speicher muss entweder schnell sein oder mit Caches angebunden, so dass die Instruktionen in den Instruktions-Cache gelegt werden können. Ist das der Fall, muss die Anzahl der Instruktionen außerdem klein genug sein, damit sie in den Instruktions-Cache passen, da sonst performance-beeinträchtigende Cache-Misses auftreten. Außerdem müssen Berechtigungen gesetzt werden, damit auch Programm-Code aus diesem Speicherbereich ausgeführt werden kann. Für Embedded Systeme mit Armv8.1M-Architektur bestehen außerdem weitere Probleme, die im Folgenden beschrieben werden.

4.1. Just-In-Time-Kompilierung auf Cortex-M55

Um JIT kompilierten Code auszuführen, benötigt man einen Speicherbereich, aus dem der Code ausgeführt werden kann. Auf modernen Linux-Systemen existiert dafür die Funktion `mmap` mit der man sich einen Speicherbereich zusichern und die benötigte Schreibbberechtigung auf ihn setzen kann. Mit der Funktion `mprotect` kann im Anschluss die Ausführ-Berechtigung gesetzt werden, nachdem die Instruktionen in den Speicherbereich kopiert wurden. Auf Embedded Hardware wie dem Ensemble E7 sieht die Situation allerdings anders aus: Hier gibt es keine Speicherverwaltungseinheit, die virtuellen Speicher verwaltet und kein Betriebssystem, mit dem man Speicher allozieren kann. Jeglicher Speicherzugriff erfolgt direkt auf den Speicher und die Konfiguration der Berechtigungen erfolgt über das Linker-Skript. Im Standard-Linkerskript für den Ensemble E7 sind alle Speicherbereiche bereits mit Schreib- und Ausführberechtigung konfiguriert. Dies bedeutet, dass man seine Instruktionen in ein Array schreiben kann und die Instruktionen durch Aufruf des Zeigers ausführen kann. Durch das Linkerskript kann außerdem bestimmt werden, in welchem Speichersegment der Code gespeichert werden soll, wobei der SRAM0 mit zusätzlichen L1-Cache sowie der TCM-Speicher zur Verfügung stehen. Zusätzlich zu den Standardseg-

menten werden für die JIT-Kompilierung die Segmente `itcm_jit` im ITCM und `sram0_jit` im SRAM0 im Linkerskript angelegt.

Wie aus den vorigen Abschnitten hervorgegangen, ist die Nutzung des TCM empfehlenswert, insofern genug Speicherplatz zur Verfügung steht. Dabei ist zu beachten, dass der Code, da in einem Array gespeichert, standardmäßig im Daten-TCM abgelegt wird. Dieser wird zwar mit vier Schnittstellen angebunden, so dass eine 32-Bit-Instruktion und zwei 32-Bit-Werte geladen werden können, da aber die Schnittstellen jeweils nur bestimmte Adressen bedienen, kann es bei Übereinstimmung von Bit 2 und 3 der Adresse der Instruktion und Daten dazu kommen, dass eine Schnittstelle zwei Werte laden müsste, was zwangsläufig zu Verzögerungen führt. Also ist es nötig, den Code im Instruktions-TCM abzulegen. Dafür wird ein eigenes Segment `itcm_jit` im Linkerskript konfiguriert. Wie auf Linux-Systemen ist es auch hier notwendig, nach dem Generieren von JIT-Code die Caches und die Instruktions-Pipeline zu leeren, da sonst beim Ausführen der Funktion möglicherweise veralteter Code ausgeführt wird, der sich noch im Cache befindet.

Wie in 2.2.3 beschrieben, ist es für optimale Performance erforderlich, die Helium-Instruktionen im Low-Overhead-Loop auf 4 Byte auszurichten. Da die Direktive `.p2align` bei JIT-Kompilierung nicht zur Verfügung steht, muss manuell vor jeder Schleife, in der dies benötigt wird, ein zusätzliches NOP eingefügt werden. Dies wird über eine Hilfsfunktion geregelt, die die Ausrichtung der jeweiligen Instruktion prüft und, wenn nötig, das NOP einfügt. Mittels Compiler-Direktiven kann sowohl die Ausrichtung der ersten Instruktion als auch das Speichersegment sichergestellt werden. Dadurch wird das Linkerskript angewiesen, den Codebuffer im jeweiligen Segment anzulegen und das erste Element auf 4 Byte auszurichten.

```
uint16_t buffer[INSTR_COUNT] __attribute__((section(".itcm_jit"), aligned(4)));
```

Zuletzt muss beachtet werden, dass beim Funktionsaufruf, d.h. das Springen zum Zeiger auf den Instruktionsspeicher, das Least-Significant-Bit (LSB) der Adresse auf Eins gesetzt werden muss, da die BLX-Instruktion, die für den Sprung zur Funktion zuständig ist, anhand dieses Bits entscheidet, in welchem Modus der Prozessor ab nun arbeitet. Von Armv8.1-M wird dabei der ARM32-Modus (LSB auf 0) und der Thumb-Modus (LSB auf 1) unterstützt. Der Cortex-M55 unterstützt nur den Thumb-Modus, womit das LSB auf Eins gesetzt werden muss, da es sonst zu einem Fehler kommt.[24] Auf die Adresse hat das LSB keine Einwirkung, da vom Prozessor das LSB automatisch auf Null gesetzt wird, wenn zur Adresse gesprungen wird.[25]

Tabelle 6 zeigt die Ergebnisse des Microbenchmarks zur Ermittlung der Peak-Performance nun ausgeführt mittels eines JIT-Kernels. Die Peak Performance kann, wie zu sehen ist, auch mittels JIT-kompilierten Code erreicht werden, egal in welchem Speichersegment der Code liegt. Zu Probleme kommt es erst, wenn SRAM0 genutzt wird und die benötigten Instruktionen größer als der Instruktions-Cache ist.

Test	FLOPS
Buffer im ITCM	1.599
Buffer im DTCM	1.599
Buffer im SRAM0	1.599

Tabelle 6: Peak Performance JIT

[24] Vgl. Arm Ltd., Cortex-M55 Generic User Guide, S. 92.
[25] Vgl. Arm Ltd., Armv8-M Reference Manual, E2.1.33, S. 1937.

Die richtige Wahl des Speichersegments ist insbesondere wichtig, wenn gleichzeitig Daten geladen werden. Tabelle 7 zeigt den Durchsatz beim Laden eines Arrays, welches im DTCM liegt. Hier kommt es, wie oben beschrieben, zu Konflikten mit den TCM-Schnittstellen, wodurch die Ausführung aus dem DTCM-Segment langsamer ist.

Test	GB/s
Durchsatz JIT ITCM	3.1904
Durchsatz JIT DTCM	2.986
Durchsatz JIT SRAM0	3.191

Tabelle 7: Nutzung von eigenem Speicherbereich für JIT-Code

4.2. Umsetzung und Schnittstelle

Der JIT-Compiler muss es erlauben, Instruktionen zum Buffer hinzuzufügen, sowie Hilfsmethoden für den Programmfluss bereitstellen. Außerdem werden Methoden zum Enkodieren der verschiedensten Instruktionen benötigt. Dieser Abschnitt zeigt die Schnittstelle des JIT-Compilers, sowie die Verwaltung des Buffers. Abschnitt 4.3 beschreibt die Enkodierung von Instruktionen während Abschnitt 4.4 die Umsetzung von Branchinginstruktionen und Schleifen zeigt.

Das Backend ist als eigene Klasse konzipiert, der bei Instanziierung ein globaler Buffer zusammen mit seiner Größe übergeben werden muss. Es ist nicht möglich, einen als privat deklarierten Buffer in der Klasse zu nutzen, da dort keine Compilerattribute für das Setzen des Segments angegeben werden können. Das Backend wird von Generator-Klassen genutzt, von denen jede einen oder mehrere Kernel erzeugen. Für die Matrixmultiplikation wird dafür der GEMM-Generator genutzt; für die vorangestellten Performancetests in 4.1 wurden eigene Generatoren genutzt. Der Generatorklasse wird ebenfalls der globale Buffer übergeben. Die Generator-Klasse kann dann folgendermaßen instanziiert werden:

```
JIT::Generators::Gemm gemmGen(buffer, INSTR_COUNT);
```

Jede Generator-Klasse implementiert mindestens eine Funktion `generate`, die einen Kernel generiert. Außerdem wird ein Typalias `Func` für die Funktion erstellt. In der `generate`-Funktion wird zum Schluss der Pointer auf die erste Instruktion zurückgegeben und zur Funktion konvertiert. Mit `getThumbAddress` wird im Backend die Adresse der ersten Instruktion angepasst, damit beim Branch auch der Thumb-Modus genutzt wird.

```
// Definition des Funktionstyps. Übergeben werden A, B und C
using Func = void (*) (float const *, float const *, float *);

// Rückgabe des Pointers als Funktion
return reinterpret_cast<Func>(backend.getThumbAddress());

return reinterpret_cast<uintptr_t>(instructions) | 0x1U; // Setzen des LSB für Thumb-Modus
```

Die `generate`-Funktion des Gemm-Generators unterstützt die Angabe der Matrixgrößen sowie ihrer „leading dimensions“ lda, ldb und ldc, d.h. die Anzahl der Elemente die für A, B bzw. C übersprungen werden müssen, um zur nächsten Reihe zu springen. Die Funktion kann dann folgendermaßen genutzt werden:

```
// Buffer der Größe INSTR_COUNT übergeben
JIT::Generators::Gemm generator(buffer, INSTR_COUNT);
JIT::Generators::Gemm::Func testFunc = generator.generate(m, k, n, lda, ldb, ldc);
testFunc(A, B, C);
```

Es existieren Methoden, um Instruktionen zum Code-Buffer hinzuzufügen, außerdem existiert eine Methode, um eine Helium-Instruktion hinzuzufügen und korrekt im Speicher anzuordnen. Dazu wird beim Einfügen eine zusätzliche NOP-Instruktion eingefügt, falls die Instruktion ansonsten nicht auf 4 Byte ausgerichtet ist. Da auch 16-Bit-Instruktionen unterstützt werden, besteht der Code-Buffer aus uint16_t-Werten, wobei ein Wert eine 16-Bit-Instruktion ist. Soll eine 32-Bit-Instruktion hinzugefügt werden, werden zwei 16-Bit-Werte hintereinander geschrieben, da die CPU anhand der ersten fünf Bits der Instruktion erkennt, ob es eine 32-Bit-Instruktion ist und dann gegebenfalls die nächsten 16 Bit aus dem Speicher lädt.[26]

4.3. Instruktionsenkodierung für Matrixmultiplikation

Es werden alle benötigten Instruktionen für Matrixmultiplikation implementiert. Dazu gehören mehrere Helium-Instruktionen, aber auch einige skalare Instruktionen, sowie Branchinstruktionen und Lade-/Speicherinstruktionen. Dabei sind für ein Großteil der Instruktionen sowohl die 16-Bit- also auch die 32-Bit-Varianten implementiert. Die übergebenen Parameter werden auf Richtigkeit geprüft; werden nicht unterstützte Parameter angegeben, wird eine Fehlermeldung ausgegeben und stattdessen ein NOP zurückgegeben. Somit kann geprüft werden, an welcher Stelle die falsche Angabe gemacht wurde. Es werden, wenn nötig, Immediate Constants[27] unterstützt. Bei der Umsetzung der verschiedenen Instruktionsenkodierungen wird sich nach den Bedürfnissen bei der Matrixmultiplikation gerichtet, sodass nicht alle Enkodierungen einer Instruktion unterstützt werden, sondern nur diese, die auch benötigt werden.

