FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 3

Rechnerarchitektur

Konstantin Müller

Ressourcenbeschränkte Inferenz von Open-Source Sprachmodellen auf FPGAs

Masterarbeit im Fach Informations- und Kommunikationstechnik

1. Dezember 2025

Please cite as:
Konstantin Müller, “Ressourcenbeschränkte Inferenz von Open-Source Sprachmodellen auf FPGAs,” Master’s Thesis (Masterarbeit), University of Erlangen, Dept. of Computer Science, 2025.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnerarchitektur

Martensstr. 3 · 91058 Erlangen · Germany

www3.cs.fau.de

Ressourcenbeschränkte Inferenz von Open-Source Sprachmodellen auf FPGAs

Masterarbeit im Fach Informations- und Kommunikationstechnik

vorgelegt von

Konstantin Müller

geb. am 13. August 1999
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 3
Rechnerarchitektur

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**M. Sc. Philipp Gündisch**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Dietmar Fey**

Beginn der Arbeit:	**1. Juni 2025**
Abgabe der Arbeit:	**1. Dezember 2025**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Konstantin Müller)
Erlangen, 24. November 2025

Zusammenfassung

Moderne Large Language Model (LLM) sind äußerst rechenintensiv und profitieren während der Inferenzphase häufig von einer Auslagerung auf spezialisierte Hardware. Field Programmable Gate Arrays (FPGA) bieten aufgrund ihrer stark parallelen Architektur ein hohes Potenzial zur Beschleunigung solcher Modelle. Ihre begrenzte Speicherkapazität stellt jedoch eine Einschränkung dar, da große LLMs typischerweise nicht vollständig auf einem FPGA ausgeführt werden können. In den vergangenen Jahren wurden mehrere kompaktere Sprachmodelle entwickelt und öffentlich zugänglich gemacht, die trotz ihres reduzierten Umfangs eine solide Leistungsfähigkeit, vor allem in spezialisierten Aufgaben, bieten. Zum einen ergeben sich dadurch neue Ansätze für eine effiziente Beschleunigung auf FPGAs. Zum anderen lassen sich diese kompakten Modelle auch auf leistungsarmer Hardware ausführen und dadurch in studentischen Arbeiten ohne teure Infrastruktur nutzen und wissenschaftlich analysieren.

Diese Arbeit untersucht, wie sich die Inferenz auf ressourcenbeschränkter Hardware umsetzen lässt und wann sich eine Auslagerung auf ein FPGA lohnen kann.

Ein erster Beitrag dazu ist eine anschauliche Beschreibung der Inferenz eines LLM sowie ihrer wichtigsten Abläufe und Komponenten. Zusätzlich wird eine Übersicht über aktuelle Optimierungstechniken gegeben, die die Laufzeit und den Speicherverbrauch von LLMs verringern können. In der Arbeit wurde die Modellfamilie Qwen3 sowie das Inferenztool `llama.cpp` verwendet und detailliert beschrieben. Eine visuelle Analyse zeigt, wie sich das Projekt in zukünftigen Arbeiten für Messungen anpassen lässt.

Am Beispiel des Modells Qwen3 wird anschließend eine theoretische Analyse des Rechenaufwands der Inferenz durchgeführt. Dafür wurde die OpenCL-Implementierung von `llama.cpp` analysiert. In einer Roofline-Analyse werden verschiedene Phasen und Architekturkomponenten der Inferenz gemessen und deren Rechenintensität verglichen. Der Einsatz verschiedener Messtools auf lokaler Hardware und auf dem NHR-Cluster der FAU zeigt die Möglichkeiten und die Herausforderungen bei der Messung der LLM-Inferenz. Durch eine Roofline-Analyse konnten rechenintensive Teile identifiziert werden, bei denen eine Auslagerung auf ein FPGA sinnvoll sein kann.

Um die Inferenz auf einem FPGA auszuführen, wurde ein am Lehrstuhl 3 entwickeltes Tool verwendet, mit dem sich OpenCL in VHDL-Code kompilieren und schließlich auf einem FPGA synthetisieren lässt. Die OpenCL Kernel von `llama.cpp` wurden dafür angepasst. Es war möglich, die Kernel in korrekten VHDL-Code zu übersetzen.

Darüber hinaus wurde anhand eines Vergleichs mit anderen Modellen beschrieben, welche zusätzlichen Schritte künftig erforderlich sind, um auch weitere Modellfamilien auf dem FPGA ausführen zu können.

Die Ergebnisse dieser Arbeit liefern eine Grundlage für die Bewertung der Einsatzmöglichkeiten von FPGAs bei der Inferenz kleiner LLMs. Durch die theoretischen Analysen und Messungen konnten Szenarien identifiziert werden, in denen die Inferenz von erhöhter Rechenleistung profitiert.

Inhaltsverzeichnis

1 Motivation

Die Veröffentlichung von ChatGPT Ende 2022 hat einen erheblichen Hype um LLMs ausgelöst. Seitdem ist das Interesse an dieser Technologie sowohl in der breiten Öffentlichkeit als auch in der wissenschaftlichen Forschung stark gestiegen.
Die Entwicklung mit der größten öffentlichen Wahrnehmung wird dabei von großen Unternehmen wie OpenAI (ChatGPT) oder Google (Gemini) vorangetrieben, die immer leistungsfähigere Modelle mit stetig wachsenden Parameterzahlen veröffentlichen. Die Fortschritte dieser großen Modelle werden vor allem auch durch längere Trainingszeiten der LLMs und einem Hochskalieren der Trainingsinfrastruktur erreicht. Dafür wird teils immense Infrastruktur bzw. große Rechenzentren verwendet.
Aufgrund der hohen Kosten und den meistens nicht öffentlichen Modellen ist unabhängige Forschung auf diesem Gebiet teilweise nur eingeschränkt möglich.
Es gibt jedoch auch zwei Trends, die in eine andere Richtung gehen.
Zum einen veröffentlichen mehrere Anbieter öffentlich zugängliche Modelle. Die Gewichte, Architekturen und Parameter dieser Modelle sind frei verfügbar. Mit passender Software lassen sich diese Modelle lokal und kostenlos ausführen. Zudem können die Modelle angepasst und analysiert werden, beispielsweise durch die Verwendung anderer Datentypen oder eine veränderte Architektur. Diese lokal ausführbaren Modelle sind sowohl für Privatnutzer als auch für Firmen interessant. Neben eingesparten Kosten ist auch die Kontrolle über die Daten ein großer Vorteil, insbesondere für Unternehmen, die aus rechtlichen Gründen Daten nur lokal verarbeiten dürfen.
Ein weiterer Trend ist der zunehmende Fokus auf kleine Modelle, die sowohl weniger Speicherplatz als auch weniger Berechnungsaufwand benötigen. Beispiele hierfür sind die Modellvarianten Qwen3 (600 M Parameter) und Gemma3 (270 M Parameter), die deutlich kompakter sind als leistungsstarke Modelle mit mehreren hundert Milliarden Pararmetern. Diese Modelle lassen sich auch auf Consumer-Grade-Laptops mit begrenzten Speicher- und Rechenressourcen verwenden. Dadurch, dass mit beschränkter Hardware gearbeitet wird, sind diese Forschungsgebiete für wissenschaftliche Arbeiten, die keine großen Datenzentren zur Verfügung haben, besonders interessant.
In dieser Arbeit sollen die beiden Trends komibiniert werden. Kompakte, öffentlich zugängliche Modelle sollen verwendet und analysiert werden. Es soll geklärt werden, wie sich diese Modelle in ressourcenbeschränkten Umgebungen nutzen lassen. Die Analyse geht dabei vor allem auf die Komplexität der benötigten Berechnungen für die Inferenz von LLMs ein.
Ziel ist es, den Berechnungsaufwand der Textgenerierung einer LLM genau zu analysieren. Anhand dieser Analyse soll geklärt werden, welche Teile der Berechnung rechnerisch aufwändig sind und wann es sich lohnt, diese auf einen Hardwarebeschleuniger auszulagern. Verwendet wird dafür das öffentliche Projekt `llama.cpp`, bzw dessen OpenCL-Implementierung. Um die Ergebnisse praktisch umzusetzen, wurde der Code von `llama.cpp` anschließend so angepasst, dass er sich mit einem vom LS3 verwendeten Tool auf ein FPGA synthetisieren lässt.
Die Frage, ab wann die Berechnung einer LLM von einer Auslagerung auf einen Hardwarebeschleuniger profitiert, bietet durch zunehmend realistische Anwendungsfälle eine wissenschaftliche Relevanz.
Die Arbeit ist dafür in drei Teile gegliedert:
Der erste Teil beschäftigt sich mit der Beschreibung von LLMs im Allgemeinen und den für die Hardwareanalyse wichtigsten Aspekten.
Im Hauptteil der Arbeit wird darauf aufbauend der theoretische Rechenaufwand und Speicherverbrauch der LLM analysiert. Hier soll geklärt werden, ob die Berechnung der LLM durch Rechenleistung oder durch das Warten auf Daten beschränkt ist.Dafür wird der Code der OpenCL-Implementierung des `llama.cpp`-Projektes verwendet und die benötigten FLOPs (Floating Point Operations) analysiert. Dadurch lässt sich der Rechenaufwand für einzelne Teile oder die gesamte LLM-Inferenz einer Anfrage abschätzen. Mit Hilfe des Roofline-Modells werden verschiedene Teile der LLM-Architektur beziehungsweise Berechnungsphasen verglichen und geklärt, wann sich die Auslagerung auf einen Hardwarebeschleuniger lohnen kann. Um die Formeln zu validieren, wurden diese theoretischen Überlegungen durch Messungen ergänzt.
Ein letzter Teil der Arbeit behandelt die Portierung des `llama.cpp`-Projekts auf den am LS3 genutzten BRIG-Compiler. Dadurch soll sich die OpenCL Implementierung auf einem FPGA ausführen lassen. Neben der Beschreibung der dafür benötigten Änderungen wird auch ein Ausblick auf zukünftig mögliche Erweiterungen für andere Modellfamilien gegeben.

2 LLM-Architektur

Um den Rechenaufwand eines LLM zu analysieren, soll zuerst eine Übersicht über die Architektur und die benötigten Berechnungen der Inferenz gegeben werden. Neben der Beschreibung der Kernmechanismen soll auch eine Übersicht über aktuelle Entwicklungen und Optimierungsmöglichkeiten gezeigt werden.

2.1 Entwicklung

Bis 2017 wurden für Aufgaben der Textverarbeitung überwiegend Recurrent neural network (RNN)-Modelle eingesetzt. Ein RNN verarbeitet den Inputtext sequentiell, wobei der interne Zustand des Netzes jeweils aktualisiert wird.
Bei langen Textsequenzen treten jedoch Probleme auf. Durch explodierende oder verschwindende Gradienten fällt es RNNs schwer, weit auseinanderliegende Abhängigkeiten zu erfassen. In Abbildung 1 ist ein unrolled RNN-Netz dargestellt, das abhängig von den Eingabewörtern seinen internen Zustand verändert und Wortvorhersagen trifft.

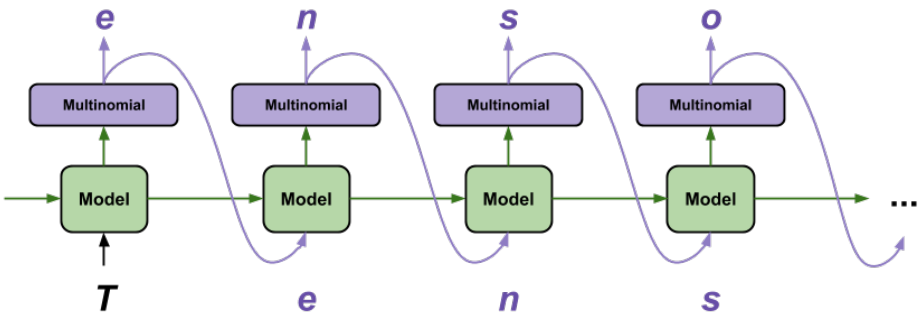


Abbildung 1: Unrolled RNN Quelle: [1]

2017 stellten Vaswani et al. im Paper *Attention Is All You Need* [26] den Attention-Mechanismus vor. Dieser verarbeitet Eingabesequenzen nicht schrittweise, sondern gleichzeitig und kann dabei Kontextinformationen zwischen allen Tokens berücksichtigen. Moderne LLMs basieren auf diesem Mechanismus für die Textvorhersage und bestehen im Wesentlichen aus drei Hauptkomponenten:

1. Umwandlung der Eingabesequenz in Vektoren fester Länge (Embeddings).
2. Verarbeitung durch mehrere aufeinanderfolgende Attention-Blöcke, die die Wortbedeutungen anhand des Kontexts anpassen.
3. Ausgabe von Wahrscheinlichkeiten für das nächste Token durch ein Feed-Forward-Netz in Kombination mit einer Softmax-Wahrscheinlichkeitsverteilung.

Frühe Modelle, wie die Architektur in der Arbeit Attention Is All You Need [26], waren in Encoder- und Decoder-Teile aufgeteilt. Der Encoder erfasst den Kontext der Eingabesequenz, während der Decoder diese Informationen nutzt, um eine neue Sequenz zu generieren, die semantisch mit der Eingabe verbunden ist.
Der Großteil der modernen LLMs zur Textgenerierung besteht heutzutage nur noch aus einem Decoder Teil. Durch diese architektonische Vereinfachung benötigen die Modelle weniger Ressourcen und lassen sich einfacher skalieren. Die in Vergleichen am besten abschneidenden Modelle wie GPT-4 oder Gemini sind vom Typ Decode-only. Ein LLM, das hingegen eine Encoder-only Architektur verwendet, ist BERT[2], ein von Google entwickeltes Modell zum Textverständnis.

2.2 Autoregressive Decoder

Die Hauptaufgabe von LLMs, die in dieser Arbeit betrachtet werden soll, sind autoregressive Textvervollständiger. Bei jeder Vorwärtsberechnung der Inferenz wird auf Basis der Eingabesequenz ein einzelnes Token erzeugt. Diese erzeugten Token werden schrittweise an den Eingabetext angehängt und als neuer

[1] https://blog.tensorflow.org/2019/02/mit-deep-learning-basics-introduction-tensorflow.html
[2] https://arxiv.org/abs/1810.04805

Input der nächsten Inferenz verwendet, bis der vollständige Antwortsatz generiert ist. Dieses Vorgehen beschreibt die autoregressive Eigenschaft von LLMs. Die Inferenz endet entweder, wenn das Modell ein spezielles Endsymbol (⟨EOS⟩) ausgibt oder wenn die maximal festgelegte Sequenzlänge erreicht ist [4].

2.2.1 Input Embedding

Damit LLMs Eingabetexte verarbeiten können, werden diese in Vektoren, sogenannte Embeddings, überführt. Im ersten Schritt, dem Input Encoding, wird der Eingabetext in Tokens zerlegt. Tokens können dabei ganze Wörter oder Wortbestandteile darstellen.
Ein Beispiel für die Tokenisierung des Satzes "Die FAU ist eine tolle Universität:

```
["Die", "F", "A", "U", "ist", "eine", "tolle", "Univers", "ität", "."]
```

Um Tokens in Vektorform zu bringen, werden die Werte aus einem festgelegten Wörterbuch mit Positionsinformationen kombiniert, die die jeweilige Position im Satz kennzeichnen. Diese Positionsinformationen sind entscheidend, da semantische Abhängigkeiten oft von der relativen Distanz zweier Wörter abhängen. Ohne Positionsdaten würde eine vertauschte Wortreihenfolge dieselbe Ausgabe liefern.
Die Dimension der Embeddings ist fest und wird durch einen Hyperparameter vorgegeben. Das LLaMA 3 Modell verwendet beispielsweise eine Embedding-Dimension von 4096.

Positional Encoding Vor allem in ersten Ansätzen zu LLMs wie zB in *Attention Is All You Need* oder BERT werden Positionsinformationen trigonometrisch codiert. Die Position *pos* und der Dimensionsindex i bestimmen den Wert der Codierung:

$$PE_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{1}$$

$$PE_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2}$$

Für gerade Dimensionen wird eine Sinus-, für ungerade eine Kosinusfunktion genutzt. Die Werte liegen im Bereich [-1,1].
Das finale Embedding ergibt sich durch Addition der Positionswerte zum Wörterbucheintrag des Tokens. Abbildung 2 zeigt die kontinuierlich verlaufende Codierung der Positionen. Bei niedrigen Dimensionen ist der Nenner der Formel klein und die Sinus/Cosinus-Funktionen unterschiedlicher Embeddings haben entlang der Positions-Achse stark unterschiedliche Werte. Bei hohen Dimensionen ähneln sich die Werte stärker. Am Anfang jedes Embeddingvektors werden lokale Positionsunterschiede dadurch unterschiedlicher dargestellt.

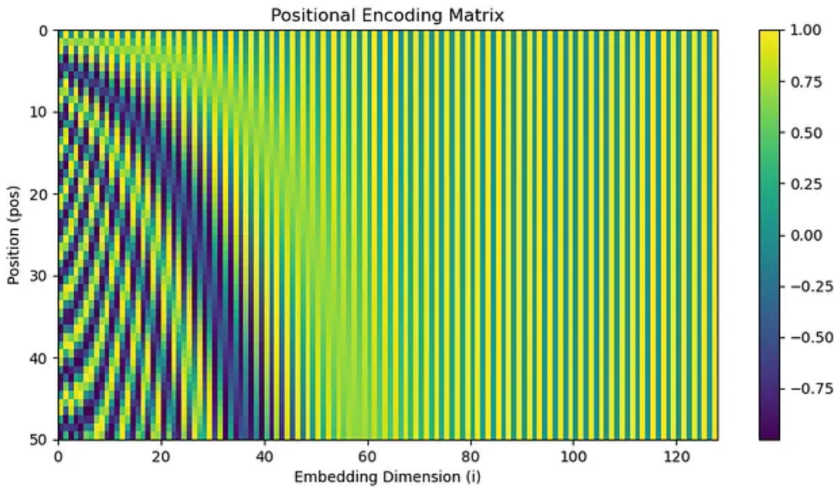


Abbildung 2: Sinus-Positional-Encoding [3]

[3] https://medium.com/@kavierim/transformers-from-scratch-part-1-input-embeddings-positional-encoding-bbce1f39040d

RoPE Ein weiterentwickelter Ansatz ist das Rotary Positional Encoding (RoPE) von Su et al. [24]. Hier werden Embeddings nicht durch trigonometrische Funktionen ergänzt, sondern mithilfe einer Rotationsmatrix im Vektorraum transformiert. Die Rotationswinkel hängen von den Positionen der Tokens ab.
Damit bleiben insbesondere bei langen Sequenzen relative Abstandsbeziehungen zwischen Tokens erhalten.
Im Gegensatz zu normalen Positional Encodings werden die Rotationswerte nicht Anfangs zu den Embeddings addiert, sondern direkt in den Attention-Mechanismus integriert.

ALiBi Attention with Linear Biases (ALiBi) Positionscodierungen [21] verwenden einen anderen Ansatz als Positional Encodings und RoPE und passen nicht die Embeddings selber an. Stattdessen werden die sogenannten Attention Scores der Attention Berechnung der LLM modifiziert. Je weiter zwei Token voneinander entfernt sind, desto stärker wird deren Einfluss aufeinander in der Berechnung der Attention durch einen Bias reduziert.