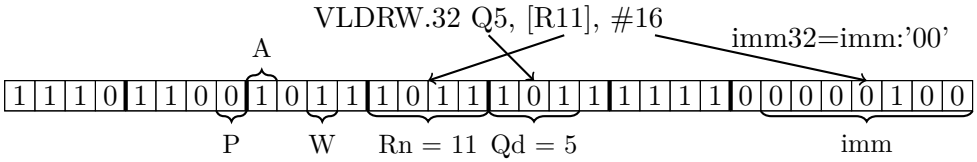


Abbildung 13: Enkodierung der VLDR-Instruktion

Abbildung 13 zeigt, wie die VLDR-Instruktion enkodiert wird, indem der Wert für beide Register, sowie das Immediate und weitere Optionen gesetzt werden. GP-Register und Vektorregister sind wie in den meisten anderen Instruktionen selbsterklärend und einfach die Zahl des Registers. Um größere Abstände zu enkodieren und nur ausgerichtete Zugriffe zu erlauben, werden in der VLDRW-Instruktion Immediates mit einem Left-Shift von Zwei enkodiert, so dass der Offset von 16 Bytes als 4 Byte enkodiert wird. Außerdem ist der Offset als positiver Post-Index angegeben, d.h. er wird nach Laden des Vektors darauf addiert und der resultierende Wert in das GP-Register zurückgeschrieben. Daher sind die Bits für „Add Immediate“ (A) und „Write-Back“ (W) auf Eins gesetzt, während das Bit für „Pre-Index“ (P) auf Null gesetzt ist.[28] In Übereinstimmung mit dem Referenzdokument für die Armv8-M-Architektur werden die weiteren Instruktionen enkodiert.

Maskierung wird unterstützt, wobei zu beachten ist, dass sich bei Maskierung der Bitcode für Instruktionen nicht ändert, was bedeutet, dass nur die zwei Instruktionen VPST und VCTP implementiert werden müssen. Zusätzlich sind zwei Helfermethoden `predicateNextInstructions` und `insertPredicatedInstruction` zum sicheren Einfügen von maskierten Instruktionen vorhanden.

[26] Vgl. Arm Ltd., Armv8-M Reference Manual, E2.1.137, S. 1980f.
[27] Vgl. Arm Ltd., Armv8-M Reference Manual, S. 473.
[28] Vgl. Arm Ltd., Armv8-M Reference Manual, C2.4.365 T7, S. 1165f.

4.4. Branching und Schleifen

Bei handgeschriebenen Assembly-Code können Label genutzt werden, um einzelne Instruktionen zu markieren, damit mithilfe einer Branchinginstruktion zu dieser Instruktion gesprungen werden kann. Die Branchinstruktion enkodiert dann den Abstand zu diesem Label. Im Assembly-Code muss nur das entsprechende Label angegeben werden und der Assembler ermittelt automatisch den korrekten Abstand zur jeweiligen Instruktion und enkodiert damit die Branchinstruktion. Diese Möglichkeit steht bei JIT-Kompilierung nicht zur Verfügung und der Abstand muss vom JIT-Compiler ermittelt und enkodiert werden. Dabei müssen folgende Anwendungsfälle berücksichtigt werden: (1) Rückwärts-Branch mit und ohne Bedingung, (2) Vorwärts-Branch mit und ohne Bedingung und (3) Low-Overhead-Loops.

Um den korrekten Abstand zwischen der Zielinstruktion und der Branchinstruktion zu ermitteln, wird die Adresse der Zielinstruktion von der Adresse der Branchinstruktion subtrahiert sowie weitere 4 Bytes abgezogen, da der Program-Counter bei Branchinstruktionen der aktuellen Adresse plus 4 Bytes entspricht.[29] Weiterhin ist es nötig, die Adresse der Zielinstruktion zu speichern, analog zu einem Label. Dafür ist die Funktion `addBranchTargetInstruction()` implementiert, die eine Instruktion zum Buffer hinzufügt und deren Adresse zurückgibt.

Für den Rückwärtsbranch wird eine eigene Funktion `addBackwardsBranchFromCurrentPosition` implementiert, die mittels der übergebenen Adresse den Abstand ermittelt und die Branchinstruktion enkodiert. Der Low-Overhead-Loop ist eine Form des Rückwärtsbranch und wird mit der Funktion `addLowOverheadBranchFromCurrentPosition()` gesetzt.

```
// Zielinstruktion
Instructions::Instruction16 * start = backend.addBranchTargetInstruction(instr);
...
// Rückwärtsbranch
backend.addBackwardsBranchFromCurrentPosition(start, Condition::LT);
// Alternative: Low-Overhead-Loop
backend.addLowOverheadBranchFromCurrentPosition(start);
```

Listing 1: Rückwärtsbranch

Beim Vorwärtsbranch wird zuerst die Branchinstruktion eingefügt, wobei die Adresse der Zielinstruktion noch nicht bekannt ist. Dieses Problem wird gelöst, indem mittels Hilfsfunktion ein NOP an der Stelle der Branchinstruktion eingefügt wird und zuerst diese Adresse gespeichert wird. Ist man an der Zielinstruktion angelangt, wird diese eingefügt und ihre Adresse gespeichert. Mittels der Funktion `setForwardsBranch` wird die NOP-Instruktion mit der nun korrekten Branchinstruktion ersetzt. Der Abstand kann mittels der beiden zwischengespeicherten Adressen ermittelt werden.

```
Instruction16 * branchInstr = backend.addBranchPlaceholder(); // NOP
...
Instruction16 * target = backend.addBranchTargetInstruction(instr); // Zielinstruktion
backend.setForwardsBranch(branchInstr, target, Condition::EQ); // NOP ersetzen
```

Listing 2: Vorwärtsbranch

[29] Vgl. Arm Ltd., Cortex-M55 Generic User Guide, S. 110.

5. Effiziente Matrixmultiplikation

Matrixmultiplikation als grundlegende mathematische Operation ist in Gleichung 1 dargestellt.

$$c_{ij} = c_{ij} + \sum_{k=1}^{n} a_{ik} * b_{kj} \tag{1}$$

Wird diese Definition in C++ nachgebildet, kommt man auf den Algorithmus in Listing 3. Dieser Algorithmus und alle weiteren folgenden interpretieren Matrizen in Spaltenmajor-Anordnung, d.h. durch Inkrementierung des Index wird in den Spalten fortgeschritten.

```
void gemm_naive(const float * a, const float * b, float * c, const uint32_t n, const
    ↪ uint32_t k, const uint32_t m, const uint32_t lda, const uint32_t ldb, const
    ↪ uint32_t ldc) {
    for (uint32_t j = 0; j < n; j++) { // j = n
        for (uint32_t i = 0; i < m; i++) { // i = m
            for (uint32_t p = 0; p < k; p++) { // p = k
                c[j * ldc + i] += a[p * lda + i] * b[j * ldb + p]; // Column Major
            }
        }
    }
}
```

Listing 3: Naive Matrixmultiplikation

Dieser Algorithmus liefert eine äußerst geringe Performance, da alle Elemente von A und B insgesamt k-mal geladen werden müssen. Um eine höhere Leistung zu erzielen, muss versucht werden so viele Elemente wie möglich vorzuhalten, um mit diesen einen größeren Teil der Matrix zu berechnen. Dafür wird von modernen Bibliotheken zur Matrixberechnung wie BLIS [35, 37] auf die grundlegenden Ideen von GotoBLAS [20] zurückgegriffen. Dort werden bei der Matrixmultiplikation einzelne Blöcke der Größe $m_r \times n_r$ abgearbeitet. Somit verringert sich die Anzahl der Zugriffe auf einzelne Elemente enorm. Auf jedes Element muss nur einmal im Microkernel zurückgegriffen werden. Abbildung 14 zeigt die Unterteilung einer 32x16 Matrix in 16x8-Blöcke. Auf die Elemente der ersten 16 Reihen in A muss nun etwa nur noch zweimal zugegriffen werden. Mittels zweier Schleifen um m und n werden dann alle Blöcke abgearbeitet.

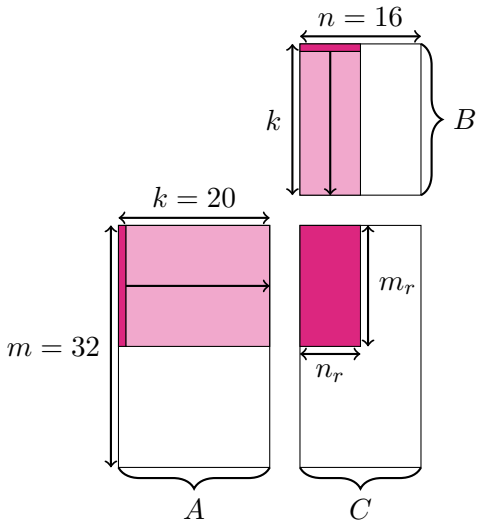


Abbildung 14: 32x16 aufgeteilt in 16x8 Microkernel

Ein einzelner Block wird mittels eines sogenannten „Microkernels“ berechnet. Dieser berechnet k-mal ein äußeres Produkt in Größe des Blocks [26]. In jedem Schritt wird ein neues äußeres Produkt berechnet und auf den Block in C addiert. Gleichung 2 zeigt die Definition des äußeren Produkts, welches im Microkernel auf eine Spalte von A und eine Reihe von B angewandt wird. Der Microkernel wird abhängig von der zugrundliegenden Architektur gestaltet, d.h. je nachdem welche Instruktionen und Möglichkeiten die CPU-Architektur bietet.

$$x \otimes y = x * y^T = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} * (y_1 \cdots y_n) = \begin{pmatrix} x_1y_1 \cdots x_1y_n \\ \vdots \ddots \vdots \\ x_my_1 \cdots x_my_n \end{pmatrix} \tag{2}$$

Bibliotheken müssen anhand der Matrixgröße entscheiden, welche Microkernel eingesetzt werden, um eine korrekte und schnelle Berechnung zu gewährleisten. Dafür existieren mehrere Techniken. Bei BLIS etwa wird ein einziger Microkernel genutzt und in weiteren Schleifen die Matrix in zusammenhängenden Speicher kopiert und mit Nullen aufgefüllt, so dass die Berechnung korrekt bleibt. In OpenBLAS werden mehrere Microkernel implementiert, die die Ränder abarbeiten und zur Laufzeit entschieden, welche dieser Microkernel genutzt werden müssen.

Als besonders performante Alternative dazu hat sich die Just-in-Time-Kompilierung eines Kernels gezeigt, der perfekt auf die Matrixdimensionen angepasst ist, wie in LIBXSMM implementiert [25]. Dabei wird anhand verschiedener Parameter ein Kernel erzeugt, der dann zur Laufzeit nur noch einen geringen Aufwand für den Programmfluss und die Auswahl der Microkernel benötigt, da diese Entscheidungen schon im Vorhinein für die jeweilige Größe getroffen wurden. Dieses Vorgehen ist äußerst effizient für kleine Matrizen, da dort der Aufwand für Programmfluss und Kopieren der Daten vergleichsweise höher ist.

Die hier vorgestellte Implementierung setzt daher auf die Erzeugung des Microkernels mittels JIT-Compiler nach dem Vorbild von LIBXSMM.

5.1. Implementierung des Microkernels für Helium

Um die Matrixmultiplikation zu implementieren, muss zuerst ein Microkernel entwickelt werden, der besonders performant einen Block der Matrix C berechnet.

Die Implementierung des Microkernels für die Helium-Erweiterung bringt aufgrund der begrenzten Anzahl an Registern, der Performanceeigenschaften und Instruktionen der Helium-Erweiterung einige Herausforderungen mit sich, die gelöst werden müssen, um eine hohe Performance zu erzielen.

Zuerst muss die Größe des Microkernels bestimmt werden. Prinzipiell sollte versucht werden, den Microkernel so groß wie möglich zu gestalten, um möglichst viele Elemente in den C-Akkumulator-Registern zu halten. Aufgrund der geringen Anzahl an acht Vektorregistern besteht dabei keine große Auswahl an verschiedenen Varianten. Wie beschrieben, muss m_r ein Vielfaches der Vektorgröße sein, d.h. für Helium ein Vielfaches von Vier. Für B werden keine Vektorregister benötigt, da die VFMA-Instruktion auch mit GP-Register genutzt werden kann. Somit werden insgesamt $n_r * \frac{m_r}{4}$ Vektorregister für C und $\frac{m_r}{4}$ für A benötigt. Für die Wahl von m_r ergeben sich damit drei Möglichkeiten: (1) Ein 4x7 Microkernel, in dem ein Register für A und sieben für C genutzt werden, sowie (2) ein 8x3 Microkernel, in dem zwei Register für A und sechs für C genutzt werden, sowie schlussendlich (3) ein 16x1 Microkernel, in dem für A und C jeweils vier Register genutzt werden.

Der 8x3-Microkernel hat den Vorteil, dass nur drei GP-Register für die Werte von B benötigt werden. Außerdem können zwei Vektor-Ladeinstruktionen pro Zyklus genutzt

werden, was erweiterte Möglichkeiten zur Verschachtelung mit sich bringt. Da der 8x3-Microkernel in den Tests die beste Performance gebracht hat, dient er hier zur Beschreibung des grundlegenden Aufbaus. Die anderen Varianten werden aber implementiert, um Kantenfälle performant abzuarbeiten.

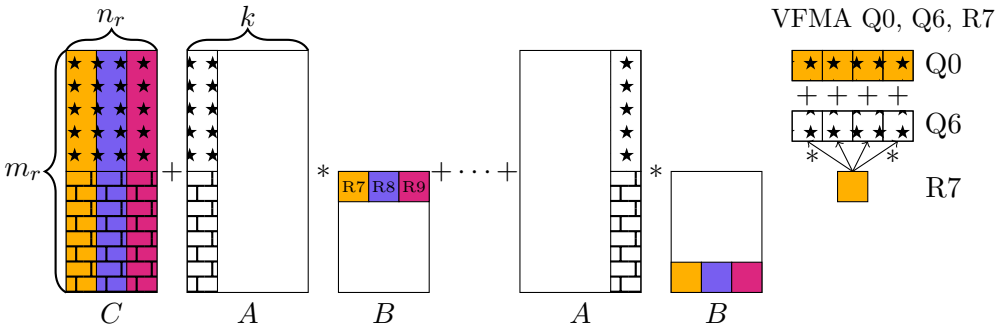


Abbildung 15: 8x3 Microkernel

Der grundsätzliche Aufbau des Microkernels auf Armv8-M ist ähnlich wie bei anderen Vektorarchitekturen wie Neon. Zu Beginn müssen die C-Akkumulator-Register geladen werden. Danach wird mit einer Schleife über k in jeder Iteration das äußere Produkt berechnet und auf C aufaddiert. Abbildung 15 und Listing 4 zeigen die Berechnung des Microkernels. Am Anfang jeder Iteration werden die zwei Vektorregister für A mit den Werten der derzeitigen Spalte - bestimmt von der Iteration der Schleife über k - von A geladen (Zeile 3-4). Danach werden die drei Werte der derzeitigen Reihe von B in drei verschiedene GP-Register geladen (Zeile 5-7). Welcher Bereich in der Matrix abgearbeitet wird, d.h. in welcher Reihe von A und Spalte von B geladen wird, hängt dabei von der aktuellen Position des Microkernels ab, wie oben beschrieben und in Abbildung 14 dargestellt. Nachdem die Elemente geladen sind, kann nun das äußere Produkt berechnet und auf C akkumuliert werden (Zeile 8-13). Die genutzte VFMA-Instruktion mit GP-Register nutzt dabei den Wert im GP-Register wie ein Broadcast-Register, d.h. dass der Wert mit jedem Wert im ersten Quellregister multipliziert wird und der resultierende Vektor auf den Zielvektor addiert wird (siehe Abbildung 15 rechts), was der Definition des äußeren Produkts (siehe Gleichung 2) entspricht. Zum Schluss jeder Iteration muss der Zeiger auf A inkrementiert werden, um zur nächsten Spalte zu gelangen. Der Zeiger auf B wird in jeder Iteration automatisch durch die letzte LDR-Instruktion inkrementiert (Zeile 7), so dass zur nächsten Reihe gesprungen wird.

```
1  dls     lr, r6 // Schleife über k
2  /* 8 Elemente aus A-Spalte laden */
3  sl: vldrw.u32       q6, [r0, #0]
4  vldrw.u32       q7, [r0, #16]
5  ldr.w   r8, [r1, #1*LDB]      // Lade aus Spalte 2 von B
6  ldr.w   r9, [r1, #2*LDB]      // Lade aus Spalte 3 von B
7  ldr.w   r7, [r1], #4          // Lade aus Spalte 1 von B. Springe zu nächster B Reihe
8  vfma.f32        q0, q6, r7    // C[0,0] (oben links)
9  vfma.f32        q2, q6, r8    // C[0,1] (oben mitte)
10 vfma.f32        q4, q6, r9    // C[0,2] (oben rechts)
11 vfma.f32        q1, q7, r7    // C[1,0] (unten links)
12 vfma.f32        q3, q7, r8    // C[1,1] (unten mitte)
13 vfma.f32        q5, q7, r9    // C[1,2] (unten rechts)
14 addw    r0, r0, #LDA          // Springe zu nächster A Spalte
15 le      lr, sl                // Nächste Iteration von k
```

Listing 4: 8x3 Microkernel in Assembly

Der Unterschied zum Microkernel für Neon entspringt neben der begrenzten Anzahl an Registern v.a. aus den unterschiedlichen Varianten der FMA-Instruktion. Auf Helium wird das Nutzen eines GP-Registers als Broadcast-Register unterstützt, was es erlaubt, ein Vektorregister einzusparen. Allerdings ist es auf Helium auch nicht möglich, etwa mehrere Werte in ein Vektorregister zu laden und einzelne Lanes als Broadcast-Register zu interpretieren, wie es bei Neon der Fall ist.

Damit Matrizen berechnet werden können, die größer sind als der Microkernel, muss dieser mehrfach ausgeführt werden. Im einfachsten Fall können dafür zwei Schleifen über m und n durchlaufen werden, was aber für Größen, in denen die Matrixdimensionen kein Vielfaches des Microkernels sind, für Probleme sorgt. Die Behandlung dieser Fälle und die grundlegende Abarbeitung der Matrizen mithilfe des Microkernels wird in 5.2 beschrieben.

Der hier in Listing 4 gezeigte Microkernel berechnet zwar einen Block der Matrix korrekt, liefert aber keine zufriedenstellende Performance.[30] Daher werden in 5.3 die wichtigsten Schritte zur Optimierung des Microkernels gezeigt.

5.2. Abarbeitung der Microkernel

Wenn die Matrix größer als der Microkernel ist, muss dieser mehrfach ausgeführt und jeweils ein bestimmten Teil der Matrix berechnen.

Grundsätzlich müssen nach dem Abarbeiten des Microkernels die Zeiger auf die Matrizen angepasst werden, sodass bei einer erneuten Ausführung nun der nächste Teil der Matrix berechnet wird. Wie in Abbildung 14 gezeigt, muss dabei für A in jeder Iteration von m der Zeiger auf den Start der jeweiligen Reihe des Microkernels gesetzt werden und für B in jeder Iteration von n auf den Start der jeweiligen Spalte. Für C müssen Spalte und Reihe gesetzt werden. Der folgende Code zeigt das Vorgehen in C++, muss aber mittels dem JIT-Compiler in Assembly umgesetzt werden.

```
for (uint32_t j = 0; j < n - (n % nr); j += 3) {
  for (uint32_t i = 0; i < m - (m % mr); i += 8) {
    microkernel_8x3(&A[0][i], &B[j][0], &C[j][i]); // Column Major
  }
}
```

Da keine drei Register geopfert werden können, um die Basisadresse für jede Matrix zu speichern, wird sich immer die Position gemerkt und anhand des Programmflusses der Zeiger angepasst. Einzig für A wird die Basisadresse gespeichert und in jeder Adresse der Zeiger auf die aktuelle Reihe mittels des Schleifenzählers berechnet, weil das Immediate in den meisten Fällen zu groß werden würde. B durchläuft im Microkernel eine gesamte Spalte, so dass nach der Ausführung nur noch zwei Spalten weiter gesprungen werden muss. Bei C wird in jeder Iteration von m nur der Abstand zu den nächsten acht Reihen addiert. Nachdem die m-Schleife durchlaufen ist, muss ebenso wie bei B zwei Spalten weiter gesprungen werden.