2.2.2 Attention Mechanismus

Im Kern moderner LLMs steht der Attention-Mechanismus. Ziel ist es, Abhängigkeiten zwischen verschiedenen Tokens zu modellieren. Der Attention Mechanismus passt die Bedeutung eines Tokens abhängig von anderen Tokens des Input an. Beispielsweise kann das Wort 'Bank' je nach Kontext eine Sitzgelegenheit oder ein Finanzinstitut bedeuten. Der Attention-Mechanismus passt die Embeddings an, indem er die semantischen Informationen aller anderen Tokens berücksichtigt. Die Formel, die der Attention Mechanismus berechnet ist:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3}$$

Als Eingabe erhält der Mechanismus die Vektoren aus dem Input Embedding. Mithilfe erlernter Gewichtsmatrizen werden daraus die Matrizen Query (Q), Key (K) und Value (V) berechnet:

$$\begin{aligned} q_i &= x_i W_q \\ k_i &= x_i W_k \\ v_i &= x_i W_v \end{aligned}$$

Dabei ist:

- x_i: Das Embedding des i-ten Tokens
- W_q, W_k, W_v: erlernte Gewichtsmatrizen
- q_i, k_i, v_i: daraus resultierende Query-, Key-, und Value-Vektoren

Die Matrix Q enthält die Eingabetokens und ist entsprechend der Inputlänge groß. Die K und V Matrizen repränsentieren die Kontextinformation der vorherig betrachteten Tokens.
Das Produkt $QK^\top/\sqrt{d\_k}$ ergibt die sogenannten Attention-Scores. Eine Matrix, die angibt, wie relevant die Tokens füreinander sind. Das Ergebnis der Berechnung ergibt eine Attention Matrix A die für jedes Token eine Wahrscheinlichkeitsverteilung beinhaltet. Element A_ij gibt an, wie stark das Token i Informationen von Token j bezieht. Die Summe aller A_ij für ein i ist dabei 1. Abbildung 3 zeigt eine Visualisierung solcher Attention-Scores in einer Übersetzungsaufgabe. Dicke Linien repräsentieren dabei stärkere Zusammenhänge: Mit einer Softmax-Funktion werden aus den Attention-Scores die entsprechenden Attention-Gewichte berechnet. Diese stellen eine Wahrscheinlichkeitsverteilung über die Attention-Scores dar.
Um jedem Token eine genaue Bedeutung zuzuweisen, werden die Attention-Scores anschließend mit der V-Matrix multipliziert.
Das Ergebnis des Attention-Blocks nach der Multiplikation der K-, V- und Q-Matrizen ist eine kontextabhängige Codierung jedes Tokens.

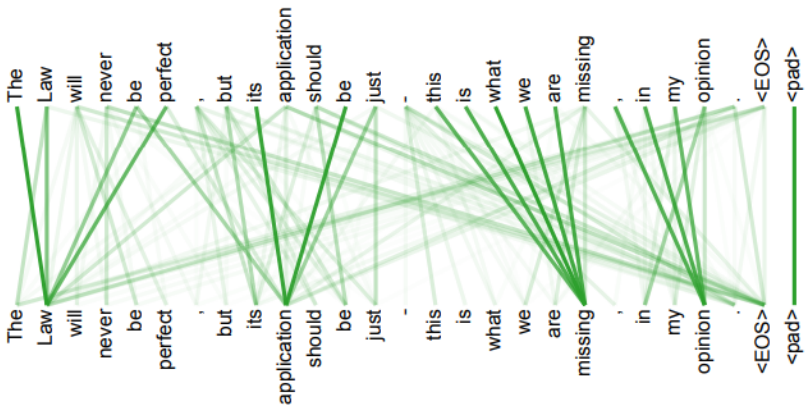


Abbildung 3: Visualisierung der Attention-Scores. Quelle: [26]

Masking In der Attention-Berechnung wird Masking durchgeführt. Der Grund dafür ist, dass Tokens, die eine spätere Position im Input-Text haben, keinen Einfluss auf Tokens in früheren Positionen nehmen sollen. Ein Token soll nur durch vorherige andere Tokens beeinflusst werden und nicht in die Zukunft sehen. Um das zu erreichen, werden manche Werte der Attention-Scores auf negativ unendliche Werte gesetzt, damit diese nach der Softmax-Ausgabe nicht berücksichtigt werden. Abbildung 4 zeigt die Attention-Scores mit durchgeführtem Masking.

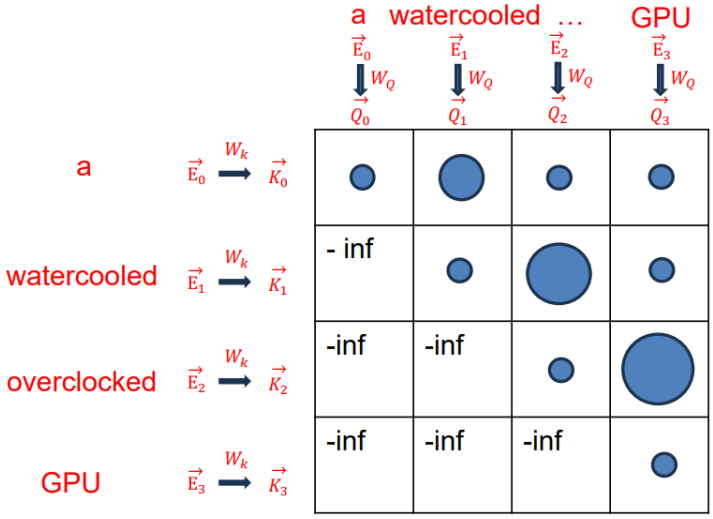


Abbildung 4: Darstellung einer Attention-Matrix. Quelle: [4]

KV Cache Die Berechnung des Attention-Mechanismus ist durch die Multiplikation großer Matrizen aufwändig. Kernziel des KV-Caches ist es, die Berechnung der K- und V-Matrizen zwischenzuspeichern und sie nicht wiederholt durchführen zu müssen. Die Q-Matrix wird dabei nicht gespeichert, da sie das neu zu verarbeitende Token repräsentiert und sich in jedem Inferenzschritt verändert. Die in dieser Arbeit betrachteten autoregressiven Decoder-Architekturen erzeugen Text, indem sie ein neu generiertes Token wieder an den Input-Prompt anhängen und die Inferenz erneut berechnen. Würde kein Caching genutzt werden, müsste die Berechnung der K- und V-Matrizen für frühere Tokens in jedem dieser Schritte erneut durchgeführt werden. Bei langen Texten würde die Rechenzeit dadurch extrem steigen. Möglich ist dieses Zwischenspeichern, da frühere Tokens durch das Masking nicht durch spätere Tokens beeinflusst werden und ihre Werte nicht anpassen müssen. Bei absoluten Positional Encodings haben neue Tokens, die hinzukommen, auch keinen Einfluss auf frühere Embeddings. In Abbildung 5 veranschaulichen die Autoren der Arbeit [4] den Ablauf in einem autoregressiven LLM mit einem KV-Cache, am Beispiel einer Textgenerierungsaufgabe. Angenommen, der ursprüngliche Eingabesatz lautet 'Happy Birthday', dann erzeugt

[4] https://hpc.fau.de/files/2025/03/2025-03-11_HPCCafe_LLMfuerDummies.pdf

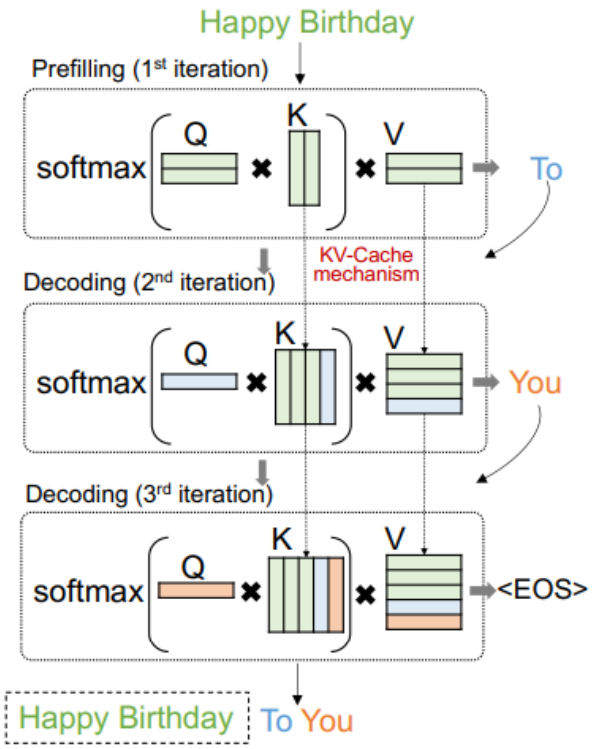


Abbildung 5: Funktionsweise des KV-Caches in autoregressiven Decodern. Quelle: [4]

das Modell in der ersten Iteration das Wort 'To'. In der zweiten Iteration wird erneut 'Happy Birthday' als Eingabe in das Modell gegeben. Nach dem zuvor beschriebenen Selbstaufmerksamkeitsmechanismus müsste nun wieder die Attention zwischen 'Happy', 'Birthday' und 'To' berechnet werden. Allerdings wurden die Aufmerksamkeitsgewichte zwischen 'Happy' und 'Birthday' bereits in der ersten Iteration berechnet. Diese Wiederholungen lassen sich vermeiden, indem zuvor berechnete Key- und Value-Werte zwischengespeichert und wiederverwendet werden.
Aufgrund des KV-Cache-Mechanismus muss der gesamte Input-Text nur in der ersten Iteration übergeben werden. In den folgenden Iterationen, der sogenannten 'Decoding-Phase', genügt es, nur das jeweils zuletzt erzeugte Token als Eingabe zu verwenden. [4]

2.2.3 Feed Forward Netz

Das Ergebnis des Attention-Mechanismus sind Embeddings, die an den Kontext des Satzes angepasst wurden und so die wirkliche Wortbedeutung wiederspiegeln.
Im zweiten Teil des Layer einer Decode-Only LLM-Architektur gehen die Embeddings in ein Feed Forward Netz (FFN). Dort werden die Embedding Informationen jedes Token weiterverarbeitet und transformiert. Der FFN-Block führt seine Berechnung dabei in einer höheren Dimension als die der Embeddings der Attention Schicht durch. Hierzu werden die Vektoren zunächst von der Modelldimension auf die Dimension d_ffn erweitert.
In dieser höheren Dimension können Kontextinformationen besser erfasst werden, was zu präziseren Vorhersagen über das nächste Token führt. Nach den Berechnungen wird der Output des FFN über eine Down-Projektion wieder auf die für die Ausgabe erforderliche Embedding-Dimension reduziert. Aufgrund der großen Anzahl an Gewichtsmatrizen in der d_ffn Dimension ist der FFN-Block in der Regel rechenintensiver als der Attention-Block. Abbildung 6 zeigt den FFN-Block, der die Vektoren der Attention-Schicht in einer vierfach höheren Dimension verarbeitet. Nach der Multiplikation der Gewichte im FFN wird eine Aktivierungsfunktion angewendet. Diese fügt dem neuronalen Netz Nichtlinearität hinzu und ermöglicht es, komplexe nichtlineare Zusammenhänge in den Daten zu erfassen. In Deep-Learning-Anwendungen wird hierfür standardmäßig die ReLU-Funktion verwendet. Sie gibt den Einga-

[5] https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mixture-of-experts

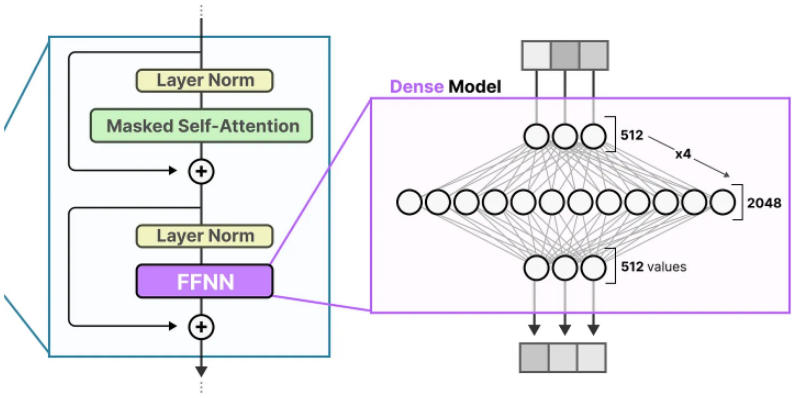


Abbildung 6: Projektion des FFN Blocks. Quelle: [5]

bewert zurück, wenn dieser positiv ist, und sonst den Wert null:

$$f(x) = \max(0, x)$$

Die meisten modernen LLMs verwenden anstelle der ReLU-Funktion komplexere Aktivierungsfunktionen wie GELU oder SiLU. Die ReLU-Funktion besitzt für negative Eingabewerte einen Gradienten von null, wodurch einzelne Neuronen während des Trainings des neuronalen Netzes absterben und nicht weiter aktualisiert werden. Dieses Problem wird durch die GELU [15] und die SiLU-Aktivierungsfunktion umgangen, die den Verlauf der ReLU-Funktion glätten. Abbildung 7 zeigt den Verlauf der GELU-Funktion (orange) im Vergleich zur ReLU-Funktion (blau).

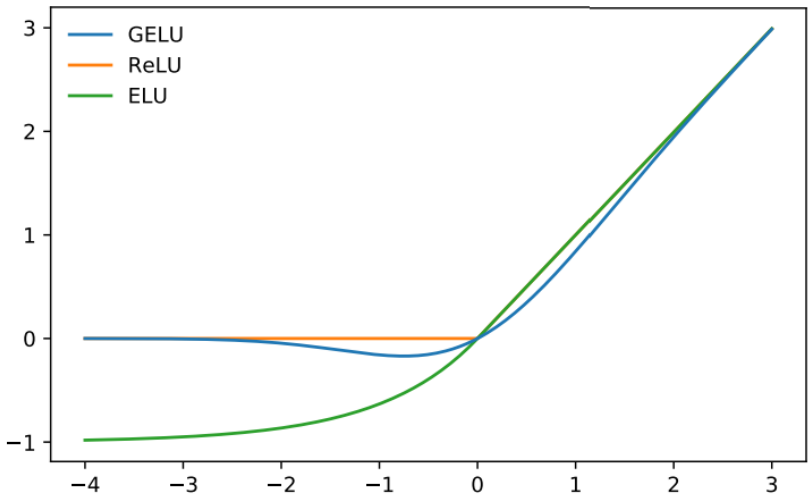


Abbildung 7: Vergleich der ReLU-, GELU- und ELU-Funktionen. Quelle: [15]

2.2.4 Sampling

Nach der Berechnung aller Layer der LLM wird aus den angepassten Embeddings eine Vorhersage darüber getroffen, welches Token mit der höchsten Wahrscheinlichkeit auf den gegebenen Eingabetext folgt. Dazu wird der Ausgabevektor mit der ursprünglichen Vokabularmatrix multipliziert.
Die daraus resultierende Matrix wird mithilfe der Softmax-Funktion in eine Wahrscheinlichkeitsverteilung überführt, die für jedes Token angibt, wie wahrscheinlich es auf den gegebenen Inputtext folgen sollte. Hierbei wird eine angepasste Softmax Funktion verwendet, die die Exponenten durch einen Wert T, die sogenannte Temperatur teilt:

$$P(w_i) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

Durch Variation der Temperatur lässt sich die Form der Wahrscheinlichkeitsverteilung steuern. Ein Wert T<1 verstärkt die Unterschiede zwischen den Wahrscheinlichkeiten, während ein Wert T>1 diese abschwächt und die Verteilung glättet. Bei einer Temperatur nahe null werden die Werte so stark skaliert, dass das wahrscheinlichste Token einer Wahrscheinlichkeit von 100 Prozent ausgewählt wird. Dies führt zu einer deterministischen Greedy-Strategie.
Um aus den Ergebnisse der Softmax Funktion das nächste Token zu wählen existieren mehrere Strategienm, sogenannte Sampling Methoden:

- **Greedy:** Wählt immer das Token mit der höchsten Wahrscheinlichkeit.
- **Top-K:** Wählt zufällig aus den besten K Tokens aus
- **Top-P:** Wählt zufällig aus einer Gruppe an Tokens (Nucleus) aus. Dafür werden die Tokens absteigend nach Wahrscheinlichkeit sortiert und so viele hinzugefügt, bis ein Grenzwert (zB 95 Prozent) erreicht sind.

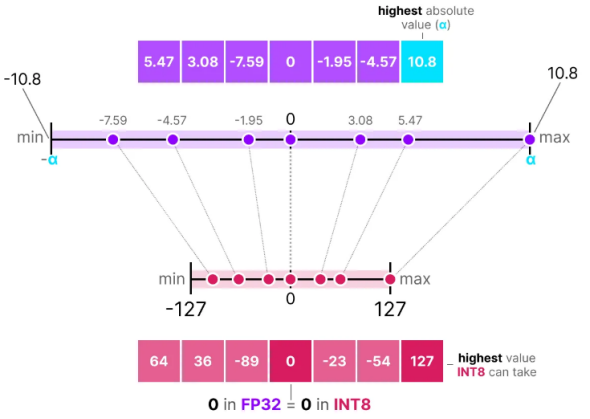
Eine Greedy-Strategie kann durch die deterministische Auswahl zu unkreativen Antworten führen. Für Tests und Anpassungen wurde eine Greedy-Strategie bzw. eine Temperatur von 0 gewählt, um Ergebnisse miteinander vergleichen zu können.
Die Sampling-Strategien werden nicht vom Sprachmodell, sondern von der ausführenden Software gewählt. Das in dieser Arbeit verwendete `llama.cpp`-Projekt unterstützt die Greedy-, Top-K- und Top-P sowie weitere Strategien.