Wenn die Dimensionen der C-Matrix nicht ein Vielfaches der Dimension des Microkernels sind, müssen die überschüssigen Elemente gesondert behandelt werden. Dabei kann es den Fall geben, dass entweder die m bzw. n Dimension bzw. beide zusammen kein Vielfaches des Microkernels sind. Bestehen nur in der m-Dimension überschüssige Elemente, wird ein neuer Microkernel generiert und nach dem Durchlaufen einer m-Iteration ausgeführt. Es müssen dabei keine Loop-Strukturen angepasst werden. Bei einer nicht passenden n-Dimension sieht die Situation anders aus: Dort wird eine neue m-Schleife generiert und ein passender Microkernel erzeugt, der dann innerhalb der m-Schleife ausgeführt wird.

[30] Siehe die Auflistung der Optimierungsstufen in Abbildung 19, „Baseline“.

Besteht dann noch ein Überschuss in der m-Dimension, wird wie im normalen Fall ein neuer Microkernel generiert, der genau auf die übrigbleibenden Elemente passt und zum Schluss die letzte Ecke berechnet.

Beim Überschuss der m-Dimension wird im Standardfall entweder ein 8x3- oder ein 4x3-Microkernel mit entsprechender Maskierung generiert. Dies ist aber im Fall von kleinen Überschüssen, d.h. bei einem Microkernel von 1x3-4x3 nicht optimal, da dort die Register nicht voll ausgenutzt werden. Aus diesem Grund wird, wenn möglich, ein (optional maskierter) 4x6-Microkernel erzeugt, der dann nur in jeder zweiten Iteration von n ausgeführt wird. Damit kann die Performance in bestimmten Kantenfällen weiter gesteigert werden.

Abbildung 16 zeigt die Abarbeitung des Microkernels bei Kantenfällen, in denen ein 4x6-Microkernel eingesetzt werden kann. Dabei zeigt der rote Pfeil die m-Schleife, der orangene Pfeil das Fortschreiten der n-Schleife, der blaue Pfeil die Kantenfälle der m-Schleife und der lilafarbene Pfeil den Kantenfall der n-Schleife inkl. eigener m-Schleife und zusätzlichem Kantenfall.

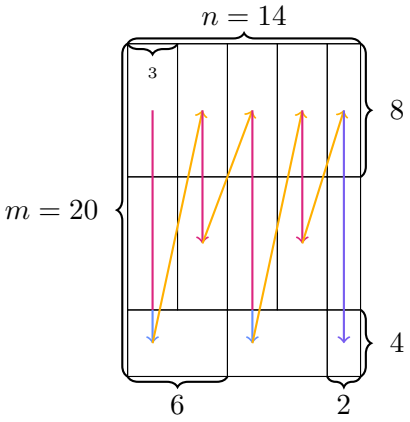


Abbildung 16: Abarbeitung der Kantenfälle ($C \in R^{20x14}$)

Diese Architektur führt also, wie abgebildet, im schlechtesten Fall zur Generierung von vier verschiedenen Microkerneln. Die Nutzung des 4x6-Microkernels ist in Abbildung 17 gezeigt und bringt einen Performancegewinn von durchschnittlich 1% bis maximal 5%, wobei der Effekt v.a. bei kleinen Größen zu sehen ist, wo die Kantenfälle einen größeren Teil der Arbeit ausmachen.

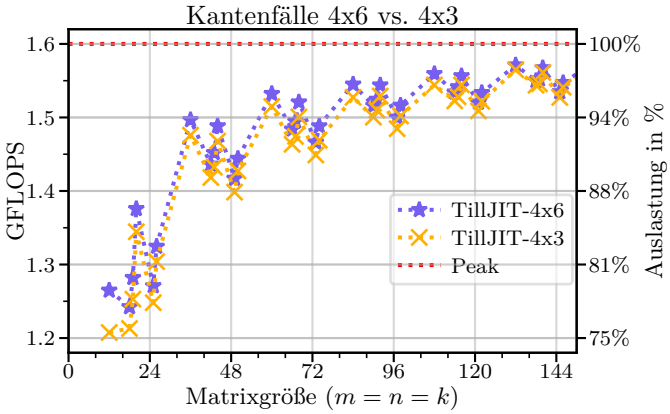


Abbildung 17: Auswirkung des 4x6 Microkernels. Nur Größen gezeigt, bei denen der 4x6 Microkernel eingesetzt wird.

5.3. Optimierungen

Der einfach angeordnete 8x3-Microkernel erreicht nur eine Performance von 0.98 GFLOPS bei einer 24x24-Matrix.[31] Wie bereits beschrieben, sind mit der optimierten Anordnung von Instruktionen sehr große Performancegewinne zu erzielen. Daher zielt ein Großteil der Optimierungen auf die Verschachtelung von Instruktionen ab, so dass durch Überlappung der Instruktionen bei gleicher Anzahl an Instruktionen weniger Zyklen benötigt werden. Um das Verhältnis von Zyklen und Instruktionen zu messen, werden bei jeder Optimierungsvariante die Instruktionen pro Zyklus (IPC) angegeben.[32] Außerdem werden die Zyklen, die die Pipeline wegen unoptimierter Helium-Instruktionen gestockt hat, angegeben.

Wie Tabelle 8 („Baseline“) zeigt, werden nicht einmal 0.7 Instruktionen pro Zyklus ausgeführt und fast jede Helium-Instruktion führt zu einem Stall. Dies führt zu einer Performance von gerade einmal knapp einem GFLOP/s.

Es werden weitere kleinere Optimierungen, wie die Nutzung von 16-Bit-Instruktionen, optimierte Verschachtelung von Instruktionen außerhalb der k-Schleife und optimiertes Branching vorgenommen, die aber im Gegensatz zur Optimierung des Microkernels und dessen k-Schleife nur minimalen Einfluss haben und deshalb nur hier erwähnt werden.

Ein Großteil der Optimierungen profitiert dabei von der Nutzung des JIT-Generators, da anhand der angegebenen Dimensionen entschieden werden kann, welche Optimierung dazu am besten passt. Etwa das Ausrollen von Schleifen und die dynamische Anpassung von Instruktionsvarianten an die jeweilige Größe zur Unterstützung großer Matrizen können so gelöst werden.

5.3.1. Überlagerung der Instruktionen

Erster Ansatzpunkt ist es, die Überlagerung der Instruktionen zu optimieren. Wie in Kapitel 2.2.1 beschrieben, bieten Helium-Instruktionen, die verschiedene Pipelines nutzen, den perfekten Ansatzpunkt dafür. Auch können skalare Instruktionen hinter Helium-Instruktionen überlappt werden. Im Microkernel bieten sich für die Überlappung also die VLDR- und VFMA-Instruktionen an, da diese verschiedene Pipelines nutzen. Werden nun diese Instruktionen verschachtelt und zudem die skalaren Instruktionen hinter den übrigbleibenden VFMA-Instruktionen ausgeführt, führt dies zum Microkernel in Listing 5.

```
dls     lr, r6
sl: ldr.w   r8, [r1, #1*LDB]  // Lade aus Spalte 2 von B
vldrw.u32       q6, [r0, #0]
vfma.f32        q2, q6, r8    // C[0,1] (oben mitte)
vldrw.u32       q7, [r0, #16]
vfma.f32        q3, q7, r8    // C[1,1] (unten mitte)
ldr.w   r9, [r1, #2*LDB]      // Lade aus Spalte 3 von B
vfma.f32        q4, q6, r9    // C[0,2] (oben rechts)
ldr.w   r7, [r1], #4          // Lade aus Spalte 1 von B. Springe zu nächster B Reihe
vfma.f32        q5, q7, r9    // C[1,2] (unten rechts)
addw    r0, r0, #LDA          // Springe zu nächster A Spalte
vfma.f32        q0, q6, r7    // C[0,0] (oben links)
vfma.f32        q1, q7, r7    // C[1,0] (unten links)
le      lr, sl                // Nächste Iteration von k
```

Listing 5: Microkernel mit Überlagerung

[31] In diesem Kapitel haben alle Matrizen A, B, C eine Größe von 24×24

[32] Vgl. Arm Ltd., Armv8.1-M Performance Monitoring User Guide, S. 39.

Dies bringt bereits signifikante Leistungssteigerung mit sich, wie in Tabelle 8 („Interleaving“) gezeigt. Insbesondere die Stallzyklen konnten erheblich verringert und dadurch das IPC-Verhältnis auf 0.82 gesteigert werden. Allerdings ist zu sehen, dass die Überlagerung nicht optimal ist, da u.a. noch zwei VFMA-Instruktionen hintereinander ausgeführt werden, was zu einem Stall in der Pipeline führt, da sich diese beiden Instruktionen nicht überlagern können. Außerdem werden die Lade- und Speicherinstruktionen für C vor und hinter der k-Schleife nicht überlagert.

5.3.2. Loop Peeling in zwei Schritten

Bisher werden alle VLDR- und VSTR-Instruktionen für das Laden und Speichern noch nacheinander ausgeführt, da es keine Instruktionen gibt, mit denen das Laden und Speichern überlagert werden kann. Außerdem ist, wie gesehen, die Überlagerung der Instruktionen noch nicht optimal. Aus diesem Grund wird als nächster Schritt die k-Schleife bearbeitet, so dass die erste und letzte Iteration nicht mehr innerhalb der Schleife, sondern davor und danach berechnet werden und somit mit dem Laden bzw. Speichern von C überlagert werden, was als „Loop Peeling“ bezeichnet wird. Das führt nicht nur dazu, dass im ersten Schritt direkt nach dem Laden eines Wertes von C die Berechnung stattfinden kann, sondern es bieten sich im zweiten Schritt auch bessere Möglichkeiten, die Instruktion in der Schleife zu überlagern.

Denn die Lade- und Speicher-Instruktionen zu überlagern bringt bereits einen Geschwindigkeitszuwachs, die Überlagerung kann aber noch weiter optimiert werden, indem einige Ladeinstruktionen aus der ersten Iteration herausgezogen und innerhalb der Schleife dann am Ende bearbeitet werden können. Damit ist es möglich, speziell die Vektorladeinstruktionen besser zu verschachteln, was zu dem Microkernel in Listing 6 führt.

```
vldrw.u32       q6, [r0, #LDA] // Lade A für nächste Iteration
addw    r0, r0, #LDA           // Nächste Iteration
ldr.w   r8, [r1, #LDB]         // Lade B[LDB] für nächste Iteration
ldr.w   r9, [r1, #2*LDB]       // Lade B[2*LDB] für nächste Iteration
dls     lr, r6                 // Starte k-Schleife
vfma.f32        q2, q6, r8     // VFMA mit vorgeladenen Elementen
ldr.w   r7, [r1], #4           // Lade B[0]
vfma.f32        q4, q6, r9
vldrw.u32       q7, [r0, #16]  // Lade A[3...]
vfma.f32        q0, q6, r7
vldrw.u32       q6, [r0, #LDA] // Lade A für nächste Iteration
vfma.f32        q1, q7, r7
addw    r0, r0, #480           // A anpassen für nächste Iteration
vfma.f32        q3, q7, r8
ldr.w   r8, [r1, #96]          // Lade B[LDB] für nächste Iteration
vfma.f32        q5, q7, r9
ldr.w   r9, [r1, #192]         // Lade B[2*LDB] für nächste Iteration
le      lr, 0x70
```

Listing 6: Microkernel mit Loop Peeling

Jetzt ist zu sehen, dass alle VLDR-Instruktionen perfekt mit den VFMA-Instruktionen verschachtelt sind und keine Stalls in der Pipeline auftreten, wie in Tabelle 8 („Loop Peeling“) zu sehen. Abbildung 18 zeigt, wie die FP-Pipeline vollständig ausgelastet werden kann. Außerdem wird jede LDR- und ADD-Instruktion direkt hinter einer VFMA-Instruktion ausgeführt, so dass auch diese überlagert werden können. Dies führt zu einer signifikanten Verbesserung der Laufzeit und einer Performance von 1.457 GFLOPs sowie einem IPC-Verhältnis von 0.98, was bedeutet, dass in fast jedem Zyklus eine Instruktion ausgeführt werden kann.

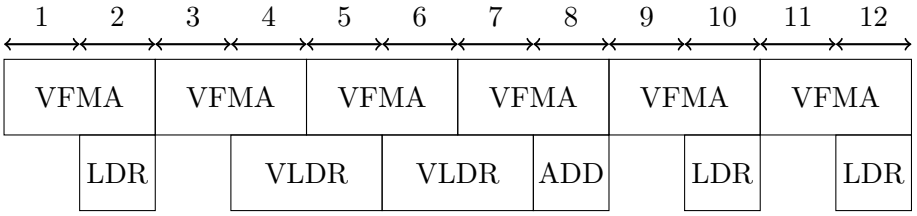


Abbildung 18: Überlagerung der Instruktionen im optimierten Microkernel. In 12 Zyklen werden sechs VFMA-Instruktionen ausgeführt, was dem maximal erreichbaren Durchsatz entspricht.

5.3.3. Ausrollen der Schleifen

Um Branchinstruktionen und dafür notwendige Vergleichinstruktionen einzusparen, können Schleifen ausgerollt, d.h. der Inhalt der Schleife mehrfach kopiert und hintereinander ausgeführt werden. Da aber mit wachsendem k mehr Zeit in der k-Schleife verbracht wird, sinkt auch der Einfluss dieser Optimierung, was sie v.a. bei kleinen Matrixgrößen nützlich macht. Außerdem muss beachtet werden, dass beim Ausrollen mehrerer Schleifen der Speicherbedarf für den Code stark ansteigt.

In der m und n Schleife wird dabei jeweils nur ausgerollt, wenn die gesamte Schleife ersetzt werden kann. Ist dies nicht möglich, d.h. m oder n sind zu groß, werden diese Schleifen auch nicht ausgerollt. Abbildung 19 zeigt den Performanceeinfluss, wenn m und n ausgerollt werden. Dieser ist allerdings minimal, insbesondere für n, da dort die wenigsten Branchinstruktionen wegfallen.

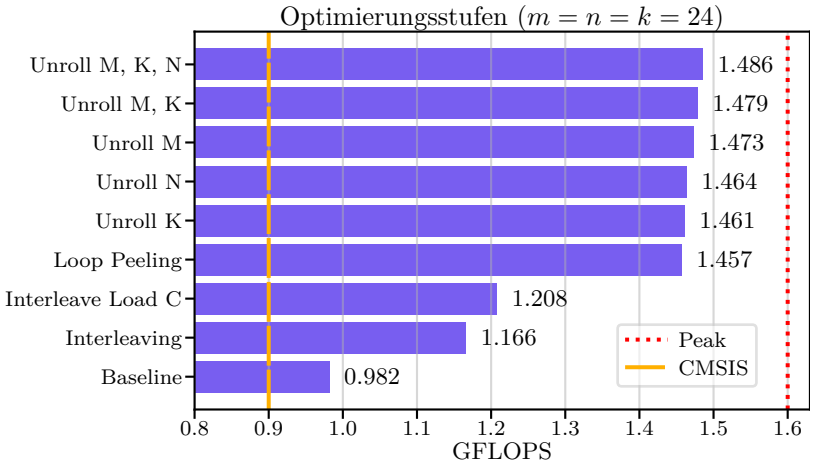


Abbildung 19: Optimierungsstufen des Microkernels bei $k = m = n = 24$

In der k-Schleife wird statt einer traditionellen Schleife eine Low-Overhead-Schleife genutzt, wo der zusätzliche Aufwand extrem gering ist. Hier kann sich allerdings zunutze gemacht werden, dass die Ladeinstruktionen für A nur mit Immediates arbeiten können. Bei kleinen Größen können dabei nicht nur die Werte für die aktuelle Spalte geladen werden, sondern auch der Zugriff auf folgende Spalten gewährleistet werden, da die Immediates klein genug sind. Beim Ausrollen können daher einige ADD-Instruktionen eingespart werden. Da die ADD-Instruktion hinter der VFMA-Instruktion überlagert wird, ist der Performancevorteil allerdings minimal, da beim Ausrollen nun wieder Stalls auftreten. Um Speicher zu sparen, wird daher das Ausrollen der k-Schleife auf fünf Iterationen begrenzt. Im Unterschied zur m und n Schleife wird die k-Schleife nicht anhand der Größe ausgerollt,

sondern basierend darauf, ob ADD-Instruktionen übersprungen werden können.

Der Einfluss auf die Performance ist generell sehr gering, weshalb das Ausrollen nur genutzt werden sollte, wenn auch genügend Speicherplatz für den Codebuffer vorhanden ist.

	Baseline	Interleaving	Loop P.	Unroll k
GFLOPS	0.982	1.166	1.457	1.461
Zeit (ms)	1629	1372	1098	1094
Zyklen	11314	9532	7635	7611
Instruktionen	7517	7517	7493	7061
Helium-Instr.	4896	4896	4896	4896
Helium-Stalls	3695	816	0	408
IPC	0.66	0.79	0.98	0.93

Tabelle 8: Performanceeigenschaften der Optimierungsstufen

Abbildung 19 zeigt die Performance der einzelnen Optimierungsstufen. Es ist zu sehen, dass insbesondere die Optimierung der Instruktionsüberlagerung durch optimiertes Loop Peeling einen großen Einfluss hat. Es ist zu sehen, dass nicht nur darauf geachtet werden muss, Instruktionen zu überlagern, sondern auch die Programmstruktur angepasst werden muss, um optimale Bedingungen für die Überlagerung zu schaffen.

5.4. Anpassungen für große Matrizen

Bisher wurden nur kleine Matrizen betrachtet, bei denen in jeder Instruktion ein Immediate genommen werden konnte. Dieses Vorgehen kommt aber schnell an seine Grenzen, speziell bei dem 4x6-Microkernel, da dort in jeder Iteration über sechs Spalten von B gesprungen werden muss. Auch bei dem 8x3-Microkernel kommt es bei Nutzung von Immediates zu Problemen bei größeren Matrizen, was seinen Grund darin hat, dass in den Vektorinstruktionen nur relativ kleine Immediates enkodiert werden können. So lässt die VLDR-Instruktion nur ein Immediate von bis zu 508 Bytes zu, was bedeutet, dass A nicht mehr mit Immediates geladen werden kann, sobald die Anzahl der Zeilen größer als 127 ist. Wie bei A gibt es auch Beschränkungen für B und C, ab denen keine Immediates genutzt werden können. So kann ab einer Größe von 512 bei B kein Immediate mehr genommen werden, wobei hier die Zahl größer ist, da eine skalare LDR-Instruktion genutzt wird in welcher ein größeres Immediate als in der VLDR-Instruktion enkodiert werden kann. Bei C kann bereits ab einer Zeilenanzahl von 64 kein Immediate mehr genutzt werden.

Aus diesem Grund muss in bestimmten Fällen auf Immediates verzichtet und stattdessen auf die Varianten der Instruktionen mit Register-Offset zurückgegriffen werden. Das grundsätzliche Problem dabei ist, dass nur eine begrenzte Zahl an Registern zur Verfügung steht. Es werden Register für das Speichern der Schleifenzähler, der Basisadressen der Matrizen und die Werte von B benötigt. Da nur insgesamt 16 Register zur Verfügung stehen und zusätzlich bereits der Stack-Pointer und Program-Counter nicht genutzt werden können, bleiben genau drei Register übrig, um alle Szenarien für große Matrizen abzudecken. Bei der VLDR-Instruktion stellt sich das zusätzliche Problem, dass hier kein Offset mittels Register vorgesehen ist und somit nur mittels Immediate gearbeitet werden kann, was schlussendlich bedeutet, dass die ADD-Instruktion nicht mehr hinter der VLDR-Instruktion ausgeführt werden kann, sondern davor.

Die drei Register reichen nicht aus, um alle benötigten Werte für große Matrizen abzuspeichern, weshalb es dort zu einer Priorisierung kommen muss, bei der bestimmte

Instruktionen nicht mit der Register-Variante arbeiten können, sondern vorher mittels zusätzlichen Instruktionen Werte in temporäre Register geschrieben werden müssen, was an der Stelle der jeweiligen Instruktion zu einer Verlangsamung führt. Daher ist es wichtig, für die performance-kritischsten Instruktionen Register zu nutzen und für die zweitrangigen Instruktionen langsamere Varianten zu nehmen.

Am wichtigsten ist dabei das Laden aus B und A innerhalb der k-Schleife. Für ldb muss daher ein Register reserviert werden, wenn k größer ist als 512. Ein Sonderfall ist A, weil es mittels VLDR-Instruktion und daher ohne die Möglichkeit eines Offsets mittels Register geladen werden muss. Bei großen n, d.h. ab 127 muss daher der Offset mittels ADD-Instruktion auf den Pointer zu A inkrementiert werden, was bis zu einer Größe von 1024 funktioniert. Danach muss auch für A ein Register genutzt werden, damit die ADD-Instruktion als Register-Offset-Variante ausgeführt werden kann. Wenn der 4x6 Microkernel genutzt wird, muss außerdem noch ein zweiter Zeiger auf B gespeichert werden. In der k-Schleife werden im 8x3 aus drei nachfolgenden Spalten Werte geladen, d.h. von den Werten B[0], B[ldb] und B[2*ldb]. Die LDR-Instruktion bietet die Möglichkeit einen Left-Shift von bis zu drei auf den Register-Offset anzugeben, was bei dem 8x3-Microkernel ausreicht. Bei dem 4x6-Microkernel ist es allerdings nicht mehr möglich nur mit diesem Shift, alle Werte abzubilden. Daher muss ein zweiter Pointer B[3*ldb] gehalten werden, der am Anfang des Microkernels berechnet wird und die Werte B[3*ldb]-B[5*ldb] mittels Left-Shift abbildet.