2.2.5 Quantisierung

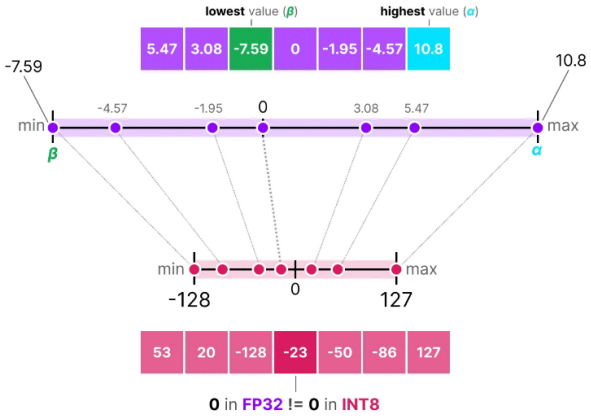
Ein Problem moderner LLMs ist ihr Speicherbedarf. Die Modellgewichte, die oft mehrere Gigabyte umfassen, sollten für eine effiziente Verarbeitung im schnellen Speicher der Hardware, vor allem dem VRAM von GPUs, gespeichert werden. Passen die Gewichte nicht vollständig in diesen Speicher, müssen sie während der Inferenz nachgeladen werden, was zu hohen Latenzen führt. Vor allem für den Anwendungsfall der Ausführung von Modellen auf ressourcenbeschränkten Geräten wie FPGAs, ist die Reduzierung des Speicherbedarfs wichtig.
Durch eine sogenannte Quantisierung der Gewichte lassen sich diese in ein Datenformat konvertieren, das weniger Bits benötigt, beispielsweise von einem Float mit 32 Bits zu einem Int mit nur 4 Bits.
Die Werte werden dafür mit einem berechneten Skalierungsfaktor auf den kleineren Wertebereich des Zieldatentyps abgebildet. Für das in dieser Arbeit verwendete Modell Qwen3 mit 0.6B Parametern verkleinert sich der Speicheraufwand beispielsweise von 1.1 GB auf 358 MB, wenn die Gewichte von einer FP16-Genauigkeit zu nur 4 Bit quantisiert werden.
Mithilfe des Skalierungsfaktors lassen sich die Werte auch wieder zurück in das genauere Datenformat dequantisieren. Diese Konvertierung ist jedoch nicht verlustfrei und führt zu einer niedrigeren Performance der LLM, der ursprüngliche Wert kann nicht exakt wiederhergestellt werden. Bei der Quantisierung von Modellgewichten findet also ein Trade-off zwischen Modellgröße und der Genauigkeit des Modells statt.
Kann die Zielhardware nicht direkt mit den quantisierten Werten arbeiten, müssen die Gewichte außerdem während der Inferenz zurück auf eine höhere Genauigkeit konvertiert werden.
Diese Dequantisierung kostet zusätzliche Rechenzeit [6].
Werte eines Tensors können symmetrisch oder asymmetrisch quantisiert werden. Bei symmetrischer Quantisierung werden die Tensorwerte von einem großen Wertebereich auf einen kleineren Wertebereich projiziert, der symmetrisch um den Nullwert liegt. Der Wert 0 wird dabei auf 0 quantisiert. Die asymmetrische Quantisierung projiziert die Werte zwischen den kleinsten und größten Wert des Tensors. Dafür müssen die Werte zusätzlich um einen Nullpunkt verschoben werden, um den gesamten Ziel-Datenbereich optimal auszunutzen. [7] Für beide Quantisierungsformen wird erst ein Skalierungsfaktor berechnet mit dem die Tensorenewerte dann multipliziert werden.
Bei der symmetrischen Quantisierung berechnet sich der Skalierungsfaktor s sich aus dem maximalen absoluten Wert des Eingabetensors X :

$$s = \frac{2 \cdot \max(|X|)}{2^n - 1} \cdot c$$

[6] `https://arxiv.org/html/2406.17415v1`
[7] `https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization`
[8] `https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization`



(a) Symmetrische Quantisierung



(b) Asymmetrische Quantisierung

Abbildung 8: Vergleich der symmetrischen und asymmetrischen Quantisierung. Quelle: [8]

Dabei ist n die Bitbreite der Quantisierung und c ein Clipping-Faktor. Der quantisierte Tensor $\bar{X}$ wird berechnet durch:

$$\bar{X} = \text{clamp}\left(\left\lfloor \frac{X}{s} \right\rceil, -2^{n-1}, 2^{n-1} - 1\right)$$

Für die asymmetrische Quantisierung werden der Skalierungsfaktor s und der Nullpunkt z wie folgt bestimmt:

$$s = \frac{\max(X) - \min(X)}{2^n - 1} \cdot c, \quad z = \left\lfloor \frac{-\min(X)}{s} \right\rceil$$

Hierbei ist X wieder der Eingabetensor und c der Clipping-Faktor, der verwendet wird, um den Dynamikbereich zu reduzieren und die Auswirkung von Ausreißern abzuschwächen. Die Elemente im quantisierten Tensor $\bar{X}$ berechnen sich durch:

$$\bar{X} = \text{clamp}\left(\left\lfloor \frac{X}{s} \right\rceil + z, 0, 2^n - 1\right)$$

Quelle: [20].
Für die Quantisierung der Modellgewichte wird meist die symmetrische Quantisierung verwendet, da diese einfacher umzusetzen ist und Modellgewichte meistens um den Nullpunkt zentriert sind. Asymmetrische Quantisierung wird dagegen für die Aktivierungen, also die Zwischenergebnisse der Inferenzberechnung, verwendet [10].

Namensgebung Quantisierte LLM-Modelle werden mit der Bezeichnung QN_Typ_Variante angegeben. Das Q steht dabei für Quantisierung und N für die Anzahl an bits die für den quantisierten Wert verwendet werden. Typ: 0 zeigt an, dass symmetrische Quantisierung verwendet wird, Typ:1 verwendet asymmetrische Quantisierung. Bei weiteren Typen wie K für K-quants Quantisierung gibt die Variante genauere Quantisierungsdetails an.

Blockqauntisierung In der sogenannten Blockquantisierung wird die Eingabematrix in Blöcke zB 32 oder 16 Werten aufgeteilt. Für jeden Block wird dann ein eigener Skalierungsfaktor und min/max berechnet und quantisiert.

Zeitpunkt Ein Modell kann nach dem Training oder schon davor quantisiert werden. Die Post-Training-Quantization (PTQ) führt die Quantisierung durch, nachdem das Modell trainiert wurde. Diese Methode ist die einfachere, da das Modelltraining nicht angepasst und nicht noch einmal durchgeführt werden muss. Der Genauigkeitsverlust ist jedoch höher, da das Modell während des Trainings nicht auf die Quantisierung reagieren kann. Das Quantization-Aware Training (QAT) wendet die Quantisierung während des Trainings an. Die Genauigkeit ist dadurch höher, der Trainingsprozess jedoch auch aufwändiger.

Literatur Ein behandeltes Thema der Literatur über Quantisierung sind sogenannte Outlier. Outlier bei der Quantisierung von LLM-Gewichten sind einzelne Parameterwerte, die deutlich größer oder kleiner als die Mehrzahl der Gewichte im Tensor sind. Wenn bei der Quantisierung der minimale und maximale Tensorwert zur Berechnung des Skalierungsfaktor verwendet werden, dominieren einzelne Ausreißer diese Berechnung. Dadurch wird der Skalierungsfaktor so gewählt, dass die viel Gewichte auf einen sehr engen Bereich im quantisierten Wertebereich abgebildet werden. So entsteht ein größerer Quantisierungsfehler. Die Autoren Dettmers et al. [7] untersuchen dieses Phänomen und finden heraus, dass die Anzahl der Outlier bei großen Modellen relevant ist. Bei untersuchten Modellen ab einer Parameterzahl ab 6.7 Milliarden treten die Outlier systematisch und vielen Layern auf.
Mehrere Arbeit bieten Lösungen für das Problem der Outliert. Gou et al [14] betrachten in einem Algorithmus die zu quantisierenden Werte als Wertepaar und verwerfen Werte, falls darin Outlier auftreten. Lin et al. [19] stellen ein Quantisierungsverfahren vor, das auf Hardwareeffizienz ausgelegt ist und für die lokale Ausführung von LLMs auf ressourcenbeschränkter Hardware gedacht ist. Eine Haupterkenntnis ist, dass nur ein sehr kleiner Teil der Gewichte, <1 Prozent, notwendigerweise in FP16-Genauigkeit bestehen muss, während sich der Rest der Gewichte ohne Qualitätseinbussen quantisieren lässt.
Die Autoren gehen auch auf den Aufwand der De-Quantisierung in SIMD-Architekturen ein. In einem selbst erstellten Inferenz-Tool werden die quantisierten Werte vor der De-Quantisierung umgeordnet, um SIMD-Operationen besser zu nutzen. Durch diese Ansatz werden drei SIMD-Instruktionen benötigt um 32 Gewichte zu De-Quantisieren, anstatt drei Skalaren Instruktionen pro Gewicht.
Eine weitere Optimierung der Quantisierung ist die Weight-Activation-Quantization. Bei dieser Methode werden nicht nur die Modellgewichte, sondern auch die Aktivierungen während der Inferenz quantisiert. Für die mathematischen Berechnungen, beispielsweise der Matrixmultiplikation, müssen die Werte dadurch nicht De-Quantisieren werden, sondern können direkt in der Darstellung mit wenigen Bits berechnet werden. Auf Hardware, die diese Operationen effizient unterstützt, hat dies einen Performancevorteil.

2.2.6 Inferenz-Phasen

Ein zentraler Aspekt, neben der Architektur und dem Aufbau von LLMs, ist der Ablauf der Inferenz. Die Textgenerierung in LLMs lässt sich in zwei Phasen unterteilen: die Prefill-Phase und die Decode-Phase. Der in dieser Arbeit betrachtete autoregressive Decoder erzeugt Text, indem er in jedem Inferenzschritt ein neues Token generiert, dieses an den bisherigen Input anhängt und anschließend als neuen Eingabetext nutzt.
In der Prefill-Phase verarbeitet das Modell den gesamten Eingabetext des Nutzers, bestehend aus mehreren Tokens, parallel. Dabei werden die K- und V-Matrizen befüllt. Die Prefill-Phase endet, sobald das erste Ausgabetoken generiert wurde. In der anschließenden Decode-Phase wird in jedem Schritt das zuvor erzeugte Token als neuer Input verwendet. Die Input-Matrix Q besteht hier in jedem Schritt nur aus einer Zeile, die dem zuletzt generierten Token entspricht. Das daraus erzeugte neue Token dient wiederum als Eingabe der nächsten Iteration. Die Decode-Phase läuft, bis entweder die maximale Anzahl zu generierender Tokens erreicht ist oder ein End-of-Sequence-Token auftritt. Abbildung 9 veranschaulicht den Ablauf der beiden Phasen. Vor allem Relevant für diese Arbeit ist die unterschiedliche Rechenauslastung der beiden Phasen. In der Prefill-Phase wird eine Eingabematrix der Größe [L, H] verarbeitet, wobei L der Länge des Nutzereingabetexts entspricht und H die Modelldimension bezeichnet. Die Decode-Phase führt die Inferenzoperationen dagegen schrittweise nur für Eingaben der Größe [1, H] aus , [1].
Durch die parallele Verarbeitung mehrerer Tokens ist die Prefill-Phase compute-bound, das heißt, sie erfordert hauptsächlich Rechenleistung. Die Decode-Phase hingegen ist aufgrund des ständigen Zugriffs auf die gespeicherten K- und V-Matrizen speicherintensiver und somit memory-bound. Ein Ziel der späteren

[9] `https://huggingface.co/blog/tngtech/llm-performance-prefill-decode-concurrent-requests`

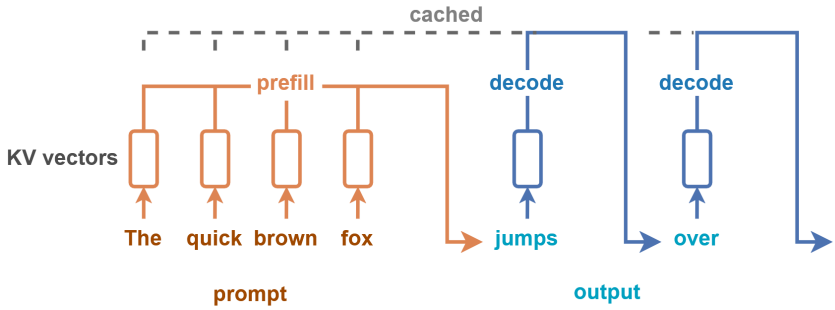


Abbildung 9: Ablaufdiagramm der Prefill- und Decode-Phase. Quelle: [9]

Betrachtung des Rechenaufwands ist es, die beiden Phasen zu analysieren, und aufzuzeigen wie sich deren Rechen und Speicheraufwand unterscheidet. So soll geklärt werden, welche Phase sich besser für das Auslagern auf ein FPGA eignen kann. Generell hat die Prefill Phase durch das parallele Verarbeiten mehrerer Tokens eine höhere GPU Auslastung. Die Decode Phase ist dagegen memory bound. Es soll untersucht werden wie stark dieser Effekt, vor allem bei kleinen Modellen mit wenigen LLM-Gewichten, ist.

2.3 Optimierungen

Die bisher beschriebenen Komponenten sind die Grundbestandteile einer LLM. Durch Forschungsarbeit der Letzten Jahre hat sich die Standardarchitektur an mehreren Stellen weiterentwickelt. Wichtige Optimierungen sollen im Folgenden beschrieben werden.

2.3.1 MHA MQA GQA

Multi Head Attention Im Paper Attention is all you need [26] wird die Attentionberechnung nicht nur in einem, sondern mehreren Attention-Mechanismen , sogenannten Heads, gleichzeitig durchgeführt. Der Textinput wird dabei nicht auf die Heads aufgeteilt, sondern von unterschiedlichen Heads parallel verarbeitet. Multi-Head-Attention (MHA) bietet gegenüber Single-Head Attention zwei wesentliche Vorteile:

- **Parallelisierung:** Die unabhängige Berechnung der Attention Heads ermöglicht parallele Verarbeitung, was Trainings- und Inferenzzeiten reduziert.
- **Erfassung komplexer Kontexte:** Jeder Attention Head kann sich auf unterschiedliche Aspekte der Sequenz konzentrieren und somit verschiedene Arten von Beziehungen zwischen den Tokens erkennen. Dies ermöglicht es dem Modell, ein umfassenderes Verständnis des Kontextes zu entwickeln, was besonders bei langen und komplexen Sequenzen wichtig ist.

Die Embeddingvektoren werden bei MHA durch erlernte Matrizen auf die jeweiligen Heads projeziert. Jeder der Attention-Heads hat dabei andere gewichtete Matrizen Wq,Wk und Wv. Dadurch extrahieren die Heads unterschiedliche Informationen aus den Embeddings. Die Ergebnisse der verschiedenen Attention-Heads werden anschließend zusammengefügt und durch eine Matrix W_0 in passende Form gebracht

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4}$$

MQA Noam Shazeer beschreibt in einer Arbeit von 2019 [23], dass in Multi-Head-Attention ein Bottleneck durch das Kopieren der Key- und Value-Matrizen entsteht. Die Inferenzgeschwindigkeit von LLMs wird vor allem durch das Laden der K- und V-Matrizen begrenzt. Bei großen Modellen und wachsenden Matrizen benötigen diese eine hohe Speicherbandbreite. In dem Ansatz der Multi-Query-Attention (MQA) teilen sich die verschiedenen Heads dieselben Key- und Value-Matrizen. Nur die Query-Matrix bleibt für jeden Head individuell. Somit lässt sich das benötigte Speichervolumen deutlich reduzieren. Tests mit Englisch-Deutsch-Übersetzungen zeigen, dass die Inferenz mit MQA um mehrere Größenordnungen schneller ist. Jedoch reduziert MQA auch die Qualität der Vorhersagen [23].

GQA Group-Query-Attention (GQA) [2] stellt einen Kompromiss zwischen MHA und MQA dar. In GQA werden die unterschiedlichen Heads Subgruppen zugeteilt. Anders als in MQA teilen sich nicht alle Heads eine K- und V-Matrix, sondern nur die Heads einer Subgruppe untereinander K- und V-Matrizen. Bei 8 Heads und 4 Subgruppen teilen sich je zwei Heads eine K- und V-Matrix. Abbildung 10 zeigt die Aufteilung der K- und V-Matrizen bei MHA, MQA und GQA.

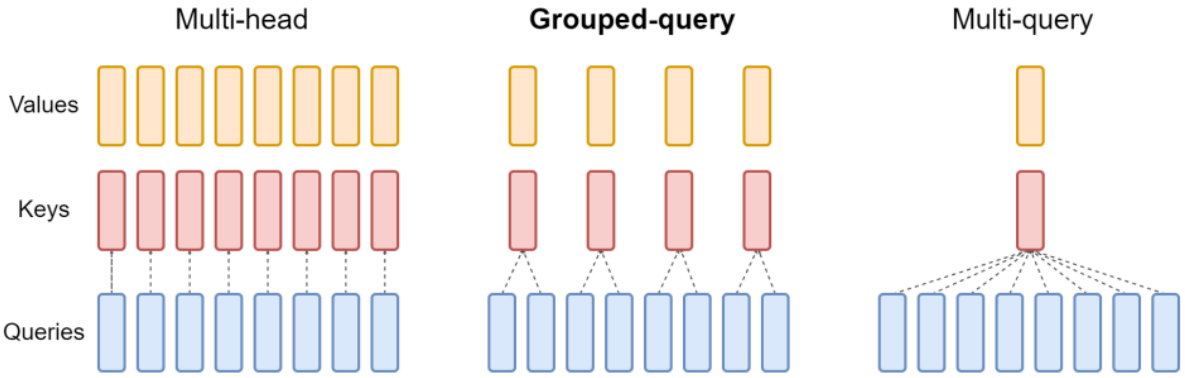


Abbildung 10: Vergleich von Multi-Head, Multi-Query und Grouped-Query Attention. Quelle: [2]

Vergleich Abbildung 11 zeigt einen Performancevergleich von MHA,MQA und GQA aus der Arbeit über Grouped-Query Attention [2]. Die gemessene Performance der drei Methoden ist eine Mittelwert über verschiedene LLM-Tasks wie die Zusammenfassung von Text oder Übersetzungen. Die Messungen wurden an dem öffentlich zugänglichem T5-Modell [10] mit den unterschiedlichen Attention Verfahren durchgeführt. Die Ergebnisse zeigen, dass MHA die größte Performance erreicht, jedoch auch die längste Laufzeit benötigt. MQA bietet die schnellste Inferenz, hat von den drei Methoden aber auch die schlechteste Performance. GQA bietet hier einen Tradeoff und erreicht fast die gleiche Performance wie MHA bei einer deutlich schnelleren Inferenz. Das in dieser Arbeit genauer analysierte LLM-Modell Qwen3 verwendet GQA.

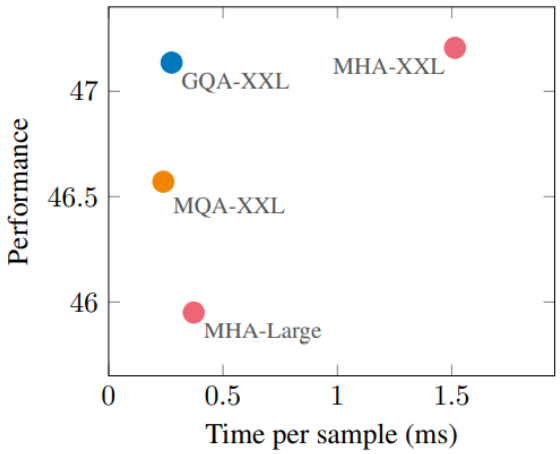


Abbildung 11: Performancevergleich von MHA, MQA und GQA. Quelle: [2]

2.3.2 Flash Attention

Eine Weiterentwicklung des Attention-Mechanismus ist FlashAttention [5]. Der Rechen- und Speicheraufwand der klassischen Attention-Berechnung steigt mit zunehmender Größe der K- und V-Matrizen. Um diese Effizienzprobleme zu adressieren, schlagen die Autoren eine angepasste Berechnungsmethode vor, die speicherschonender arbeitet und damit die Performance von Transformer-Modellen verbessert.

[10] `https://huggingface.co/docs/transformers/model_doc/t5`

In ihrer Arbeit analysieren die Autoren die Speicherhierarchie von GPUs im Detail. Dabei liegt ein besonderer Fokus auf der Optimierung von Lese- und Schreiboperationen unter Berücksichtigung der unterschiedlichen Zugriffsgeschwindigkeiten der Speichertypen

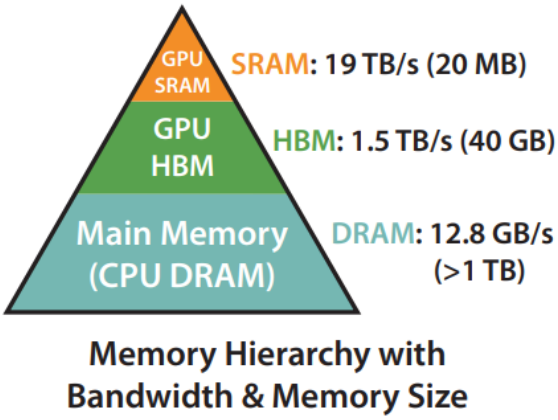


Abbildung 12: Schematische Darstellung der GPU-Speicherhierarchie. Quelle: [5]

Abbildung 12 veranschaulicht exemplarisch die Größenverhältnisse und Zugriffsgeschwindigkeiten verschiedener Speicherbereiche innerhalb einer GPU. Ziel des vorgestellten Algorithmus ist es, Zugriffe auf den vergleichsweise langsamen HBM (High Bandwidth Memory) möglichst zu vermeiden und stattdessen primär den schnellen SRAM (Static Random Access Memory) zu nutzen.
Ein zentrales Konzept von FlashAttention ist das sogenannte Tiling. Dabei wird die Attention-Matrix in kleinere Blöcke (Kacheln) unterteilt, die sequenziell im schnellen SRAM verarbeitet werden. Dies reduziert die Anzahl der Speicherzugriffe auf den HBM deutlich und verbessert somit die Gesamteffizienz der Berechnung. Die Blockgrößen sind definiert durch $B_c = \lceil \frac{M}{4d} \rceil$ und $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$, wobei M die verfügbare SRAM-Größe und d die Modelldimension darstellt. Dadurch wird sichergestellt, dass die jeweiligen Blöcke vollständig im SRAM abgelegt werden können, wodurch auf das Nachladen großer Matrizen aus dem HBM verzichtet werden kann [5]. Sowohl die Attention-Berechnung als auch die Anwendung der Softmax-Funktion erfolgt schrittweise auf Basis dieser Blöcke.