Die Generator-Klasse ermittelt, welche Register benötigt werden und weist je nach Priorität die jeweiligen Register zu.

6. Ergebnisse

6.1. Vergleichslösungen

Wie beschrieben, gibt es außer der CMSIS-DSP-Bibliothek von ARM keine Bibliotheken, die Matrixmultiplikation auf Armv8-M unterstützen. Deshalb dient diese Bibliothek als Referenz für die Performance. Außerdem wird die entwickelte Lösung (im folgenden: Till-JIT) mit dem naiven Matrixmultiplikationsalgorithmus (im folgenden: Naiv) aus Listing 3 sowie einem einfachen, mit intrinsischen Funktionen implementierten 8x3-Microkernel (im folgenden: Intrinsics) verglichen.

Um einen fairen Vergleich zu gewährleisten, werden für alle Lösungen die optimalen Compileroptionen gesetzt. Es wird die neueste Version von Armclang verwendet, welche zum aktuellen Zeitpunkt 6.24 ist.

Arm CMSIS-DSP Bibliothek Die Bibliothek arbeitet mit einer Row-Major-Datenstruktur und es wird keine Angabe der *leading dimensions* unterstützt. Aufgerufen wird die Matrix mit folgenden Befehlen:

```
arm_matrix_instance_f32 armA; // Arm Datenstruktur mit Zeiger auf Matrix und Dimensionen
arm_matrix_instance_f32 armB;
arm_matrix_instance_f32 armC;
arm_mat_init_f32(&armA, m, k, A); // Matrixobjekte erstellen mit angegebenen Dimensionen
arm_mat_init_f32(&armB, k, n, B);
arm_mat_init_f32(&armC, m, n, C);
arm_mat_mult_f32(&armA, &armB, &armC); // Ausführen
```

Zudem muss beachtet werden, dass nicht auf C addiert wird, sondern C beim Start auf Null gesetzt wird. Die Bibliothek arbeitet mit einem 4x4-Microkernel der über Intrinsics realisiert wird.

Um optimale Leistung zu erreichen, muss die Bibliothek mit Armclang kompiliert werden und das Flag ARM_MATH_HELIUM angegeben werden. Aktuellste Version ist 1.16.2 die auch für die nachfolgenden Tests genutzt wurde.

Intrinsics Microkernel Nach der Architektur von 3.2 wird dieser Microkernel entworfen. Es werden nur Dimensionen unterstützt, die ein Vielfaches des Microkernels sind.

Naive Matrixmultiplikation Der naive Matrixmultiplikationsalgorithmus dient als Referenz für die anderen Lösungen und berechnet für jeden Wert in C das Skalarprodukt.

6.2. Performance

Um die Performance zu messen, werden verschiedene Tests ausgeführt, um die Charakteristik der entwickelten Matrixmultiplikation zu ermitteln.

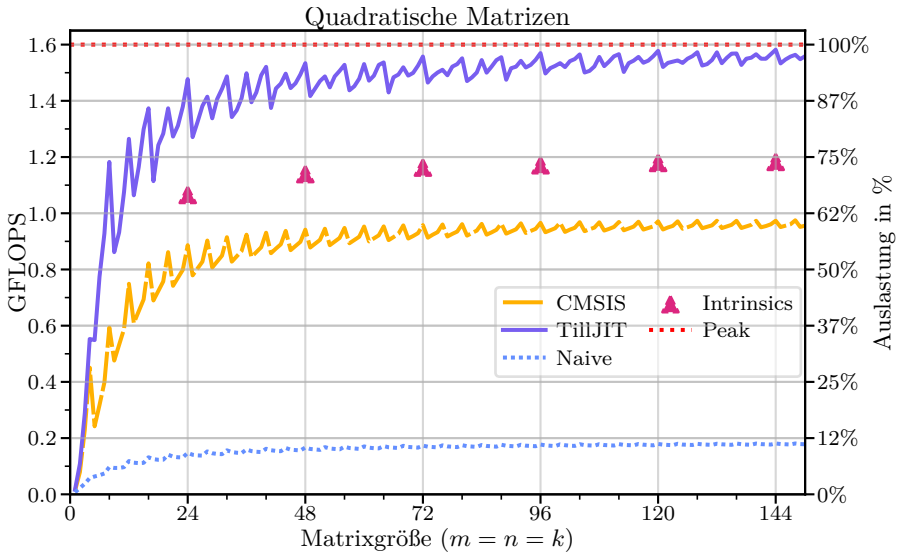
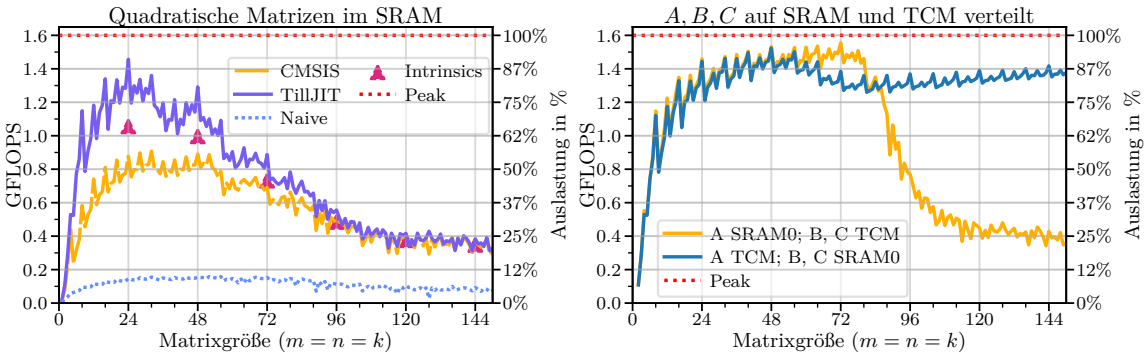


Abbildung 20: Test quadratischer Matrizen. Auf der rechten Seite ist die erreichte Auslastung der theoretisch möglichen Peak Performance angegeben.

Der Test von quadratischen Matrizen, gezeigt in Abbildung 20, zeigt die Performancecharakteristik des 8x3 Microkernels bereits deutlich. Immer wenn m ein Vielfaches von Acht ist, wird ein lokales Maximum erreicht. Direkt danach sinkt die Leistung stark ab, da nun die Instruktionen maskiert werden müssen und somit weniger Elemente pro Instruktion berechnet werden können. Danach wächst die Performance weiter bis vier überschüssige Elemente berechnet werden müssen, da nun wieder auf die Maskierung verzichtet werden kann. Danach muss wieder auf Maskierung zurückgegriffen, bis m wieder ein Vielfaches von Acht ist und wieder ein Maximum an Performance erreicht wird. Je größer dabei k wird, desto höher wird auch die Performance, da mehr Zeit in der k-Schleife verbracht wird.

Sowohl auf HP- als auch HE-Kern konnte dabei die theoretische Peak Performance zu 99% erreicht werden und ein durchschnittlicher Performancegewinn von 64% gegenüber CMSIS-DSP erzielt werden. In diesen und den folgenden Tests waren die Ergebnisse für HP- und HE-System sehr ähnlich, weshalb die Ergebnisse für das HE-System weggelassen werden.



(a) Quadratische Matrizen in SRAM0 (b) Test der Speicherbelegung bei TillJIT

Abbildung 21: Tests mit SRAM0

Wie Abbildung 21a zeigt, kann die Performance nicht gehalten werden, wenn langsamer Speicher mit Cache genutzt wird. Hier bricht die Performance ein, wenn die Matrizen nicht mehr in den Cache passen, wobei sich die Performance nun CMSIS-DSP annähert. In Abbildung 21b ist zu sehen, dass dies vor allem an der Matrix A liegt. Liegt diese im schnellen Speicher, sind immer noch Werte von bis zu 1.5 GFLOPS, d.h. 90% der Peak Performance erreichbar. Liegt A hingegen im langsamen SRAM0-Speicher, bricht die Performance ein. Der Grund liegt darin, dass in jedem Microkernel in der k-Schleife mittels der VLDR-Instruktion acht Elemente von A und mittels der LDR-Instruktion drei Werte für B geladen werden. Somit ist einerseits die Anzahl der benötigten Werte für A und höher und wie Abbildung 11 gezeigt hat, gleicht sich der Durchsatz der VLDR-Instruktion dem der LDR-Instruktion an, wenn die Daten nicht aus dem Cache geladen werden können.

In den folgenden Tests war das gleiche Verhalten zu beobachten, wobei der Performanceeinbruch aufgrund kleinerer Problemgrößen erst später eintritt. Daher sind die Grafiken für SRAM0 weggelassen.

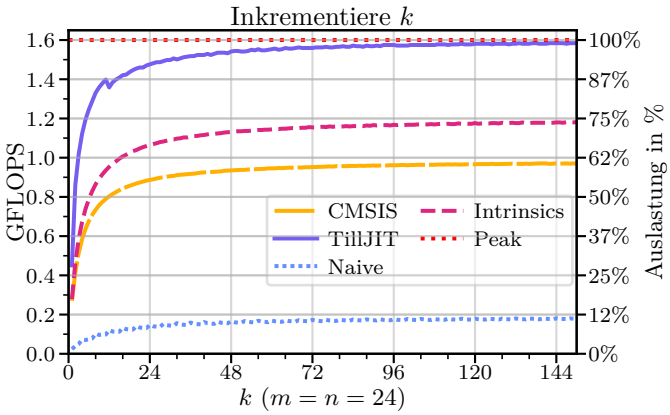
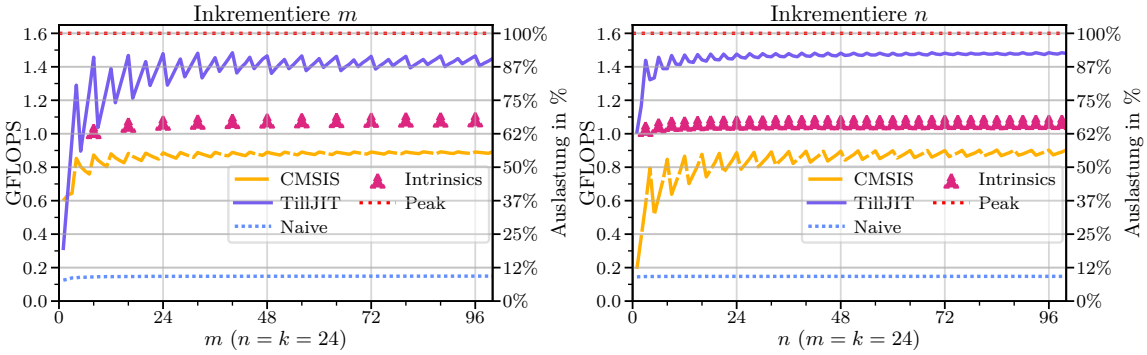


Abbildung 22: Test wachsendes k

Mit einem wachsenden k kann die Peak Performance getestet werden. Dabei ist in Abbildung 22 sehen, dass alle Implementierungen mit größeren k auch höhere Performance liefern. Dabei erreicht CMSIS-DSP allerdings nur 60% der Peak Performance während JIT bei knapp 100% landet und somit erneut eine durchschnittliche Verbesserung von 60% erzielt. Es tritt beständig, sowohl auf HP- als auch HE-System eine kleine Senke bei $k = 12$ auf, die auch nicht verschwindet, wenn der Ausrollfaktor für k angepasst wird.