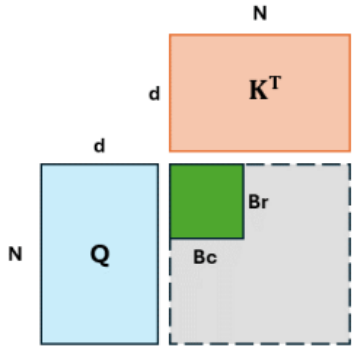


Abbildung 13: Darstellung des Tiling-Mechanismus. Quelle: [12]

Abbildung 13 zeigt exemplarisch die Verarbeitung kleinerer Tiling-Blöcke anstelle einer vollständigen Attention-Matrix.
Im Vergleich zur konventionellen Attention reduziert FlashAttention die Anzahl der HBM-Zugriffe um bis zu den Faktor 9 [5]. In experimentellen Messungen erzielten die Autoren eine bis zu dreifache Geschwindigkeitssteigerung gegenüber der PyTorch-Referenzimplementierung, die keine Optimierung der Speicherzugriffe vornimmt

2.3.3 Batching

Die Inferenz für eine Textanfrage, vor allem während der Decode-Phase, ist meist speichergebunden. Es muss eine große Zahl an Gewichten geladen werden, wobei der Berechnungsaufwand für den Input von

einem neuen Token vergleichsweise gering ist. Eine Optimierungstechnik, die den Durchsatz erhöht, ist das Batching. Beim Batching werden mehrere Anfragen gleichzeitig an das LLM gestellt, um diese parallel auszuführen. Die Speicherzugriffe und Ladezeiten für die Gewichte fallen dadurch pro Anfrage weniger stark ins Gewicht. Am meisten profitiert dabei das FFN-Netz. Die Werte der K-V-Caches sind für jede Anfrage individuell und können nicht geteilt werden. Abbildung 14 zeigt den steigenden Durchsatz mit höheren Batches. Der Test wurde von Guldogan et al. [13] mit einem Phi-3.5-Modell auf einer Nvidia-A100-Grafikkarte durchgeführt. Man kann zwischen zwei Arten des Batchings unterscheiden:

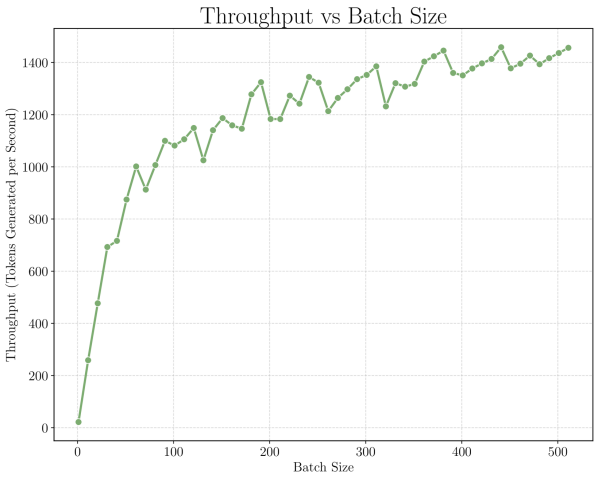


Abbildung 14: Erhöhter Durchsatz durch Batching. Quelle: [13]

- **Statisches Batching**: Verarbeitet eine vorgegebene Batchgröße. Geeignet für vorhersehbare Aufgaben. Beispiel: Gleichzeitiges Zusammenfassen mehrerer Dokumente.
- **Dynamisches Batching**: Gruppiert eingehende Anfragen in Echtzeit. Wird für Chatbots oder Suchsysteme verwendet.

2.3.4 MoE

Neue LLMs, wie etwa Qwen3, veröffentlichen neben den klassischen Transformer-Modellen auch sogenannte Mixture-of-Experts (MoE)-Modelle. MoE-Modelle stellen eine Erweiterung des klassischen Feed-Forward-Netzwerks (FFN) dar. Sie bestehen aus mehreren FFN-Teilnetzen, den sogenannten Experts, sowie einem Router, der für jede Eingabe den jeweils am besten geeigneten Experten auswählt. Bei der Inferenz wird dadurch nur ein Teil der gesamten Modellgewichte aktiviert.
Trotz der größeren Gesamtarchitektur mit mehr Parametern sind sowohl das Training als auch die Inferenz von MoE-Modellen schneller als bei klassischen, sogenannten 'dense' Modellen. Der Grund dafür ist, dass bei einem Durchlauf, sowohl im Training als auch bei der Inferenz, nur ein Teil des Netzes aktiv ist. Es müssen nicht alle Gewichte in der Berechnung verwendet werden. MoE-Netze benötigen bei gleicher Datenmenge weniger Rechenzeit, extrahieren mehr Informationen aus den Trainingsdaten und nutzen die vorhandenen Rechenressourcen effizienter [11].
Eine Herausforderung beim Training von MoE-Modellen besteht darin, die Experten gleichmäßig zu trainieren. Es kann vorkommen, dass nur bestimmte Experten regelmäßig aktiviert und trainiert werden, während andere kaum genutzt werden. Ein besseres Load Balancing während des Trainings kann erreicht werden, indem die zu trainierenden Experten teilweise zufällig ausgewählt werden oder indem für jeden Experten ein Maximum an Tokens pro Trainingsdurchlauf festgelegt wird. Diese zusätzlichen Techniken erhöhen jedoch die Komplexität des Trainingsprozesses im Vergleich zu klassischen dichten Modellen. Abbildung 15 zeigt einen Decoderblock mit Router und mehreren Experten. Neben den Vorteilen in Bezug auf Trainingseffizienz und Inferenzleistung weisen MoE-Modelle auch bedeutende Nachteile auf.MoE netzen benötigen einen Routing-Mechanismus, der bei jeder Anfrage bestimmt, welcher Experte genutzt wird. Dieser zusätzliche Berechnungsschritt führt zu einer zusätzlichen Latenz während der Inferenz.
Darüber hinaus besitzen MoE-Modelle aufgrund der Vielzahl an Experten eine deutlich höhere Anzahl

[11] https://arxiv.org/pdf/2410.05661
[12] https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mixture-of-experts

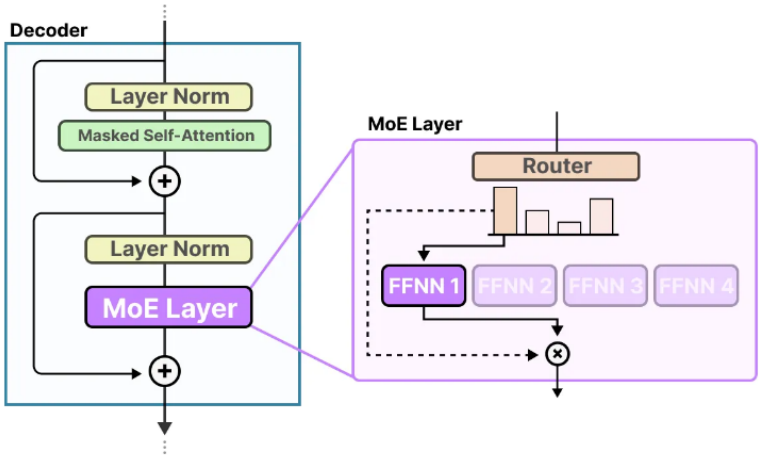


Abbildung 15: Darstellung eines MoE-Layers mit Router[12]

an Gewichten. Die Modelle sind dadurch insgesamt größer und benötigen mehr Speicherplatz. Vor allem aufgrund dieses erhöhten Speicherbedarfs werden MoE-Modelle in dieser Arbeit nicht verwendet. Für den Einsatz im Edge Computing auf einem FPGA werden kompakte Modelle benötigt, die mit den begrenzten Hardware-Ressourcen effizient arbeiten können.

2.4 Qwen3

Die **Qwen3** Modellfamilie wurde im April 2025 veröffentlicht und umfasst mehrere Modellgrößen von 0.6B bis 32B Parametern [22]. Neben den Basis Modellen gibt es auch spezialisierte Varianten die gezielt auf Coding Aufgaben oder auf das Erstellen von Text Embeddings trainiert worden sind [13].
In dieser Arbeit wurde das Qwen3 Modell mit 0.6B Parametern verwendet. Dieses besonders kleine Modell eignet sich durch den niedrigen Speicheraufwand für das Ziel der Berechnung auf einem FPGA.
Eine Performanceevalution aus dem Technischen Report von Qwen3 [22] zeigt, dass das Modell in mehreren Tests besser ist als vergleichbar kleine Modelle wie Gemma3-1B von Google oder vorherige Modelle von Qwen wie Qwen 2.5

Architektur Das Qwen3 0.6B-Modell besitzt eine Decoder-only-Architektur. Es umfasst 28 Layer, eine Sequenzlänge von 32.000 Tokens und eine Embedding-Dimension von 1.024 [14]. Qwen3 verwendet ein Vokabular von rund 150.000 Tokens. Zusammen mit den Modellgewichten benötigt die Variante mit einer Q4-Quantisierung der Tensoren etwa 400 MB Speicherplatz. Für den Attention-Mechanismus kommt Grouped-Query Attention (GQA) in Kombination mit einem KV-Cache zum Einsatz. Die Positionsinformationen der Tokens werden mittels RoPE kodiert. Als Aktivierungsfunktion wird die SiLU-Funktion (Sigmoid Linear Unit) verwendet. Aufgrund der kompakten Modellgröße wird bei Qwen3 0.6B kein Mixture-of-Experts (MoE) verwendet, das Feed-Forward-Netzwerk eines Layers ist dense aufgebaut. Bei der Nutzung des Modells mit `llama.cpp` kann die Sampling-Temperatur für die Textgenerierung frei eingestellt werden. Abbildung 16 zeigt eine schematische Übersicht der Qwen-Architektur. Jedes Layer besteht aus einem Attention-Block und einem Feed-Forward-Block (FFN). Die Darstellung verdeutlicht neben den zuvor beschriebenen Architekturdetails auch den Ablauf zentraler Operationen, etwa die Anwendung der RoPE-Positionskodierung nach der Erzeugung der K- und Q-Matrizen, sowie die Residual Connections innerhalb beider Blöcke.
In dieser Arbeit wurde die Q-4 Quantisierte Variante des Modells verwendet.

[13] `https://qwen.ai/blog?id=1e3fa5c2d4662af2855586055ad037ed9e555125&from=research.research-list`
[14] `https://huggingface.co/Qwen/Qwen3-Embedding-8B`

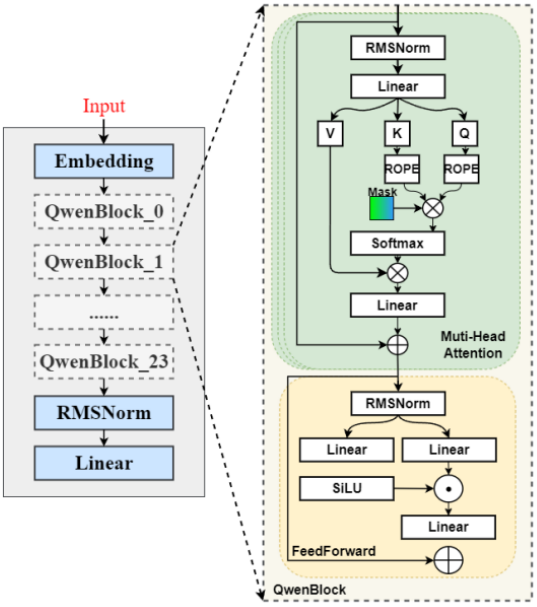


Abbildung 16: Schematische Ansicht eines Qwen3 Layers. Quelle: [27]

3 Rechenaufwand der Inferenz

3.1 LLaMA-Cpp

Um LLMs auszuführen, wird neben den Gewichten des Modells noch ein Inferenz-Framework benötigt [15]. Dieses übernimmt im Wesentlichen die Verarbeitung der Text Ein- und -Ausgabe sowie die Durchführung der mathematischen Berechnungen während der Inferenz. Beispiele hierfür sind vLLM oder ExL2. In dieser Arbeit wird das öffentlich zugängliche Tool `llama.cpp` verwendet [16].
Das `llama.cpp`-Projekt ist ein populäres und weit verbreitetes Framework zur lokalen Ausführung von LLMs. Es ist auf GitHub öffentlich verfügbar und wurde überwiegend von Georgi Gerganov entwickelt. Mit `llama.cpp` lassen sich eine Vielzahl von LLM-Architekturen ausführen. Darüber hinaus bietet das Projekt einen breiten Hardwaresupport, sowohl für die CPU- als auch die GPU-Ausführung, und ist auch mit Android-Geräten kompatibel. `llama.cpp` unterstützt zahlreiche GPU-Backends, darunter CUDA, OpenCL, Vulkan und Metal. Für den Anwendungsfall dieser Arbeit ist `llama.cpp` geeignet, da es leichtgewichtig ist und stark auf die jeweils verwendete Hardware optimiert wurde.
Das `llama.cpp`-Projekt befindet sich derzeit noch in aktiver Entwicklung und wird regelmäßig erweitert und angepasst. In dieser Arbeit wurde `llama.cpp` in der Version des Commits 814f795e analysiert und verwendet [17].

Modelle können in `llama.cpp` mit verschiedenen Tools ausgeführt werden:

- **CLI-Tool:** Eine Kommandozeilenschnittstelle zur direkten Textinferenz und Modellinteraktion.
- **HTTP-Server:** Eine REST-API, über die Anfragen an das Modell gesendet und Antworten automatisiert verarbeitet werden können.
- **Python-Bindings:** Ermöglichen die Integration des Modells in Python-Skripte.
- **C/C++-Bibliothek:** Die Kernfunktionen des Frameworks können direkt aus C- oder C++-Code aufgerufen werden.

In dieser Arbeit wurde das CLI-Tool von `llama.cpp` verwendet, um die Textausgabe der Modelle zu testen.
Die Modelle, die mit `llama.cpp` ausgeführt werden, müssen im GGUF-Format vorliegen. GGUF ist ein binäres Speicherformat, das speziell für das schnelle Laden und Speichern von Modellen optimiert wurde.

[15] `https://hpc.fau.de/files/2025/03/2025-03-11_HPCCafe_LLMfuerDummies.pdf`
[16] `https://github.com/ggml-org/llama.cpp`
[17] `https://github.com/ggml-org/llama.cpp/commit/814f795e063c257f33b921eab4073484238a151a`

Darüber hinaus unterstützt es verschiedene Quantisierungstypen, die es ermöglichen, Modelle in unterschiedlicher Genauigkeit und Größe zu speichern.`llama.cpp` beinhaltet Skripte, mit denen bestehende Modelle aus dem verbreitetem GGML-Format in das GGUF-Format konvertiert werden können.

Um auch auf Systemen mit begrenzten Ressourcen effizient arbeiten zu können, unterstützt `llama.cpp` eine Vielzahl an Quantisierungsmethoden, Multithreading sowie weitere CPU-Optimierungen. Das Projekt ist vollständig in C/C++ implementiert und kommt ohne Frameworks wie PyTorch oder TensorFlow aus.
Beim Aufruf von `llama.cpp` können mehrere Optionen angegeben werden. Neben dem Modell und dem Input prompt Text lässt sich das Verhalten der LLM-Berechnung anpassen:

- **LLM-Parameter**
  - –top-k - Top-K Sampling (Standard: 40)
  - –top-p - Top-P/Nucleus Sampling
  - –repeat-penalty
  - –rope-freq-base
- **Hardwareoptionen**
  - -ngl, –n-gpu-layers - Anzahl Layer auf GPU auslagern
  - –tensor-split - Tensor-Verteilung auf mehrere GPUs
  - –main-gpu - Haupt-GPU ID auswählen
  - -t, –threads - Anzahl der CPU-Threads
  - –mlock - Modell im RAM sperren
- **Art der Textgenerierung:**
  - –interactive
  - –instruct

Das `llama.cpp`-Projekt lässt sich in zwei Hauptkomponenten gliedern.
Einen Teil für `llama.cpp` selbst und einen für das in `llama.cpp` stark verwendete GGML-Framework. Im Folgenden werden beide Komponenten näher beschrieben. Dabei wird insbesondere dargestellt, welche Teile der Software für die Ausführung der Inferenz relevant sind und welche Komponenten angepasst werden können, um die Modellausführung zu verändern oder spezifische Tests durchzuführen.
In dieser Arbeit wurde `llama.cpp` beispielsweise so modifiziert, dass sich die Anzahl der Layer sowie die Anzahl der Inferenzschritte anpassen lassen. Das GGML-Framework wurde verwendet, um die zugrunde liegenden mathematischen Operationen zu analysieren und um diese an die eigene Hardwareumgebung anzupassen.

3.1.1 Aufbau

Für das Arbeiten mit `llama.cpp` sind drei Teile wichtig:

- Die Hauptmethode, die den Input Text einliest und die Inferenz aufruft
- Das Erstellen eines Berechnungsgraphen für das jeweilige Modell und dessen Komponenten
- Der Decode Aufruf, der die Inferenzberechnung für das nächste Token durchführt und für seine Berechnungen GGML verwendet.

Die Hauptmethode des `llama.cpp`-Projekts befindet sich in der Datei `tools/main/main.cpp`. Dort werden die Parameter, vor allem das LLM-Modell und der Input-Prompt, eingelesen. In einer Schleife wird das Verhalten eines autoregressiven Decoders umgesetzt.
`llama.cpp` ruft die Inferenz wiederholt auf, bis ein Abbruchkriterium erreicht ist. Anschließend wird der Text an den Nutzer ausgegeben. Die Decode-Methode ist in der Datei `context.cpp` definiert.
In dieser wird zunächst der Modellgraph für das jeweilige Modell erstellt, der die nötigen Berechnungsschichten enthält. Dafür wird die Datei `llama_model.cpp` aufgerufen, die für jedes von `llama.cpp` unterstützte Modell den Graphen erstellt. In `llama_model.cpp` sind Berechnungsgraphen für alle von

`llama.cpp` unterstützten Modelle definiert. Nachdem in der Decode-Methode der Graph erstellt wurde, wird `graph_compute` aufgerufen, um den Operatorbaum auszuführen bzw. die Inferenz mit den Eingabetensoren zu berechnen. Hierfür wird die GGML-Bibliothek verwendet.
Abbildung 17 zeigt den Kontrollfluss der wichtigsten Methoden von `llama.cpp` im Zusammenspiel mit GGML.

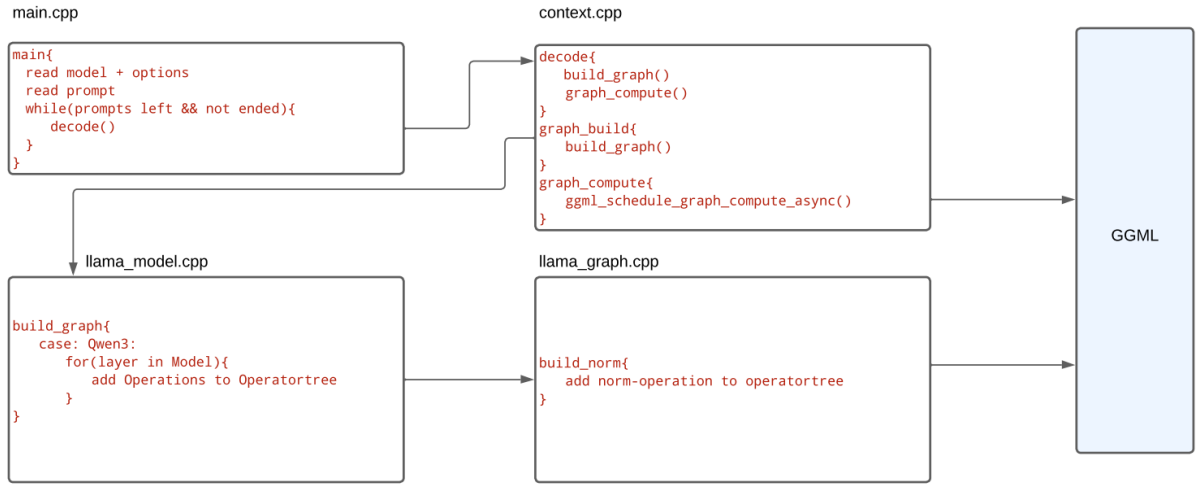


Abbildung 17: Übersicht über den Codeablauf von `llama.cpp` und GGML

Speichern der Modellparameter Wird die Berechnung nicht voll auf der CPU ausgeführt, muss der Kontext und die darin enthaltenen Tensoren auf den Speicher des Backend verschoben werden. Dies geschieht in der Methode `common_init_from_params(params)`, die in der Main Methode von `llama.cpp` aufgerufen wird. In deren Call-Hierarchie wird später die Methode `load_tensors` aus der Datei `llama-model.cpp` aufgerufen. Dort werden in einer switch-Klausel für das gewählte Modell die benötigten Tensoren erstellt und die Methode `load_all_data` aufgerufen, die die Tensoren auf das Backend Gerät verschiebt.

3.1.2 GGML

Um die eigentlichen Berechnungen durchzuführen verwendet `llama.cpp` die GGML(Georgi Gerganov Machine Learning)-library. GGML ist eine low-level C/C++-Bibliothek die für die effiziente Inferenz von `llama.cpp` ausgelegt ist.
GGML wird auch von anderen Inferenz libraries wie ollama oder GPT4All verwendet. Die GGML-library führt die im Operatorbaum erstellten mathematischen Berechnungen aus. Berechnungen wie Tensormultiplikation sind in GGML hardwärenäher als in Frameworks wie Pytorch. Die Implementierungen der angebotenen mathematischen Funktionen sind stark auf die jeweilige Hardware und vorliegende Datentypen optimiert. GGML bietet Code für verschiedene GPU Backends wie CUDA, OpenCL oder Vulkan an. In dieser Arbeit wurde ausschließlich die OpenCL Implementierung beachtet und verwendet. Nach der Dokumentation wurde die GGML-OpenCL Implementierung vor allem für Qualcomm Adreno GPUs erstellt, kann aber auch auf bestimmten Intel GPUs laufen. GGML-OpenCL unterstützt Q_4 und Q6_K Quantisierung, ist aber nur auf Q_4 optimiert.

Datenstrukturen Für die genaue Umsetzung der Berechnung sind die verwendeten Datenstrukturen relevant. Die wichtigsten werden im Folgenden beschrieben.

Operatorbaum Die Ausführung der Inferenz geschieht über einen Operatorbaum. Die Knoten des Baums sind bei der Ausführung der inferenz die durchzuführenden mathematischen Operationen des LLM-Modells. Die Kanten sind die Tensorinputs für die nächste Operation. Der Operatorbaum spiegelt die LLM Architektur wieder. Abbildung 18 zeigt einen Ausschnitt des Operatorbaums der Attention-Berechnung. Der Operatorbaum ist in GGML statisch und wird vor dem Decode-Aufruf einmal erstellt. In anderen Deep Learning Frameworks wie PyTorch wird der Operatorbaum während der Berechnung dynamisch erstellt.

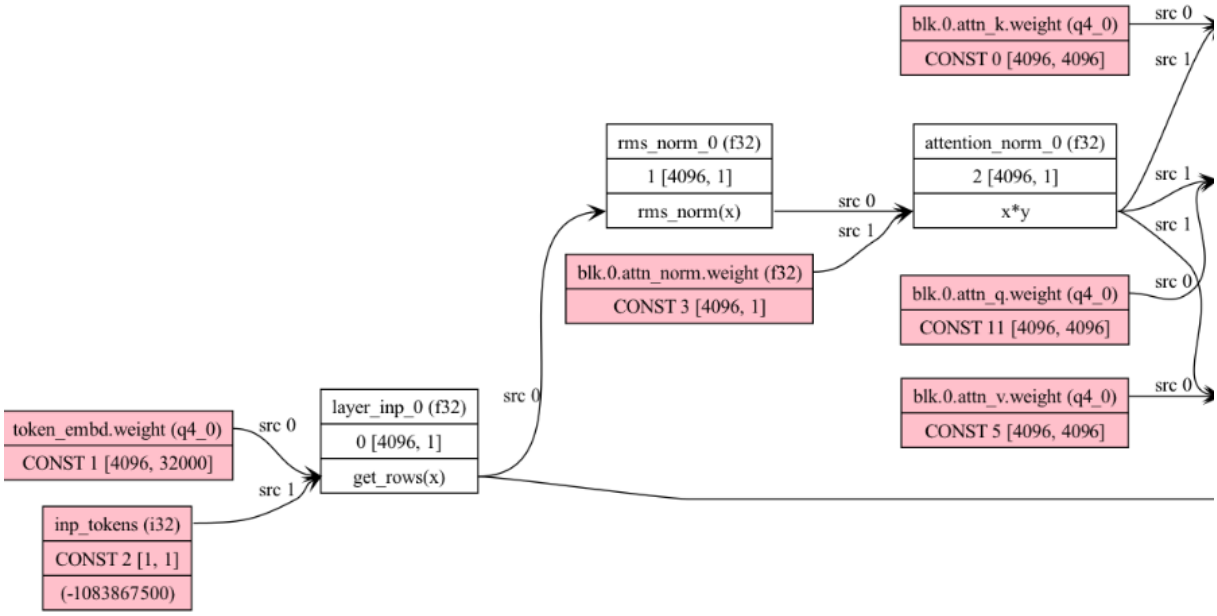


Abbildung 18: Operatorbaum der Berechnung der Self-Attention [26]

Graph Print Das GGML-Framework bietet eine Methode zur Visualisierung des Operatorbaums an, der während der Modellerstellung aufgebaut wird. Die Funktion `ggml_graph_dump_dot` befindet sich in der Datei `llama_context` und ist im Quellcode standardmäßig auskommentiert. Wird diese Methode aktiviert und aufgerufen, erzeugt sie eine `.dot`-Datei, mit der sich ein Bild des Baumes erstellen lässt. Dieses Dot-File kann allerdings nicht für die gesamte LLM-Architektur erstellt werden. In ersten Versuchen trat bei der Generierung des vollständigen Graphen ein Fehler auf. Vor dem erstellen muss die Anzahl der erstellten Layer in `llama_model` auf ein Layer reduziert werden.

```
for (int il = 0; il < 1; ++il) {
// for (int il = 0; il < n_layer; ++il) {
```

Das resultierende Dot-File kann anschließend mit dem folgenden Befehl in eine grafische Darstellung umgewandelt werden:

```
dot -Tpng graph.dot -o graph.png
```

So entsteht ein PNG-Bild des Operatorbaums, das die interne Struktur des jeweiligen Layers visualisiert. Abbildung 18 zeigt einen Ausschnitt des für das Qwen3-Modell generierten Operatorbaums.

GGML Tensor Der zentrale Datentyp für die Darstellung von Tokens und die Durchführung von Berechnungen in LLMs ist der Tensor. Tensoren in GGML können sowohl Vektoren als auch Matrizen darstellen [16]. In GGML wird ein Tensor als `struct` implementiert, das mehrere Attribute enthält. Diese Attribute, insbesondere die Größeninformationen und Speicherlayouts, werden in den mathematischen Operationen intensiv genutzt und sind daher wichtig für das Verständnis der internen Berechnungsabläufe.

Feld	Beschreibung	Feld	Beschreibung
type	Datentyp der Elemente	src	Zeiger auf Quell-Tensoren
backend	Ausführungs-Backend	data	Zeiger auf Rohdaten
n_dims	Anzahl der Dimensionen	name	Optionaler Debug-Name
ne	Anzahl Elemente pro Dimension		
nb	Stride in Bytes pro Dimension		
op	Erzeugende Operation		

Tabelle 1: Beschreibung der Felder in der `ggml_tensor` Struktur

Das Feld *data* enthält entweder das Ergebnis des Tensors oder ist *null*, wenn noch kein Ergebnis vorliegt. Beim Aufruf von *ggml_compute_forward()* werden die Operationen des Operatorbaums ausgeführt, und der Ergebnistensor wird mit einem Pointer befüllt, der die berechneten Daten enthält. Das Attribut *n_dims* gibt die Dimension des Tensors an und kann zwischen 1 und 4 liegen.

Dimensionen Die Arrays *ne* und *nb* beschreiben Anzahl und Speicherabstand der Elemente pro Dimension. *ne* gibt die Anzahl der Elemente je Dimension an, während *nb* den Speicherabstand zwischen den Elementen in Byte enthält.
In `llama.cpp` ist die Länge von *ne[]* und *nb[]* fest auf 4 gesetzt. Sind weniger Dimensionen erforderlich, werden die nicht genutzten auf 1 gesetzt. In den Berechnungen der untersuchten Modelle wurden höchstens drei Dimensionen verwendet.
Die Dimensionsangaben des `ne` arrays sind anders herum als in anderen LLM Frameworks wie PyTorch.
Bei einem dreidimensionalen Tensor enthällt:

- *ne[0]* die Breite einer Zeile (Anzahl der Spalten)
- *ne[1]* die Anzahl der Zeilen pro Ebene und
- *ne[2]* die Anzahl der Ebenen

Quelle: [16] Das Array *nb* gibt für jede Dimension den Speicherabstand zwischen den Elementen an. *nb[0]* enthält den Abstand zwischen den Spalten. *nb[1]* enthält den Abstand zwischen den Zeilen und berechnet sich aus aus der Menge an Spalten in einer Zeile mal der Elementgröße *ne[0]*nb[0]*. Durch die Angabe der Tensordimensionen und Längen lässt sich direkt auf Elemente zugreifen. In Abbildung 19 wird ein Beispiel für die Felder nb und ne gegeben.

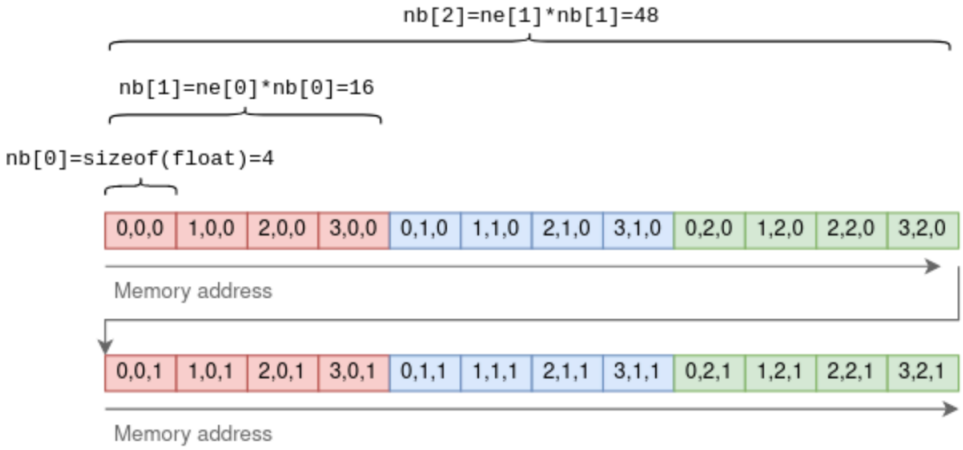


Abbildung 19: Dimensionsangaben nb und ne eines dreidimensionalen Tensors. Quelle:[16]

Context Daten werden in GGML in einem Kontext-Objekt gepseichert. Sowohl die Tensoren als auch der Operatorbaum. Es konnte zum aktuellen Zeitpunk keine offizielle Dokumentation von GGML gefunden werden, ein User analyisiert jedoch in einem detaillierten Blogbeitrag die Speicherverwaltung von GGML [18]. Das ggml_context Objekt enthällt eine Größenangabe und einen Pointer auf den gepseicherten Inhalt. In dem Kontext werden object-structs gepseichert, die jeweils einen Tensor oder Operatorbaum enthalten. Die Objekt-Structs enhalten neben den Speicher und Typangaben(Tensor, Operatorbaum) einen zeiger auf das nächste Objekt im Kontext, dadurch entsteht im Kontext eine Linked list. Abbildung 20 zeigt das Speicherlayout des GGML-Kontext der zwei Objekte mit Tensoren enthällt.
Das Reservieren von Speicher für den Operatorbaum innerhalb des Kontextes ist komplexer als für die Tensoren. Es wird Speicher reserviert um einen Grafen der Grösse GGML_DEFAULT_GRAPH_SIZE = 2048 zu speichern. Dafür wird Speicherplatz für Metadaten, Pointer auf mögliche Tensoren und eine Hashtabelle allokiert. Anschließend lässt sich der Operatorbaum mit den in Abbildung 17 beschriebenen Operationen befüllen.

[18] `https://xsxszab.github.io/posts/ggml-deep-dive-ii/`
[19] `https://xsxszab.github.io/images/ggml-deep-dive-II/graph5.png`

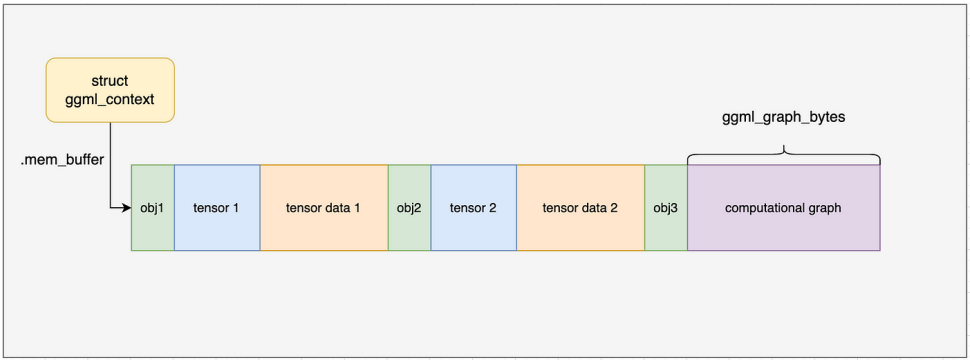


Abbildung 20: Speicherlayout des GGML-Kontext. Quelle: [19]

Ausführung Die beschriebenen Datenstrukturen werden in GGML auf folgende Weise verwendet um Berechnungen durchzuführen [11]:

1. Allokieren eines `ggml_context`, um Tensordaten zu speichern.
2. Erstellen von Tensoren und Setzen der Daten.
3. Erstellen eines `ggml_cgraph` der alle Rechenoperationen erhählt
4. Rekursives abarbeiten des operatorbaumes bzw desen operatioenn.
5. Abrufen der Ergebnisse (Ausgabetensoren).
6. Speicher freigeben und beenden.

OpenCL In dieser Arbeit wurde die OpenCL Implementierung von GGML verwendet. Sowohl in der Codeanalyse als auch der späteren Portierung der Kernel auf ein FPGA. OpenCL ist ein offener Programmierstandard für die Programmausführung auf heterogenen Systemen. In OpenCl geschrieben Programme können auf verschiedenen Hardwaretypen (CPU,GPU,FPGA) laufen und Ressourcen Parallel nutzen. Die Hardware auf der der Code läuft wird von OpenCL abstrahiert. Anders als z.B. CUDA ist OpenCl ein offener Standard und wurde nicht für Hardware eines bestimmten Herstellers geschrieben. OpenCL kann dadurch für die LLM-Inferenz auf verschiedenen GPUs verwendet werden.
Um OpenCL für `llama.cpp` zu aktivieren, muss beim Kompilieren des Projekts die Build-Option `-DGGML_OPENCL=ON` gesetzt werden. Für Nicht-Adreno-GPUs wird zusätzlich empfohlen, die Option `-DGGML_OPENCL_USE_ADRENO_KERNELS=OFF` zu setzen. Mit der Option `--n-gpu-layers` beim Aufruf von `llama.cpp` kann angegeben werden, wie viele Layer auf die GPU ausgelagert werden. Die Anzahl der auf die GPU ausgelagerten Layer wird beim Start von llamacpp ausgeprinted. Werden die Layer der LLM nicht auf der GPU ausgelagert, wird für die Berechnungen standardmässig nicht OpenCL, sondern die CPU Implementierung verwendet.

Aufbau OpenCL-Code besteht aus zwei Teilen: dem **Host**- und dem **Kernel**-Code. Der Host-Code verwaltet die OpenCL-Umgebung. Er initialisiert die Plattform und die verwendeten Geräte, lädt den auszuführenden Kernel-Code, startet dessen Ausführung auf den Geräten und verwaltet den Speicher. Der Kernel-Code definiert die eigentlichen Berechnungen und wird in einer `.cl`-Datei geschrieben. Er nutzt den vom Host bereitgestellten Speicher. Die Kommunikation zwischen verschiedenen Geräten (z.B. CPU und mehreren GPUs) erfolgt in OpenCL in einem **Kontext**. Dieser Kontext bildet die Umgebung, in der der Code ausgeführt und synchronisiert wird. Um Kernel auszuführen, können innerhalb eines Kontextes Warteschlangen erstellt werden, in denen die Kernel an die Geräte übergeben und asynchron ausgeführt werden.