(a) Test wachsendes m (b) Test wachsendes N

Abbildung 23: Test von wachsenden n und m

Beim Inkrementieren von m und n ist erneut die Performancecharakteristik des 8x3-Microkernels zu sehen. Beim Inkrementieren von m, abgebildet in 23a sind die lokalen Maxima bei Vielfachen von 4 und 8 zu sehen, während bei n (abgebildet in 23b) in jedem dritten Schritt ein Maximum erreicht wird. Für m und n werden Verbesserungen von 61% respektive 67% gegenüber CMSIS-DSP erzielt.

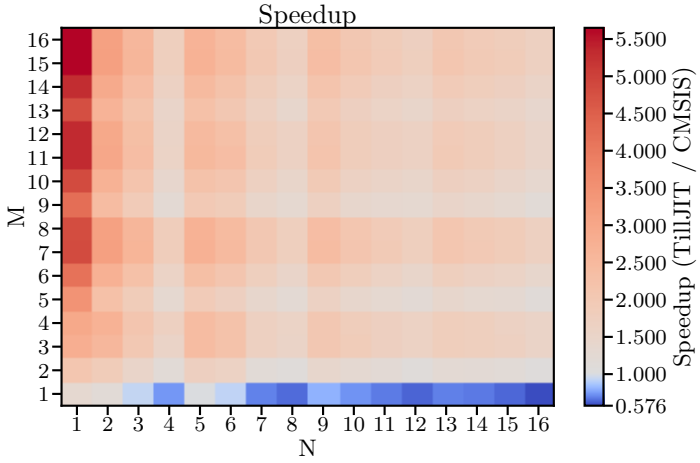


Abbildung 24: Speedup von TillJIT bei allen Größen $m, n \in \{1, \ldots, 16\}$ und $k = 16$

Zuletzt werden alle möglichen Größen von n und m von 1 bis 16 getestet, wobei k auf 16 festgelegt wird. Dabei ist in Abbildung 24 zu sehen, dass nur im Fall von $m = 1$ und einem großen n, TillJIT langsamer ist als CMSIS-DSP. In allen anderen Fällen ist die Geschwindigkeit höher. Insbesondere für den umgekehrten Fall, dass m groß ist und $n = 1$ ist dafür die TillJIT bis zu 5x schneller als CMSIS-DSP.

Durchschnittlich wird ein Geschwindigkeitszuwachs von 94% erzielt.

7. Forschungsstand

Zu effizienter Matrixmultiplikation gibt es eine Vielzahl an Literatur [20, 26, 35, 37], wobei sich die vorliegende Arbeit dabei v.a. auf BLIS und LIBXSMM [25] konzentriert hat.

Die bisherige Forschung zu Cortex-M-Prozessoren hat sich bislang beinahe ausschließlich auf die älteren Prozessoren wie Cortex-M7 und Cortex-M4 gestützt [28, 29].

Für die Anwendung der Helium-Erweiterung existiert nur Literatur für Multiplikation

von Zahlen [15, 16] und dünnbesetzte Matrix-Vektor-Multiplikation [36]. Erwähnenswert ist außerdem das SLOTHY-Framework [1] zur Optimierung der Instruktionsanordnung für die Helium-Erweiterung.

Von Arm gibt es mehrere Dokumente, die eine Beschreibung der Helium-Erweiterung liefern [5, 8, 21–24, 30, 34], auf die sich insbesondere der erste Teil dieser Arbeit gestützt hat.

8. Zusammenfassung und Ausblick

In dieser Arbeit habe ich eine Bibliothek zur Matrixmultiplikation für die Helium-Erweiterung der Cortex-M-Prozessoren vorgestellt. Dazu wurden die Performanceeigenschaften der Armv8.1-Architektur und der Helium-Erweiterung herausgearbeitet, ein JIT-Compiler für Matrixmultiplikation entwickelt und damit eine Bibliothek zur Matrixmultiplikation implementiert.

Dabei hat sich gezeigt, dass die Helium-Erweiterung einerseits eine solide Grundlage für die performante Implementierung von Matrixmultiplikation darstellt, andererseits aber auch ein großer Aufwand betrieben werden muss, um eine optimale Instruktionsanordnung zu erreichen. Die Instruktionsanordnung ist dabei der wichtigste Bestandteil für das Erreichen der Peak Performance, wobei zudem entscheidend ist, dass auch entsprechende Instruktionen zum Überlagern vorhanden sind. Hilfreich haben sich dabei auch die Low-Overhead-Loops gezeigt.

In einem Großteil der Fälle war die vorgestellte Implementierung schneller als die bisherige Referenzbibliothek von Arm mit einem durchschnittlichen Geschwindigkeitszuwachs von 94%. Zudem konnte die theoretisch maximal erreichbare Leistung bei großen k-Werten zu 99% erreicht werden. Optimierungspotenzial besteht dabei allerdings noch, wenn $m = 1$ ist.

Für die Performance förderlich war dabei zudem auch der schnelle TCM-Speicher, welcher es erlaubt, ohne Wartezeiten auf Speicher zuzugreifen. Weitere Tests sind allerdings nötig, um das Verhalten des SRAM-Speichers zu erklären, da dieser einen starken Einbruch zeigt. Für bessere Leistung im langsamen Speicher wäre der nächste Schritt, die Matrixmultiplikation in größere Blöcke einzuteilen, von denen jeder in den L1-Cache passt, um damit den Cache besser auszunutzen und die Leistungsfähigkeit zu steigern.

Da Matrixmultiplikation nur ein kleiner, wenn auch wichtiger, Teil der Tensoroperationen ist, müssen noch weitere Operationen hinzugefügt werden, um weitergehende Berechnungen auszuführen und die Einbettung in weitere Anwendungen voranzutreiben.

Literaturverzeichnis

[1] A. Abdulrahman, H. Becker, M. J. Kannwischer und F. Klein, *Fast and Clean: Auditable High-Performance Assembly via Constraint Solving*, https://eprint.iacr.org/2022/1303, 2022. besucht am 25. Juni 2025.

[2] Alif Semiconductors, *Datasheet Ensemble Family E7 Series ADTS0005 v2.11*, Jan. 2025.

[3] Arm Ltd., *Arm Cortex-M Processor Comparison Table*, https://armkeil.blob.core.windows.net/developer/Files/pdf/product-brief/arm-cortex-m-processor-comparison-table.pdf, 2023. besucht am 24. Juni 2025.

[4] Arm Ltd., „Arm® Cortex®-M55 Processor Devices Generic User Guide," Nr. 03, 2024.

[5] Arm Ltd., „Arm® Cortex®-M55 Processor Software Optimization Guide," *Arm*, Nr. 03, 2022.

[6] Arm Ltd., „Arm® Cortex®-M55 Processor Technical Reference Manual," Nr. 03, 2024.

[7] Arm Ltd., „Arm® Cortex®-M85 Processor Software Optimization Guide," Nr. 01, 2023.

[8] Arm Ltd., „Armv8-M Architecture Reference Manual," Nr. DDI0553B.y, 2015.

[9] Arm Ltd., „Armv8.1-M Performance Monitoring User Guide," Nr. 1.186, 2020.

[10] Arm Ltd., *CMSIS: Introduction*, https://arm-software.github.io/CMSIS_6/latest/General/index.html. besucht am 20. Juli 2025.

[11] Arm Ltd., *Compare IP*, https://developer.arm.com/compare-ip/#cortex-m-cpu-performance---scalar. besucht am 9. Juli 2025.

[12] Arm Ltd., „Introduction to SVE," Nr. 01, 2025.

[13] Arm Ltd., *TOSA 1.0.0 Specification*, https://www.mlplatform.org/tosa/tosa_spec.html#_matmul, 2025. besucht am 21. Juli 2025.

[14] J.-L. Aufranc, *Bestechnic BES2700YP Arm Cortex-M55 Bluetooth Audio SoC targets headphones, earbuds, portable speakers - CNX Software*, https://www.cnx-software.com/2025/06/02/bestechnic-bes2700yp-arm-cortex-m55-bluetooth-audio-soc-targets-headphones-earbuds-portable-speakers/, Juni 2025. besucht am 14. Juli 2025.

[15] H. Becker, V. Hwang, M. J. Kannwischer, L. Panny und B.-Y. Yang, „Efficient Multiplication of Somewhat Small Integers Using Number-Theoretic Transforms," in *Advances in Information and Computer Security*, C.-M. Cheng und M. Akiyama, Hrsg., Bd. 13504, Cham: Springer International Publishing, 2022, S. 3–23, ISBN: 978-3-031-15254-2 978-3-031-15255-9. DOI: 10.1007/978-3-031-15255-9_1. besucht am 12. Juni 2025.

[16] H. Becker, J. M. B. Mera, A. Karmakar, J. Yiu und I. Verbauwhede, *Polynomial Multiplication on Embedded Vector Architectures*, https://eprint.iacr.org/2021/998, 2021. besucht am 17. Apr. 2025.

[17] T. Christina, *GCC 15 Continuously Improving AArch64 - Tools, Software and IDEs blog - Arm Community blogs - Arm Community*, https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/gcc-15-continuously-improving, Juni 2025. besucht am 9. Juli 2025.

[18] R. Dirvin, *Next-Generation Armv8.1-M Architecture: Delivering Enhanced Machine Learning and Signal Processing for the Smallest Embedded Devices*, https://newsroom.arm.com/news/next-generation-armv8-1-m-architecture-delivering-enhanced-machine-learning-and-signal-processing-for-the-smallest-embedded-devices, Feb. 2019. besucht am 8. Juli 2025.

[19] E. Georganas u. a., *Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning & HPC Workloads*, Nov. 2021. DOI: 10.1145/3458817.3476206. arXiv: 2104.05755 [cs]. besucht am 27. März 2025.

[20] K. Goto und R. A. V. D. Geijn, „Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, Jg. 34, Nr. 3, S. 1–25, Mai 2008, ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1356052.1356053. besucht am 15. Apr. 2025.

[21] T. Grocutt, *Making Helium: Bringing Amdahl's law to heel*, https://community.arm.com/arm-research/b/articles/posts/making-helium-bringing-amdahl-s-law-to-heel, März 2019. besucht am 30. Apr. 2025.

[22] T. Grocutt, *Making Helium: Going around in circles*, https://community.arm.com/arm-research/b/articles/posts/making-helium-going-around-in-circles, Feb. 2019. besucht am 30. Apr. 2025.