Work Groups und Items Die einzelnen Recheneinheiten, die aus dem Kernel-Code auf dem Gerät ausgeführt werden, nennt man **Work-Items**. Diese werden von einem Thread ausgeführt. Diese Work-Items werden in **Work-Groups** organisiert, die gemeinsam auf einem Rechenkern ausgeführt werden. Innerhalb einer Work-Group können die Work-Items synchronisiert und miteinander koordiniert werden. Abbildung 21 zeigt die Aufteilung einer Matrixberechnung auf Work Groups, in denen Work Items berechnungen durchführen. Um das Datenelement aus der Matrix zu finden auf dem ein WorkItem arbeiten

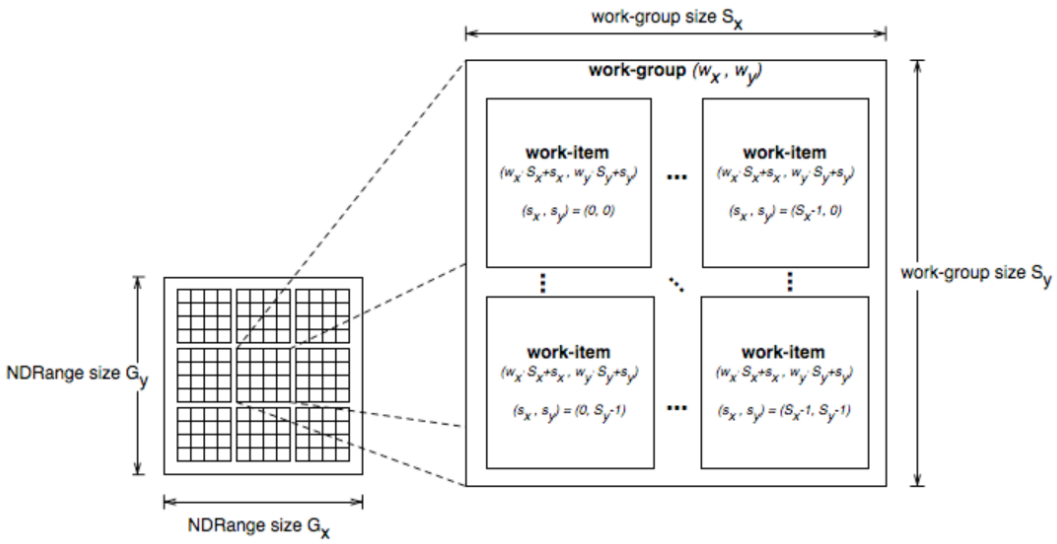
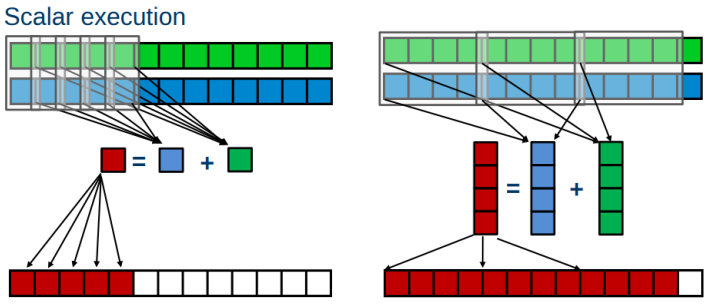


Abbildung 21: Work-Groups und Work-Items im OpenCL-Ausführungsmodell.[20]

soll, werden OpenCL Funktionen verwendet um IDs des WorkItems und der Work Group abzufragen:

- get_global_id: Position des Work-Items im gesamten NDRange
- get_local_id: Lokale ID in der lokalen Work-Group
- get_group_id: ID der Work-Group

SIMD/Subgroup Die OpenCL Kernel nutzen eine Vielzahl an Optimierungen um die teils aufwändigen Tensorberechnungen effizient auf der Hardware auszuführen. Eine davon ist SIMD(Single Instruction Multiple Data). In Rechenarchitekturen die SIMD unterstützen, kann eine Berechnung gleichzeitig auf mehrere Operanden angewendet werden. Die Rechenoperation verwendet dafür Register die mehr als nur einen Werte enthalten, sondern zb 4(AVX) oder 8 Werte(AVX512). Die gleichzeitige Ausführung macht die Tensorberechnung schneller, vor allem bei Parallelen berechnungen wie den Multiplikationskerneln. Abbildung 22 zeigt den Vergleich einer skalaren und einer SIMD Operation. Zu sehen ist, dass die SIMD Addition deutlich weniger sequentielle Schritte für die Berechnung benötigt. In vielen Sprachen wird



(a) Skalare Addition (b) SIMD Addition

Abbildung 22: Skalare Addition im Vergleich zur vektorisieren Ausführung mittels SIMD. [21]

SIMD entweder automatisch durch die Auto-Vektorisierung des Compilers genutzt oder explizit durch den Einsatz von Intrinsics bzw. datenparallelen Vektor-APIs. In den OpenCL-Kerneln wird SIMD durch die OpenCL-Funktion der Subgroups[22] verwendet. Die Subgroup-Funktionalität ist eine Erweiterung für

[20] https://www.researchgate.net/figure/OpenCL-execution-model-work-groups-depicted-as-groups-of-squares-corresponding-to_fig4_271848367
[21] https://moodle.nhr.fau.de/pluginfile.php/3967/mod_resource/content/1/08_SIMD.pdf
[22] https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/subGroupFunctions.html

OpenCL, die ab OpenCL-Version 1.2 speziell für Intel (`cl_intel_subgroups`) und ab OpenCL-Version 2.0 für plattformunabhängige Geräte (`cl_khr_subgroups`) verfügbar ist. Ab Version 2.1 sind Subgruppen im Standard-OpenCL enthalten. Die Subgroup-Funktionalität bildet eine weitere Schicht zwischen den Workitems und Workgroups. Subgroups existieren innerhalb einer Workgroup. Innerhalb dieser Gruppe kann auf die gleichen Daten zugegriffen werden, und SIMD-Funktionen können parallel auf den Daten ausgeführt werden. Abbildung 23 zeigt Subgroups innerhalb der OpenCL-Hierarchie. Die Größe einer

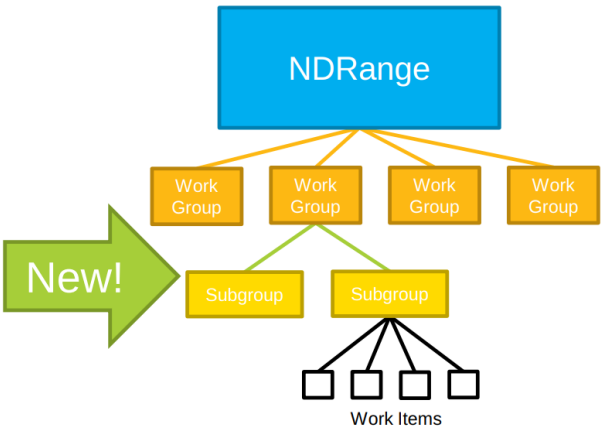


Abbildung 23: Einteilung der Subgroup in die OpenCL-Hierarchie Quelle: [23]

Subgroup, also wie viele Workitems sie enthält, ist abhängig von der Hardware und wird vom Compiler gewählt. Gängige Subgroup-Größen beinhalten 8, 16 oder 32 Workitems. Der Nutzer kann die Größe jedoch durch das Kernelattribut `intel::reqd_sub_group_size` manuell ändern [24].

Subgroup-Funktionen:

Um Informationen über Subgruppen auszulesen oder SIMD-Aufrufe zu nutzen, stellt OpenCL mehrere eingebaute Funktionen bereit. Diese Funktionen ermöglichen es, die zu berechnenden Daten aufzuteilen und zu bestimmen, welche Elemente der Tensoren von den jeweiligen Subgruppen verarbeitet werden sollen. Die Funktion `get_num_sub_groups(void)` gibt die Anzahl der Subgruppen an, in die die aktuelle Work-Group unterteilt ist. Im GGML-OpenCL-Kernel beträgt dieser Wert oft 1, d. h. jede Work-Group besteht aus genau einer Subgroup.

Die Funktion `get_sub_group_id(void)` liefert die ID der aktuellen Subgroup zurück, also eine Zahl im Bereich von 0 bis `get_num_sub_groups() - 1`.Mit `get_sub_group_local_id(void)` kann die eindeutige Work-Item-ID innerhalb der aktuellen Subgroup abgefragt werden. Die Zuordnung zwischen `get_local_id()` und `get_sub_group_local_id()` bleibt während der gesamten Lebensdauer der Work-Group unverändert.

Mit verschiedenen Subgroup-Funktionen lassen sich mathematische Operationen über SIMD-Aufrufe innerhalb der Workitems einer Subgroup effizient ausführen. Bei den Inferenzberechnungen in GGML dient dabei ein Array als Eingabe, das die Daten bzw. Zwischenergebnisse aller einzelnen Workitems einer Subgroup enthält.

- `float sub_group_reduce_add(float x)`
- `float sub_group_reduce_min(float x)`
- `float sub_group_reduce_and(float x)`

Die Shuffle-Funktion zum direkten Austausch von Daten zwischen Work-Items ist ausschließlich in der Intel-Erweiterung verfügbar. Durch ihren Einsatz können Daten effizient innerhalb einer Subgroup übertragen werden, ohne dass dafür der lokale Speicher verwendet werden muss[25].

Zusätzlich ermöglicht die Intel-Erweiterung optimierte Speicherzugriffe durch sogenannte Block-Reads. Dabei wird ein zusammenhängender Speicherbereich für die gesamte Subgroup in einem Zugriff gelesen, anstatt dass jedes Work-Item separat auf den Speicher zugreift. Dies reduziert den Adressierungsaufwand

[23] `https://www.iwocl.org/wp-content/uploads/iwocl2017-ben-ashbaugh-subgroups.pdf`

[24] `https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/sub-groups-and-simd-vectorization.html#SUB-GROUP-SIZE-VS-MAXIMUM-SUB-GROUP-SIZE`

[25] `https://registry.khronos.org/OpenCL/extensions/intel/cl_intel_subgroups.html`

auf eine einzelne Speicheradresse und erlaubt ein effizienteres Laden größerer Datenmengen[26].
Im Gegensatz zur Khronos-Erweiterung garantiert die Intel-Erweiterung jedoch nicht, dass Subgroups unabhängig voneinander ausgeführt werden. Subgroups können daher unter Umständen aufeinander warten müssen. In der Intel-Implementierung werden alle Work-Items einer Subgroup durch denselben Hardware-Thread (Vector Engine bzw. Execution Unit Thread) ausgeführt[27]. Dadurch sind sämtliche Work-Items innerhalb der Subgroup automatisch miteinander synchronisiert.

Kernel Relevant für die Analyse des Rechen und Speicheraufwands der Berechnung von einem LLM-Layer sind die OpenCL-Kernel und deren konkrete Umsetzung der mathematischen Funktionen. Für jede benötigte mathematische Operaiton einer LLM existiert mindestens ein Kernel. Für die meisten existieren jedoch mehrere, die auf die jeweilgen Datentypen und die verwendete Hardware optimiert sind. Für die Umsetzung der Matrixmultiplikation existieren beispielsweise 12 Dateien. Der konkrete Kernel wird im Host Code abhängig vom Datentyp und Hardware gewählt In dieser Analyse werden die Kernel betrachtet, die für die Inferenz mit dem Modell Qwen3 0.6B verwendet werden.

HostCode-Aufbau Der GGML-OpenCL-Code besteht aus der Datei `ggml-opencl.cpp`, in der der Host-Code implementiert ist, sowie einem Unterordner `kernels`, der die Kernelprogramme für die einzelnen Tensoroperationen enthält. Die Ausführung der Kernel wird vollständig durch den OpenCL-Host-Code gesteuert. Zunächst werden die Plattform und die verfügbaren Geräte ermittelt und die relevanten Geräteinformationen abgefragt. Anschließend wird ein Kontext erstellt, der die Ausführungsumgebung definiert, und eine Warteschlange eingerichtet, über die die Kernelbefehle an die Geräte übermittelt werden. Die Datenübertragung zwischen Host und Gerät erfolgt über OpenCL-Speicherpuffer (`cl_mem`), die Eingabe- und Ausgabedaten enthalten können.
In der Methode `load_Kernels` werden alle Kernel geladen, kompiliert und als Kernel-Objekte erzeugt. Dazu werden die Quelldateien geöffnet und mithilfe von `clCreateKernel` die entsprechenden Kernel initialisiert. In der hier verwendeten Version von GGML werden sämtliche Kernel vor Beginn der Inferenz erstellt und nicht erst zur Laufzeit.
Die Berechnung des Graphen erfolgt über die Methode `compute_forward`, die wiederholt für die einzelnen Knoten aufgerufen wird. Für jede im Inferenzprozess benötigte Operation existiert eine eigene Funktionsdefinition, zum Beispiel für Matrixmultiplikation oder Softmax. Falls mehrere Varianten eines Kernels für eine Operation vorhanden sind, wählt GGML automatisch diejenige aus, die zur verwendeten Hardwarearchitektur und zu den Datentypen passt. Die übergebenen Parameter werden dabei an die Dimensionen und Formate der Tensoren angepasst.

3.2 Berechnungsaufwand

Ein Hauptziel dieser Arbeit war die Untersuchung der Frage, unter welchen Bedingungen es sinnvoll ist, ein kleines lokales LLM von der CPU auf eine GPU bzw. ein FPGA auszulagern. GPUs bieten grundsätzlich erhebliche Vorteile bei der Inferenz von LLMs. Sie verfügen über eine große Anzahl kleiner Recheneinheiten, die insbesondere Tensor-Multiplikationen parallel ausführen können. Da sich die Berechnungen eines LLM gut parallelisieren lassen, profitieren solche Modelle stark von der Ausführung auf einer GPU oder FPGA. Ein Problem ist jedoch die benötigte Zeit, um die Gewichte der LLM auf den Hardwarebeschleuniger zu übertragen. Dieser Datentransfer kann je nach Umfang beträchtliche Zeit beanspruchen. Bei kleinen Modellen, insbesondere wenn nur kurze Prompts verarbeitet werden, stellt sich daher die Frage, ob sich die Auslagerung auf eine GPU tatsächlich lohnt. In solchen Fällen können die Vorteile der schnelleren Berechnung durch die hohen Transferkosten vollständig aufgehoben werden. Die schnelle Hardware wird nur genutzt wenn die Operationen durch fehlende Rechenkapazität begrenzt sind. Ein Ziel dieser Arbeit ist es, diesen Trade-off systematisch zu untersuchen. Hierfür werden der Rechenaufwand und der Speicherbedarf der LLM-Inferenz im Rahmen des Projekts `llama.cpp` sowie dessen OpenCL-Implementierung analysiert. Das LLM, das in dieser Arbeit betrachtet wird, ist das beschriebene Qwen3 0.6B-Modell. Es soll für jede mathematische Berechnung analysiert werdem, wie viele Floating Point Operationen (FLOPs) diese benötigt. Durch diese Analyse lässt sich in einem späteren Schritt die Rechenintensität mit der Datennutzung vergleichen und prüfen ob sich schnellere Hardware lohnt.

[26] `https://www.iwocl.org/wp-content/uploads/iwocl2017-ben-ashbaugh-subgroups.pdf`
[27] `https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/sub-groups-and-simd-vectorization.html`

3.2.1 Vorgehensweise

Um den Rechenaufwand zu bestimmen und rechenintensive Komponenten der Inferenz zu identifizieren wurde für jeden verwendeten OpenCL-Kernel der teilweise komplexe Code zunächst beschrieben und anschließend hinsichtlich des Rechenaufwands untersucht. Dafür wurden die im Code enthaltenen FLOPs bestimmt. Zur Analyse des Rechenaufwands wurde der benötigte Umfang an FLOPs aus dem Code abgeleitet und die Ergebnisse anschließend mit experimentellen Messungen verglichen. Diese FLOP-Messungen wurden mit dem an der FAU entwickelten Tool LIKWID durchgeführt. Zur Bestimmung der theoretischen FLOPs wurde der jeweilige Kernel-Code im Detail analysiert und ausschließlich die auftretenden Floating-Point-Operationen gezählt. Die Anzahl der FLOPs wurde dabei zunächst pro Work-Item ermittelt. Sie hängt von der Größe des Inputs ab, insbesondere von der Dimension ne00, da eine Workgroup jeweils auf einer Zeile arbeitet, während jedes Work-Item mehrere Spalten verarbeitet.

LIKWID Um zu überprüfen, ob die zuvor theoretisch bestimmten FLOPs korrekt sind, sollten die tatsächlich während der Ausführung eines Kernels auftretenden FLOPs gemessen werden. Für diese Hardwaremessungen wurde das Tool `LIKWID` verwendet, das von der RRZE-HPC-Gruppe der FAU entwickelt wurde [28]. Das Tool benötigt keine externen Bibliotheken und erlaubt es, mit vergleichsweise einfachen Befehlen die ausgeführten FLOPs einer OpenCL-Berechnung zu erfassen. Neben der FLOP-Messung selbst können mit LIKWID außerdem die maximale Rechenleistung (in FLOPs) sowie die Speicherbandbreite eines Systems bestimmt werden, was insbesondere für eine spätere Roofline-Analyse relevant ist.
Die von einem Kernel ausgeführten FLOPs lassen sich beispielsweise mit folgendem LIKWID-Befehl messen:

```
likwid-perfctr -C 0 -g FLOPS_SP -m ./flop_test
```

Dieser Befehl wurde sowohl für die Messung einzelner Kernel als auch die Messungen an `llama.cpp` vorgenommen. Die Option `-C` pinnt die Codeausführung an einen Thread um auf diesem die Messung durchzuführen. Die Messgruppe `FLOPS_DP` die Anzahl von Double-Precision-Operationen an, während `FLOPS_SP` für Single Precision steht. In der Ausgabestatistik erscheinen neben den FLOPs auch weitere Kennzahlen wie Laufzeit, Zyklen pro Instruktion (CPI), unterschiedliche FLOP-Typen (beispielsweise Double Precision oder SIMD-Operationen) sowie die Vektorisierungsrate. Die Ausgabe der Anzahl der FLOP zahlen wird in den Messungen in Folgender Tabelle dargestellt:

Instruktionstyp	**Events**	**FLOPs**
FP_ARITH_128B	1	4
FP_ARITH_SCALAR	1	1
FP_ARITH_256B	1	8
FP_ARITH_512B	1	16
Gesamt		**29**

Tabelle 2: Darstellung der Ergebnisse der FLOP-Messungen

Für den Instruktionstyp FP_128B zählt jedes Event 4 FLOP, Skalar 1 FLOP, FP_256B 8 FLOP und FP_512B 16 FLOP.
Um einen spezifischen Bereich einer Ausführung zu messen, wird im OpenCL-Hostcode mit LIKWID-Markern die gewünschte Messregion definiert: Zunächst wurde versucht, die LIKWID-Messungen direkt innerhalb des normalen Ablaufs von `llama.cpp` durchzuführen, indem die entsprechende Kernelausführung mit LIKWID-Markern versehen wurde. Dies war jedoch nicht erfolgreich. Zwar ließ sich LIKWID in das Makefile von `llama.cpp` integrieren und auch innerhalb des Projekts verwenden, allerdings konnten die OpenCL-Kernels von `llama.cpp` nur auf der GPU genutzt werden. Mit dem in dieser Arbeit eingesetzten Rechner war es nicht möglich, LIKWID-Messungen auf der GPU auszuführen, die verwendete Hardware unterstützt ausschließlich CPU-Messungen.
Wenn hingegen ein Layer von `llama.cpp` auf der CPU ausgeführt wird, greift das Projekt nicht auf die OpenCL-Implementierung zurück, sondern verwendet stattdessen eine speziell optimierte CPU Implementierung, die sich stark von der OpenCL-Variante unterscheidet und daher für diese Arbeit nicht geeignet war.
Als Lösung wurden gesonderte Hostcode-Dateien erstellt, mit denen sich die OpenCL-Kernels auf der CPU ausführen und mit LIKWID messen ließen. Dabei wurden als Eingabeparameter vergleichbare Größen

[28] `https://github.com/RRZE-HPC/likwid/wiki`

Listing 1: Befehle für eine Likwid Messung

```
LIKWID_MARKER_INIT();

// Messregion starten
LIKWID_MARKER_START("compute_kernel");

// Der zu messende Kernel
compute_kernel(a, b, c, n);

// Messregion beenden
LIKWID_MARKER_STOP("compute_kernel");
```

wie im `llama.cpp`-Projekt verwendet.
Für die Abschätzung der von einem einzelnen Work-Item benötigten FLOPs wurden in der Dimension 0 gleich viele WorkItems wie im llamacpp Projekt gestartet. Die anderen Dimensionen für die Zahl der Attention Heads und Batching wurden auf 1 gesetzt. Anschließend wurde die gesamte Anzahl ausgeführter FLOPs durch die Zahl der Work-Items dividiert. Diese Methode lieferte in ersten Messungen realistischere Ergebnisse als das Ausführen der Berechnung mit nur einem Workitem

Speicheraufwand Die Messung des Speicheraufwands war mit LIKWID auf dem verwendeten Rechner nicht möglich. Daher wurde der erforderliche Speicherbedarf der einzelnen Kernel ausschließlich durch Codeanalyse bestimmt. Betrachtet wurden die benötigten Ein- und Ausgabetensoren sowie der private Speicher der Work-Items für temporäre Zwischenspeicherungen während der Berechnung. Der insgesamt für die Modellausführung auf der GPU reservierte Speicher wird beim Start von `llama.cpp` ausgegeben. Dadurch lässt sich der Speicherverbrauch sowohl für das gesamte Modell als auch isoliert für einen einzelnen Layer auslesen.

Annahmen Die Messungen mit LIKWID wurden auf einem Rechner mit Intel-CPU und integrierter Grafikeinheit durchgeführt. Das Qwen3 0.6B-Modell kam in einer Q4-Quantisierung zum Einsatz. In den Messungen fiel auf, dass aufwändige mathematische Funktionen, insbesondere die Exponentialfunktion, einen hohen FLOP-Bedarf verursachen. Die genaue Anzahl erforderlicher FLOPs hängt von der gewählten Implementierungsgenauigkeit und der verwendeten Hardware ab. Für das Kompilieren der OpenCL-Kernel wurden die gleichen Compiler-Flags für mathematische Optimierungen wie in `llama.cpp` verwendet:

`-cl-mad-enable -cl-unsafe-math-optimizations -cl-finite-math-only -cl-fast-relaxed-math`

Für die theoretische Abschätzung wurde der FLOP-Bedarf solcher Funktionen aus Messungen übernommen. Für die Exponentialfunktion werden 240 FLOPs für `float4` und 195 FLOPs für `float` angenommen. Die pow() Funktion benötigt 608 FLOPs. Die Funktionen `min()` und `max()` werden dabei nicht als FLOPs gezählt.

3.2.2 Qwen3 Kernel

Im Folgenden werden die für die Inferenz mit dem Qwen3 Modell verwendeten Kernel analysiert.

Add In GGML-OpenCL existieren zwei Kernel für die Addition von Tensoren: `kernel_add` und `kernel_add_row`. `kernel_add` kann für die allgemeine Addition zweier Tensoren verwendet werden, die auch nicht zusammenhängend im Speicher liegen. `kernel_add_row` ist effizienter, kann aber nur Tensoren addieren, die kontinuierlich in einer Zeile liegen. In der Ausführung von Qwen3 wird `kernel_add` nur in der Prefill-Phase verwendet, in der Decode-Phase nur noch `kernel_add_row`.
`kernel_add`:
Führt Tensoraddition mit flexiblem Broadcasting über alle Dimensionen durch. Jedes Work-Item bearbeitet mehrere Elemente in einer Schleife und verwendet komplexe Adressberechnung für beliebige Tensorformen. Unterstützt nicht zusammenhängende Speicherlayouts durch stride-basierte Indizierung.
`kernel_add_row`:

Führt vektorisierte Addition für Zeilen-Broadcasting durch. Jedes Work-Item bearbeitet genau vier Elemente parallel mit `float4`-Vektoren. Verwendet optimierte Modulo-Berechnung und setzt eine zusammenhängende Speicheranordnung voraus.

Theoreitsche Flop

- kernel_add: (ne0/Workgroup-size) x 1 FLOPs pro Workitem
- kernel_add_row: 4 FLOPs pro Workitem

Cpy Zum Kopieren von Daten verwendet Qwen3 0.6B die Kernel:

- `kernel_cpy_f32_f16`
- `kernel_cpy_f32_f32`

Die Copy-Kernel übernehmen das Umkopieren von Tensor-Daten im globalen Speicher. Dabei ist jedes Work-Item für einen Teilbereich des Tensors zuständig. Über Integer-Berechnungen werden die mehrdimensionalen Indizes des aktuellen Work-Items in lineare Speicheradressen übersetzt. Anschließend lädt das Work-Item die Quellwerte aus dem Eingabepuffer und schreibt sie an die entsprechende Position im Zielpuffer. Im Fall von `cpy_f32_f32` erfolgt eine direkte Kopie von 32-Bit-Fließkommazahlen, während `cpy_f32_f16` zusätzlich eine Konvertierung in 16-Bit-Werte durchführt, um den Speicherbedarf zu reduzieren.

Theoretische FLOPs Beide Kernel verwenden keine FLOPs, führen jedoch mehrere Integer-Operationen zur Berechnung von Speicheradressen aus.

RMS-Norm Als Normierungsfunktion verwendet Qwen3 die RMS_Norm. Die *Root Mean Square* (RMS)-Norm ist eine spezielle Form der Norm, bei der der Betrag eines Vektors durch den Quadratmittelwert seiner Komponenten bestimmt wird. Für einen Vektor $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ ist sie definiert als

$$\|x\|_{\mathrm{RMS},\varepsilon} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \varepsilon}$$

Epsilon wird verwendet, um sehr kleine Werte und negative Ergebnisse in der Normierung zu vermeiden. In GGML ist die Funktion im Kernel `rms_norm` implementiert. Beim Aufruf erhält der Kernel den Eingabevektor inklusive Metadaten, den Epsilon-Wert sowie ein Array zur temporären Zwischenspeicherung von Werten. Abbildung 24 zeigt das Berechnungsschema des `rms_norm`-Kernels und die Arbeitsaufteilung innerhalb einer Subgruppe. Der Eingabetensor wird zu `float4` konvertiert. Dieser Datentyp fasst jeweils vier `float`-Werte in einem Arrayeintrag. Jede Work-Group bearbeitet eine Tensorzeile.
Jedes Work-Item quadriert anhand seiner ID die ihm zugeordneten `float4`-Werte und schreibt diese in `sumf`. Die quadrierten Werte werden innerhalb einer Subgruppe mit der SIMD-Funktion *reduce_add* aufsummiert und im Array `sum` abgelegt. `sum` wird dem Kernel als Puffer übergeben und enthält je Subgruppe einen Eintrag. Innerhalb der Work-Group werden die in `sum` gespeicherten Werte anschließend aggregiert und als Gesamtsumme in `sum[0]` abgelegt.
Aus der Gesamtsumme wird der RMS-Wert berechnet, mit dem die Elemente normiert und in `dst` geschrieben werden. Für Elemente am Zeilenende, die kein vollständiges `float4` bilden, erfolgt eine Restbehandlung, sowohl bei der Summenbildung als auch beim Schreiben nach `dst`.

Theoretische FLOPs Die Berechnung umfasst zwei Schritte: die Bestimmung der Wurzel der Quadratsumme (RMS-Wert) und die anschließende Normierung aller Einträge durch diesen Wert.
Eine Zeile des Tensors wird parallel von einer Work-Group bearbeitet. In einer Schleife verarbeitet jedes Work-Item jeweils einen `float4`-Vektor mehrfach, konkret
(# float4-Elemente pro Zeile)/(# Work-Items pro Work-Group) Iterationen.
Pro Iteration werden die vier Skalare des `float4` quadriert und akkumuliert. Bei Nutzung von SIMD lassen sich die Operationen auf alle Komponenten des `float4`-Typs gleichzeitig anwenden[29]. Andernfalls erfolgt die Verarbeitung komponentenweise. Die theoretische FLOP-Anzahl ergibt sich damit aus der Summe der Multiplikationen und Additionen pro `float4` über alle Iterationen der Work-Items.

[29] `https://www.intel.com/content/www/us/en/docs/opencl-sdk/developer-guide-processor-graphics/2019-4/using-vector-data-types.html`

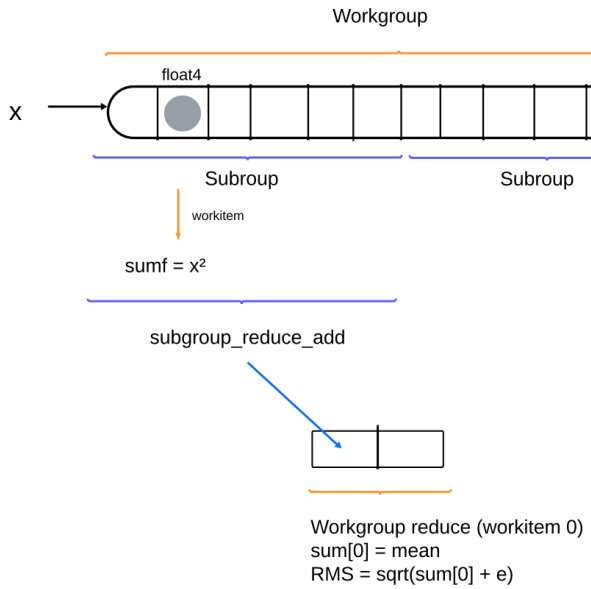


Abbildung 24: Schematische Darstellung der Berechnung der RMS-Norm

1. **Quadratberechnung:**
   - Pro Work-Item: 8 FLOPs * (ne00/4)/64
2. **Reduktion:**
   - All-sum Addition: 4 FLOP
   - Subgroup Reduce: log2(subgroup_size) Additionen = 5 FLOP
3. **Skalierungsfaktor:** 3 FLOP
4. **Normieung:** Jedes Workitem normiert (Anzahl Float4 Werte/Größe der Workgruppe) Werte. Pro Iteration werden die 4 Werte normiert und addiert
   - Pro Workitem: 4 * (ne00/4)/64 FLOP

Die **Gesamtzahl der FLOPs** ist (12 * (ne00/4)/64) + 12 = (12 * (ne00/256)) + 12 = (3*ne00)/64 pro WorkItem.

Work Item Setup

- nth = MIN(64, ne00)
- global worksize: (size_t)ne01*nth, 1, 1
- local worksize: (size_t)nth, 1, 1

Gemessene FLOP Die Messung der FLOPs ergab folende Werte:

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	0	0
FP_ARITH_SCALAR	13	13
FP_ARITH_256B	0	0
FP_ARITH_512B	228	3648
Gesamt		**3661**

Tabelle 3: Messung des RMS-Norm Kernel

Bei insgesamt 3661 FLOPs ergeben sich 3661/64 = 57 FLOPs pro WorkItem. Der Wert stimmt mit den theoretischen Überlegungen überein.

Softmax Die Softmax-Schicht wird in LLMs benötigt um eine Wahrscheinlichkeitsverteilung über die möglichen Ausgaben zu erhalten. Die Softmaxfunktion ist definiert als:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

wobei $z = (z_1, z_2, \ldots, z_K) \in \mathbb{R}^K$ ein Vektor von Eingabewerten ist und K die Anzahl der Klassen. Die Softmax-Implementierung von GGML verwendet einige Optimierungen im Gegensatz zur normalen Softmax Funktion:

- **Max-Subration:** Das Berechnen der Exponentialfunktion für große Tensorwerte kann zu Overflow führen. Deshalb werden alle Tensorwerte vor der Exponentialberechnung durch den max Tensorwert subtrahiert.
- **Temperatur:** Die Exponentialfunktionen der Softmaxfunktion werden durch den in Abschnitt 2.2.4 beschriebenen Temperaturwert T skaliert.
- **Attention Mask:** Vor der Softmaxberechnung kann eine Maske auf die Attention-Werte angewendet werden für:
  - Token-Padding: Bei Batchverarbeitung brauchen die Sequenzen gleiche längen und werden wenn nötig durch Padding aufgefüllt. Die Paddingeinträge werden durch die Maske auf -∞ gesetzt.
  - Masking, 2.2.2: Der Decoder darf nicht in die Zukunft sehen. Zukünftige Tokens werden durch die Maske auf -∞ gesetzt.
  - Kontext: Tokens die für die Vorhersage nicht relevant sind sollen weniger stark einfliessen.
- **ALiBi:** Der Softmax Kernel verwendet ALiBi. Die Positionsinformation der Embeddings wird nicht beim erstellen der Emebddings addiert sondern in der Softmax berechnung integriert, vergleiche Kapitel 2.2.1:

  $$\text{Score}_{i,j} = \frac{Q_i \cdot K_j}{\sqrt{d}} - m_h \cdot |i - j|$$

  Für jeden Attention-Score zwischen Token i und j wird ein linearer Bias subtrahiert, der vom Abstand $|\, i - j \,|$ abhängt. Das Modell bevorzugt dadurch Tokens, die näher beieinander liegen. Die Positionsinformation muss nicht explizit auf die Vektoren addiert werden.

  m ist eine gewählte Bias-Slope, die steuert, wie stark weiter entfernte Tokens benachteiligt werden. Im Kernel wird in einer Schleife für jeden Kernel die Slope berechnet. Die Positionsdifferenz $|\, i - j \,|$ ist in pmask enthalten. slope $\cdot$ pmask[i00] ist der lineare Positions-Bias, der auf die Attention-Werte addiert bzw. von ihnen subtrahiert wird.

Die finale Softmaxformel ergibt sich durch die Erweiterungen zu:

$$\text{softmax}(x_i) = \frac{\exp\left(x_i \cdot \text{scale} + \text{mask}_i \cdot \text{slope} - \text{max}\right)}{\sum_j \exp\left(x_j \cdot \text{scale} + \text{mask}_j \cdot \text{slope} - \text{max}\right)}$$

Theoretische FLOPs Qwen3 0.6B verwendet die F32 Version des Softmax Kernel. Die Totale FLOP-Anzahl des Kernels, für ein WorkItem berechnet sich folgendermassen:

- **ALiBi Berechnung**: 5 FLOPs. Wobei die letzte Berechnung pow(base, exp) aufwändiger ist. Aus Messungen ist zu sehen, dass die pow() Funktion 37 512-gepackte und rund 15 Skalare Operationen und dadurch rund 608 FLOPs benötigt.
- **Maximum Suche**:
  - Mit Pmask: [16 FLOPs x (ne00/4)/Workgroup-Size] + 3 + log2(Subgroup-Size)
  - Ohne Pmask: [12 FLOPs x (ne00/4)/Workgroup-Size] + 3 + log2(Subgroup-Size)
- **Exponentialberechnung + Summe**:
  - Max-Schleife: Mit Pmask [(ne00/4)/Workgroup-Size] x [4 x (4 FLOP)] + FLOPs Exp()
  - Max-Schleife Ohne Pmask: [(ne00/4)/Workgroup-Size] x [4 x (3 FLOP)] + FLOPs Exp()

  - Addition: 3 FLOP
  - Subgroup Reduce: log2(Subgroup-Size)
- **Division**: [4 FLOPs x (ne00/4)/Workgroup-Size]

In `llama.cpp` wird ALiBi nicht verwendet, da die Positionskodierung mittels RoPE erfolgt. Eine P-Maske wird in jedem Durchlauf angewandt. Für die Exponentialfunktion wurde auf der in dieser Arbeit verwendeten Hardware ein FLOP-Bedarf von 240 pro `float4` gemessen.
Daraus ergibt sich die Formel:

- **Maximum:** [16 FLOPs x (ne00/4)/Workgroup-Size] + 3 + log2(Subgroup-Size)
- **Exp + Summe:** [(ne00/4)/Workgroup-Size x (16 + 240 FLOP)] +3 + log2(Subgroup-Size)
- **Division:** [4 FLOPs x (ne00/4)/Workgroup-Size]
- **Gesamt:** [72 * ((ne00 / 4) / Workgroup-Size)] + 6 + 2*(log2(Subgroup-Size))

Insgesamt ergeben sich daraus: [276 * ((1024 / 4) / 64)] + 2*(log2(64)) + 6 = 1104 + 12 + 6 = 1122 FLOPs pro WorkItem

Work Item Setup

- nth = MIN(32, ne00)
- global worksize: (size_t)ne01*nth, (size_t)ne02, (size_t)ne03
- local worksize: (size_t)nth, 1, 1

Gemessene FLOPs Die Messung der FLOPs ergab folende Werte:

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	16	64
FP_ARITH_SCALAR	72	72
FP_ARITH_256B	0	0
FP_ARITH_512B	4584	73344
Gesamt		**73480**

Tabelle 4: Messung des Softmax Kernel

Mit den 64 WorkItems ergeben sich 73480/64=1148 FLOP pro WorkItem. Das entspricht der theoretischen Überlegung.

Rope RoPE, siehe Abschnitt 2.2.1, wird in Qwen3 für die Positionsinformation der Tensoren verwendet. RoPE wird auf die `Q`- und `K`-Vektoren angewendet. Im Attention-Mechanismus erfolgt dies nach der Multiplikation der Embeddings mit den erlernten `Q`- und `K`-Matrizen und vor der Berechnung der Attention-Scores. Die RoPE-Implementierung in GGML stellt folgende Varianten bereit, abhängig vom Anwendungsfall:

- **neox:** Standardimplementierung für sequentielle Textdaten.
- **vision:** Angepasste Variante für zweidimensionale Positionskodierung in Bild- oder Multimodalmodellen.
- **mrope:** Modifizierte RoPE-Version für längere Sequenzen oder spezielle Kontextanforderungen.

Für das Qwen3 Modell mit 0.6B Parametern und der Verwendung des llama-cli Tools zur Textgenerierung wird die neox Variante verwendet. Genauer die Version mit Floating Point 32 Genauigkeit.
Die RoPE-Implementierungen in GGML unterstützen YaRN (Yet another RoPE extension)[30]. YaRN ist eine effiziente Methode zur Erweiterung der Kontextlänge von Transformer-Modellen mit RoPE. Der Ansatz behandelt unterschiedliche Rotationsfrequenzen unterschiedlich. Niedrige Frequenzen werden stärker

[30] `https://arxiv.org/pdf/2309.00071.pdf`

skaliert, während hohe Frequenzen möglichst unverändert bleiben oder nur moderat angepasst werden, um lokale Positionsbeziehungen zu erhalten. Dadurch lässt sich der für die Inferenz nutzbare Kontext über den ursprünglichen Trainingskontext hinaus vergrößern. Für die Standard-RoPE-Berechnung werden zunächst die Frequenzen θ_d für jedes rotierende Wertepaar der Einbettung bestimmt. Damit wird anschließend für jeden Wert der Roationswinkel bestimmt. Für jedes Wertepaar wird damit abschließend RoPE durch die Formel

$$x'_1 = x_1 \cos(m\theta) - x_2 \sin(m\theta)$$
$$x'_2 = x_2 \cos(m\theta) + x_1 \sin(m\theta)$$

berechnet.
YaRN erweitert mehrere Teile dieser RoPE-Berechnung und passt die effektive Frequenzskalierung kontextabhängig an, um längere Sequenzen verarbeiten zu können. YaRN erweitert die Standard-ROPE-Theta-Berechnung durch eine sogenannte NTK-by-parts Interpolation, die verschiedene Frequenzbereiche der Rotationsmatrix selektiv behandelt. Während Standard-ROPE alle Dimensionen gleichmäßig mit dem Skalierungsfaktor s interpoliert, implementiert YaRN eine frequenzabhängige Interpolation basierend auf einer Ramp-Funktion. Dabei werden hohe Frequenzen ($r < \alpha$) nicht interpoliert ($\gamma = 0$), um lokale Positionsinformationen zu bewahren, niedrige Frequenzen ($r > \beta$) vollständig interpoliert ($\gamma = 1$) für langreichweitige Abhängigkeiten, und mittlere Frequenzen graduell interpoliert basierend auf der Ramp-Funktion $\gamma(r) = (r - \alpha)/(\beta - \alpha)$. Diese selektive Behandlung ermöglicht es YaRN, die Kontextgröße zu erweitern, während sowohl lokale als auch globale Positionsmuster erhalten bleiben. Dies führt zu einer besseren Extrapolationsfähigkeit auf längere Sequenzen.
Die RoPE-YaRN Funktion bekommt als Input auch den Parameter ext_factor. Der Wert bestimmt ob YaRN bei der der Positionsbestimmung Interpolation (ext_factor = 0) oder Extrapolation verwenden soll. In dem betrachteteten Qwen3 Modell ist ext_factor immer = 0.

Theoretische FLOPs Betrachtet wird die FLOP-Anzahl des Neox-kernel. Min/Max Funktionen werden nicht als FLOP gezählt

- **Hilfsmethoden:**
  - **rope_yarn_ramp**: 5 FLOP
  - **rope_yarn**
    - Ohne Extrapolation: 19 FLOP:
    - Mit Extrapolation: 43 FLOP:
  - **rope_yarn_corr_factor**: 22 FLOP
  - **rope_yarn_corr_dims**: 49 FLOP
- **Berechnung:**
  - Aufruf der Hilfsfunktion und Berechnung von inv_ndims: 50 FLOP
  - Hauptschleife (ne0/2*Subgroup-Size):
    - 45 FLOPs
- **Gesamt:** 50 + [(ne00/(nth*2)) * 45] = 50 + 45 = 95 FLOP pro WorkItem

Work Item Setup

- nth = 64 (Intel)
- Global Work-Size: (ne01*nth, ne02, ne03)
- Local Work-Size: (nth, 1, 1)

Gemessene FLOPs Die Messung der FLOPs ergab folende Werte:
Bei insgesamt 53584 FLOPs ergeben sich 53584/512 104 = FLOPs pro WorkItem. Der Wert stimmt ungefähr mit den theoretischen Überlegungen überein.