[23] T. Grocutt, *Making Helium: Sudoku, registers and rabbits*, https://community.arm.com/arm-research/b/articles/posts/making-helium-sudoku-registers-and-rabbits, Feb. 2019. besucht am 30. Apr. 2025.

[24] T. Grocutt, *Making Helium: Why Not Just Add Neon?* https://community.arm.com/arm-research/b/articles/posts/making-helium-why-not-just-add-neon, Feb. 2019. besucht am 30. Apr. 2025.

[25] A. Heinecke, G. Henry, M. Hutchinson und H. Pabst, „LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT: IEEE, Nov. 2016, S. 981–991, ISBN: 978-1-4673-8815-3. DOI: 10.1109/SC.2016.83. besucht am 19. Juni 2025.

[26] J. Huang und R. A. van de Geijn, *BLISlab: A Sandbox for Optimizing GEMM*, Sep. 2016. DOI: 10.48550/arXiv.1609.00076. arXiv: 1609.00076 [cs]. besucht am 22. Juli 2025.

[27] L. Lai, N. Suda und V. Chandra, *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs*, Jan. 2018. DOI: 10.48550/arXiv.1801.06601. arXiv: 1801.06601 [cs]. besucht am 27. März 2025.

[28] I. Lucan Orășan, C. Seiculescu und C. D. Căleanu, „A Brief Review of Deep Neural Network Implementations for ARM Cortex-M Processor," *Electronics*, Jg. 11, Nr. 16, S. 2545, Aug. 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11162545. besucht am 25. Juni 2025.

[29] A. Maciá-Lillo, S. Barrachina, G. Fabregat und M. F. Dolz, „Optimizing Convolutions for Deep Learning Inference on ARM Cortex-M Processors," *IEEE Internet of Things Journal*, Jg. 11, Nr. 15, S. 26 203–26 219, Aug. 2024, ISSN: 2327-4662. DOI: 10.1109/JIOT.2024.3395335. besucht am 1. Apr. 2025.

[30] J. Marsh, *Arm Helium Technology Reference Book*. Arm Education Media, 2020, ISBN: 978-1-911531-24-1.

[31] Qualcomm, *Snapdragon W5+ Gen1 Product Brief*, https://docs.qualcomm.com/bundle/publicresource/87-43671-1_REV_B_Snapdragon_W5__and_Snapdragon_W5_Gen_1_Wearable_Platforms_Product_Brief.pdf, Juli 2024. besucht am 25. Juli 2025.

[32] M. Sauter, *Exynos W920: Samsung hat ersten 5-nm-Chip für Smartwatches - Golem.de*, https://www.golem.de/news/exynos-w920-samsung-hat-ersten-5-nm-chip-fuer-smartwatches-2108-158780.html, Aug. 2021. besucht am 14. Juli 2025.

[33] A. Semiconductors, *Alif Ensemble Family Is Available Now*, https://alifsemi.com/press-release/alif-semiconductor-ensemble-mcus/, Nov. 2023. besucht am 23. Juni 2025.

[34] A. Skillman und T. Edso, „A Technical Overview of Cortex-M55 and Ethos-U55: Arm's Most Capable Processors for Endpoint AI," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, Palo Alto, CA, USA: IEEE, Aug. 2020, S. 1–20, ISBN: 978-1-7281-7129-6. DOI: 10.1109/HCS49909.2020.9220415. besucht am 3. Apr. 2025.

[35] T. M. Smith, R. V. D. Geijn, M. Smelyanskiy, J. R. Hammond und F. G. V. Zee, „Anatomy of High-Performance Many-Threaded Matrix Multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, Phoenix, AZ, USA: IEEE, Mai 2014, S. 1049–1059, ISBN: 978-1-4799-3800-1 978-1-4799-3799-8. DOI: 10.1109/IPDPS.2014.110. besucht am 15. Apr. 2025.

[36] E. Trommer, B. Waschneck und A. Kumar, *dCSR: A Memory-Efficient Sparse Matrix Representation for Parallel Neural Network Inference*, Nov. 2021. DOI: 10.48550/arXiv.2111.12345. arXiv: 2111.12345 [cs]. besucht am 22. Juli 2025.

[37] F. G. Van Zee und R. A. Van De Geijn, „BLIS: A Framework for Rapidly Instantiating BLAS Functionality," *ACM Trans. Math. Softw.*, Jg. 41, Nr. 3, S. 1–33, Juni 2015, ISSN: 0098-3500, 1557-7295. DOI: 10.1145/2764454. besucht am 15. Apr. 2025.

[38] A. Vaswani u. a., *Attention Is All You Need*, Aug. 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs]. besucht am 21. Juli 2025.

[39] Y. Xu, T. M. Khan, Y. Song und E. Meijering, „Edge Deep Learning in Computer Vision and Medical Diagnostics: A Comprehensive Survey," *Artificial Intelligence Review*, Jg. 58, Nr. 3, S. 93, Jan. 2025, ISSN: 1573-7462. DOI: 10.1007/s10462-024-11033-5.

[40] J. Yiu, „Blending DSP and ML features into a low-power general-purpose processor – how far can we go?" *Arm Ltd.*, 2020.

[41] J. Yiu, „Cortex-M for Beginners," *Arm Ltd.*, März 2017.

[42] J. Yiu, „Introduction to the Arm Cortex-M55 Processor," *Arm Ltd.*, Feb. 2020.

[43] J. Yiu, *Using Cortex-M55 and Armv8.1-M processors - Architectures and Processors blog - Arm Community blogs - Arm Community*, https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8_2d00_m-based-processor-software-development-hints-and-tips, Okt. 2021. besucht am 25. Juni 2025.

Anlagen

Reproduzierbare Tests

Im Folgenden eine Auflistung welches Projekt und welche Funktion für die jeweiligen Ergebnisse zuständig ist. Die einzelnen Projekte sind im src-Ordner zu finden:

Kapitel 2

- Benchmark Dual Issue: Projekt helium_instructions, Datei: main.cpp, instructions.s, Funktion: testDualIssue
- Benchmark Low Overhead Loops: Projekt: helium_instructions, Datei: main.cpp, instructions.s, Funktion: testBranching()
- Benchmark Alignment: Projekt: helium_instructions, Datei: main.cpp, instructions.s, Funktion: testAlignment()

Kapitel 3

- Benchmark Peak Performance: Projekt micro_benchmarks, Datei: micro_benchmark.cpp, Funktion: benchmarkFlops().
- Benchmarks Durchsatz: Projekt micro_benchmarks, Datei: micro_benchmark.cpp, Funktion: benchmarkThroughput().

Kapitel 4

- Benchmark Peak Performance: Projekt jit_test, Datei: main.cpp, jit_tests.cpp, Funktion: testPeakPerformance()
- Benchmark Throughput: Projekt jit_test, Datei: main.cpp, jit_tests.cpp, Funktion: testThroughput()

Kapitel 5

- Benchmark 4x6 Microkernel: Wie der Square-Test in Kapitel 6. Dabei wurde Nutzung des 4x6-Microkernels deaktiviert bzw. aktiviert in der Generator-Klasse.
- Ablation: Tests in eigenem Branch „ablation“. Dort existiert für die Baseline („TEST NoInterleaving“), für die einfache Überlagerung („TEST NoLoopPeeling“) und für die erste Variante des Loop Peelings („TEST BadLoopPeeling“) ein Commit. Ausrollfaktoren im normalen Branch manuell angepasst. Jede Variante mit der Funktion constSizeTest() in gemm_tests.cpp ausgeführt.

Kapitel 6 Ergebnisse im results-Ordner unter jit_gemm zu finden.

- Square (inkl. SRAM): Projekt jit_test, Datei: main.cpp, gemm_tests.cpp, Funktion: testSquareShapes(). Jeweils mit verschiedenen Arrays getestet für SRAM0.
- Inkrementiere K: Funktion: testGrowingK()
- Inkrementiere N: Funktion: testGrowingN()
- Inkrementiere M: Funktion: testGrowingM()
- Vergleich aller Größen: Funktion: testAllSizes()

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Eigenständigkeitserklärung

1. Hiermit versichere ich, dass ich die vorliegende Arbeit - bei einer Gruppenarbeit die von mir zu verantwortenden und entsprechend gekennzeichneten Teile - selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich trage die Verantwortung für die Qualität des Textes sowie die Auswahl aller Inhalte und habe sichergestellt, dass Informationen und Argumente mit geeigneten wissenschaftlichen Quellen belegt bzw. gestützt werden. Die aus fremden oder auch eigenen, älteren Quellen wörtlich oder sinngemäß übernommenen Textstellen, Gedankengänge, Konzepte, Grafiken etc. in meinen Ausführungen habe ich als solche eindeutig gekennzeichnet und mit vollständigen Verweisen auf die jeweilige Quelle versehen. Alle weiteren Inhalte dieser Arbeit ohne entsprechende Verweise stammen im urheberrechtlichen Sinn von mir.

2. Ich weiß, dass meine Eigenständigkeitserklärung sich auch auf nicht zitierfähige, generierende KI-Anwendungen (nachfolgend „generierende KI“) bezieht. Mir ist bewusst, dass die Verwendung von generierender KI unzulässig ist, sofern nicht deren Nutzung von der prüfenden Person ausdrücklich freigegeben wurde (Freigabeerklärung). Sofern eine Zulassung als Hilfsmittel erfolgt ist, versichere ich, dass ich mich generierender KI lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss deutlich überwiegt. Ich verantworte die Übernahme der von mir verwendeten maschinell generierten Passagen in meiner Arbeit vollumfänglich selbst. Für den Fall der Freigabe der Verwendung von generierender KI für die Erstellung der vorliegenden Arbeit wird eine Verwendung in einem gesonderten Anhang meiner Arbeit kenntlich gemacht. Dieser Anhang enthält eine Angabe oder eine detaillierte Dokumentation über die Verwendung generierender KI gemäß den Vorgaben in der Freigabeerklärung der prüfenden Person. Die Details zum Gebrauch generierender KI bei der Erstellung der vorliegenden Arbeit inklusive Art, Ziel und Umfang der Verwendung sowie die Art der Nachweispflicht habe ich der Freigabeerklärung der prüfenden Person entnommen.

3. Ich versichere des weiteren, dass die vorliegende Arbeit bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt wurde oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen ist.

4. Mir ist bekannt, dass ein Verstoß gegen die vorbenannten Punkte prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass meine Prüfungsleistung als Täuschung und damit als mit „nicht bestanden“ bewertet werden kann. Bei mehrfachem oder schwerwiegendem Täuschungsversuch kann ich befristet oder sogar dauerhaft von der Erbringung weiterer Prüfungsleistungen in meinem Studiengang ausgeschlossen werden.

_____

Ort und Datum

_____

Unterschrift