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	0	0
FP_ARITH_SCALAR	336	336
FP_ARITH_256B	0	0
FP_ARITH_512B	3328	53248
Gesamt		**53584**

Tabelle 5: Messung des Rope Kernel

Silu Für die SiLU-Operation sind zwei OpenCL-Kernel `kernel_silu` und `kernel_silu_4` implementiert. Der Kernel `kernel_silu_4` arbeitet auf `float4`-Vektoren und wird verwendet, wenn die Tensorlänge durch vier teilbar ist. Das Qwen3-Layer nutzt `kernel_silu_4`. Der SiLU-Kernel berechnet die Aktivierungsfunktion $\text{SiLU}(x) = x \cdot \sigma(x)$ mit der Sigmoid-Funktion $\sigma(x)$. Jedes Work-Item berechnet die Aktivierungsfunktion für sein zugeordnetes Element im Array `src0` und schreibt das Ergebnis in `src1`.

Theoretische FLOPs Die Silu-Berechnung besteht aus einer Codezeile:

```
dst[get_global_id(0)] = x / (1.0f + exp(-x));
```

Die Berechnung braucht drei FLOPs und die benötigten FLOP für die exp() Berechnung. Gesamt:

```
3 + Exp() FLOPs
```

Für exp() werden hier wieder 240 FLOP angenommen. Daraus ergeben sich für die in dieser Arbeit verwendeten Hardware 243 FLOPs pro WorkItem. Die genaue Zahl ist stark abhängig von der Implementierung der exp-Funktion.

Work Item Setup

- n = num_elements(dst) / 4 = ne00/4
- global worksize:(size_t)n, 1, 1
- local worksize: (size_t)64, 1, 1

Gemessene Flops Mit Likwid Perfctr wurden die durchgeführten Operationen für einen Kernel gemessen Gemessen wurde der Float 4 Kernel: 190464/768 = 248 FLOPs pro Work Item. Die hohe Anzahl

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	1536	6144
FP_ARITH_SCALAR	0	0
FP_ARITH_256B	0	0
FP_ARITH_512B	11520	184320
Gesamt		**190 464**

Tabelle 6: Messung des Silu Kernel

an FLOPs, die einen deutlich höheren Wert haben als in der theoretischen Überlegung, entsteht durch die aufwändige Implementierung der Exponentialfunktion. Wird die Exponentialfunktion aus dem Code entfernt ergeben sich die gemessenen Werte aus Tabelle 7. Mit den 768 WorkItems ergeben sich in dieser

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	0	0
FP_ARITH_SCALAR	0	0
FP_ARITH_256B	0	0
FP_ARITH_512B	384	6144
Gesamt		**6144**

Tabelle 7: Messung des Silu Kernel ohne Exponentialfunktion

Messung 6144/768 = 8 FLOPs pro WorkItem. Das entspricht ungefähr der theoretischen Analyse. Die Exponentialfunktion verursacht in der Testumgebung pro WorkItem 240 FLOPs. Wie in den Messannahmen beschrieben, zeigt sich hier die starke Abhängigkeit des Berechnungsaufwands von der konkreten Umsetzung mathematischer Operationen auf der verwendeten Hardware.

Mul Die Datei `mul.cl` implementiert die elementweise Multiplikation, im Unterschied zu den Matrix-`mul`-Kerneln ohne Skalarprodukt.
In `mul.cl` befinden sich zwei Kernel zur Tensor-Multiplikation: `kernel_mul` für die allgemeine elementweise Multiplikation zweier Tensoren sowie der optimierte `kernel_mul_row`, der den Eingabepuffer `src1` als einzelne Zeile interpretiert und diese über alle Zeilen von `src0` multipliziert. `kernel_mul_row` verwendet zudem `float4` anstelle von `float`. `kernel_mul` berechnet für jedes zu multiplizierende Element mithilfe von Byte-Strides die Zielposition und lässt sich dadurch auch auf nicht-kontinuierliche Daten anwenden. Die Adressierung erfolgt über Modulo-Berechnungen. Eine Work-Group verarbeitet zeilenweise in einer Schleife, wobei die Work-Items mehrere Elemente der Zeile übernehmen.
`kernel_mul_row` benötigt nur eine Modulo-Operation, da alle relevanten Werte innerhalb einer Zeile liegen.

Theoretische FLOPs

- **kernel_mul:**
  - Adressberechnung: 3*6 FLOP
  - Schleife: (ne0/local_size) * 7 FLOP
- **Gesamt:** 18 + (ne0/local_size) * 7 FLOP
- **kernel_mul_row:**
  - Adressberechnung: 3 + 3 FLOP
  - Multiplikation: 1 FLOP
- : **Gesamt:** 7 FLOP

Work Item Setup

- nth = MIN(64, ne0)
- global worksize: ne01*nth, (size_t)ne02, (size_t)ne03
- local worksize: nth, 1, 1

Mat Mul Die Matrixmultiplikation ist die aufwändigste Berechnung in der LLM-Inferenz. In GGML existieren dafür mehrere Kernel-Varianten, die an Hardwareeigenschaften und Modellquantisierung angepasst sind. Die Varianten unterscheiden sich in der verwendeten Speicherpräzision bzw. Quantisierung sowie in Optimierungen, die je nach Quantisierungsform Vorteile bringen. Unterstützt werden Kernel für 32- und 16-Bit-Floating-Point-Tensoren sowie für Q4_0- und Q6_K-Quantisierungen in unterschiedlichen Kombinationen. Eingesetzte Optimierungen umfassen Loop-Unrolling, verschiedene SIMD-Parallelisierungsgrade und das Aufteilen der Work-Items. Abbildung 25 zeigt die in GGML-OpenCL implementierten Multiplikationskernel und ihre wichtigsten Unterschiede.

Kernel, die mit quantisierten Tensoren arbeiten, passen die Berechnung an das jeweilige Datenlayout an und sind dadurch deutlich komplexer. Alle Kernel nutzen SIMD über OpenCL-Subgruppen. Die konkreten Einstellungen hängen von der Zielhardware (z.B. Adreno vs. Intel-GPU) ab. Für nicht quantisierte Kernel wird bei Vektorgrößen über 128 zusätzlich Loop-Unrolling eingesetzt.Die Auswahl des zu verwendenden Multiplikationskernels geschieht in der `mat_mul` Operation in der Hauptdatei der OpenCL Implementierung `ggml_opencl`, abhängig von der verwendeten Hardware und Quantisierung. Für den in dieser Arbeit verwendeten Intel Rechner und das Qwen3 0.6B Modell mit Q4 quantisierten Gewichten wählt `llama.cpp` zwei Matrixmultiplikationskernel aus:

- kernel_mul_mat_f16_f32_l4
- kernel_mul_mat_q4_0_f32_8x_flat

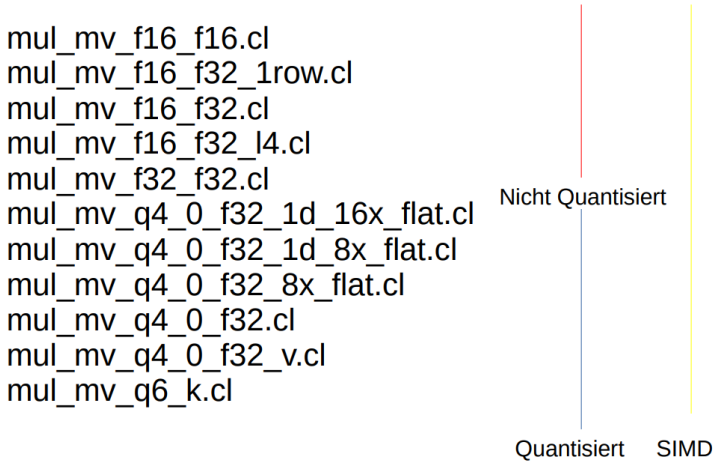


Abbildung 25: Übersicht über die Multiplikationskernel

Aus dem in Abschnitt 3.1.2 beschriebenen Berechnungsgraphen lässt sich ablesen, wann welche der beiden Varianten verwendet wird.

- Attention Schicht
  - Projektionsmatrizen W_q, W_k und W_v: F16_F32
  - Attention-Berechnung: Q4-Quantisiert
  - Output-Gewicht: Q4-Quantisiert
- FFN
  - FFN Norm: F16_F32
  - FFN Gate: Q4-Quantisiert
  - FFN Up: Q4-Quantisiert
  - FFN Down: Q4-Quantisiert
  - Output-Norm: F16_F32

Mat Mul F32 Der Matrixmultiplikations-Kernel `kernel_mul_mat_f16_f32_l4` verwendet Tensoren unterschiedlicher Präzision (`src0`: FP16, `src1`: FP32). Der Ausgabetensor hat die Präzision FP32.

Der Kernel nutzt Subgroup-Funktionalität in Kombination mit Vektor-Datentypen. Die Werte des FP16-Tensors werden als `half4` eingelesen, die Werte des FP32-Tensors als `float4`. Eine Work-Group bearbeitet eine Zeile von `src0` und multipliziert diese mit `ne11` Zeilen aus `src1`. In einer Schleife multiplizieren die Work-Items die Vektordaten und akkumulieren die Produkte mittels `sub_group_reduce_add`. Vor der Multiplikation werden die FP16-Werte nach `float` konvertiert. Das Work-Item der Subgroup mit id = 0 schreibt das Ergebnis zurück. Jedes Work-Item verarbeitet somit `ne11` $\cdot \lceil$(`ne00`/4)/`subgroup_size`$\rceil$ `float4`-Elemente.

Theoretische FLOPs Der theoretische Berechnungsaufwand pro Multiplikation eines float4 Wertes beträgt: ne11 * [((ne00/4)/subgroup-size) * (12 + log2(Subgroup Size))] FLOP pro WorkItem.

Work Item Setup

- nrows = ne11 ny = (nrows+ne11-1)/ne11 nth0: 32 nth1: 1
- global worksize:(size_t)ne01*nth0, (size_t)ny*nth1, (size_t)ne12*ne13
- local worksize: (size_t)nth0, (size_t)nth1, 1

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	128	512
FP_ARITH_SCALAR	64	64
FP_ARITH_256B	0	0
FP_ARITH_512B	1088	17 408
Gesamt		**17 984**

Tabelle 8: Messung des Mat Mul F32 Kernel

Gemessene FLOPs Mit Likwid Perfctr wurden die durchgeführten Operationen für einen Kernel gemessen: Bei insgesamt 17984 FLOPs ergeben sich 17984/1024 = 17.5 FLOPs pro WorkItem.
Bei einer Mutliplikation mit den Tensordimensionen ne00 = 128 und ne 11 = 1 ergeben sich die theoretischen FLOPs zu 17 FLOP pro Workitem
Der gemessene Wert stimmt mit den theoretischen Überlegungen überein.

Mat Mul Q4 8x

Q4 Kernel Die Matrixmultiplikation in den quantisierten OpenCL-Kerneln ist komplexer als bei Fließkommawerten. Die in GGML verwendeten Kernel verwenden Blockquantisierung. Dabei werden die quantisierten Daten in einem Struct gespeichert, das jeweils 32 Werte sowie einen Skalierungsfaktor enthält:

```
struct block_q4_0
{
half d;
uint8_t qs[QK4_0 / 2];
};
```

Im Array qs sind pro Eintrag zwei 4-Bit-Werte in einem uint8_t gespeichert. Das Array umfasst 16 Einträge, die zusammen 32 quantisierte Werte enthalten.
Die Multiplikation der Matrizen erfolgt in der Methode `block_q_4_0_dot_y`. Hierbei werden die 4-Bit-Werte schrittweise per Bitmasken aus dem Array qs extrahiert und mit den vorbereiteten Werten des Vektors yl multipliziert. Erst danach erfolgt die Dequantisierung über den Skalierungsfaktor:

```
float block_q_4_0_dot_y
for (int i = 0; i < 8; i+=2) {
acc.s0 += yl[i + 0] * (qs[i / 2] & 0x000F)

yl[i + 1] * (qs[i / 2] & 0x0F00);
acc.s1 += yl[i + 8] * (qs[i / 2] & 0x00F0)

yl[i + 9] * (qs[i / 2] & 0xF000);
}
return d * (sumy * -8.f + acc.s0 + acc.s1);
```

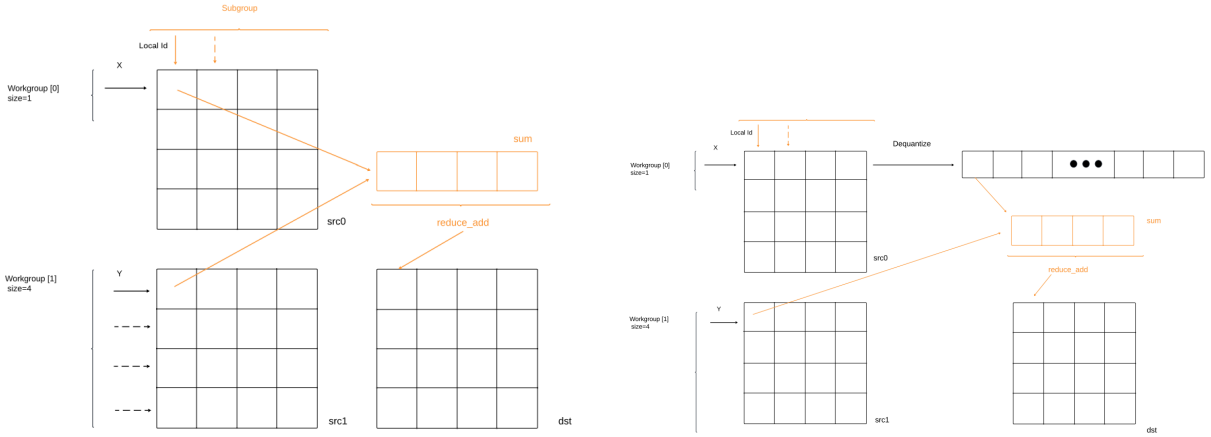
Da die 4-Bit-Werte aus qs per Bitshift extrahiert werden, muss der Vektor y vorab durch Division angepasst werden. So stehen die Werte an den richtigen Positionen für die Multiplikation. Die Ergebnisse der Multiplikation werden anschließend mit einer Subgroup-Reduce-Funktion aufaddiert und in den Ziel-Tensor geschrieben. Die zu verarbeitenden Daten werden auf OpenCL-Workgroups und Subgroups verteilt. Eine Workgroup verarbeitet jeweils eine Zeile der Matrix, wobei die Zeilen per Workgroup-ID bestimmt werden:

```
int r0 = get_group_id(0);
int r1 = get_group_id(1);
```

In den GGML-Kerneln entspricht eine Workgroup genau einer Subgroup. Die Elemente der Zeile werden über get_sub_group_local_id() auf die Threads verteilt:

```
for (int i = get_sub_group_local_id(); i < ne00; i += get_max_sub_group_size()) {
    sumf += (float) x[i] * (float) y[i];
}
```

Jeder Thread berechnet seine zugeteilten Elemente und springt dann entlang der Zeile weiter. Abbildung 8 zeigt die unterschiedliche Berechnung der Float und quantisierten Kernel.



(a) Floating Point Multiplikation

(b) Quantisierte Multiplikation

Abbildung 26: Multiplikation Schaubild

Q4_0_f32_8x_flat Der Kernel **mul_mv_q4_0_f32_8x_flat** unterscheidet sich vom Basis Q4-Kernel in drei Arten:

- Die quantisierten Daten liegen in der 8x-Version im SOA (Structure of Array) und nicht Array of Structures vor. Die quntisierten Daten und Skalierungsfaktoren sind nicht in einem Struct sondern über direkte Pointer gespeichert.
- Loop Unrolling.
- N_DST = 8. Pro Thread werden 8 Zeilen aus src0 mit src1 multipliziert.

Theoretische FLOPs Wie in der normalen Q4 Kernelversion ist für die FLOP Analyse die Multiplikationsmethode block_q_4_0_dot_y und die Hauptschleife, in der die Tensoren vorbereitet werden und die Methode aufgerufen wird, relevant.

- **Multiplikationsblock** (block_q_4_0_dot_y):
  - Mutliplikation: 16 Multiplikationen + 16 Additionen = 32 FLOP
  - Skalierung: d * (sumy * -8.f + acc) = 3 FLOPS
  - Gesamt: 35 FLOPs
- **Hauptschleife:** (ne00/32)/8 Iterationen:
  - Anpassen der Werte aus src1 an die q4 Multiplikation: 12 Additionen + 16 Divisionen = 28 FLOPs
  - Multiplikationsblock für N_DST (8) Zeilen: (8 * [35(Multiplikationsblock) + 1 (Addition)]) = 288 FLOPs
  - yb update: 3 FLOP
  - Subgroup-Addition: 8 * 5 FLOP
  - Gesamt: 28 + 288 + 3 + 40 = 359 FLOP
- **Gesamt:** (ne00/32)/8 * 359 FLOP = (ne00/256) * 359 FLOP

Work Item Setup

- nth0: 16 nth1: 1 ndst = 8
- global worksize:(ne01 + ndst-1)/ndst*nth0, ne11*nth1, ne12*ne13 =(ne01 + 7)/8*16, ne11, ne12*ne13;
- local worksize: (size_t)nth0, 1, 1;

Gemessene FLOPs Mit Likwid Perfctr wurden die durchgeführten Operationen für einen Kernel gemessen:

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	2048	8192
FP_ARITH_SCALAR	1024	1024
FP_ARITH_256B	0	0
FP_ARITH_512B	324608	5193728
Gesamt		**5 197 944**

Tabelle 9: Messung des Mat Mul Q4 Kernel

Bei 2048 WorkItems ergeben sich 5197944/2048= 2538 FLOP pro WI
Die Anzahl an erwarteten theoretischen FLOPs für diese Ausführung ist: (ne00/256) * 359 FLOP = (2048/256) * 359 FLOP = **2872 FLOP pro WI**.
Die gemessene FLOP-Zahl ist somit rund 12 Prozent niedriger als die theoretisch erwartete.

Get_Rows Der *get_rows* Kernel wird verwendet um neue Tokens für die Inferenz aus der Embedding Matrix zu laden. In dem verwendeten Qwen3 0.6B Modell wird dafür die Kernelvariante *kernel_get_rows_f32* für FP32 Tensoren verwendet. In einer Schleife kopieren die Workitems dafür die Daten von einem src in einen dst Vektor.

Theoretische FLOPs

- Berechnung der Speicheradresse: 3 + 4 FLOP
- Schleife: [ne00/localsize] * 8
- **Gesamt:** [ne00/localsize] * 8 + 7

Work Item Setup

- global worksize: (size_t)ne10, (size_t)ne11, 1;
- local worksize: 1, 1, 1;

Sonstige Operationen Der Aufwand für die restlichen Operationen ist vernachlässigbar. Für Veränderungen der Tensorform müssen nur die Metadaten ne und nb angepasst bzw vertauscht werden, beispielsweise für die Transopose oder Permute Operationen.

3.2.3 Layer

Um den gesamten Berechnungsaufwand eines Layers zu berechnen, wurde ein Python-Programm erstellt. Dafür wurde das NetworkX-Paket[31] verwendet, um einen Graphen aufzubauen, sowie Pyplot, um den Graphen zu zeichnen. Der Graph wurde dabei als Nachbildung des Operatorbaums eines Qwen3-0.6B-Layers erstellt. Um die benötigten FLOPs für die Berechnung eines Layers auszugeben, wurde interne Logik eingefügt. Eine Methode `compute_flops` berechnet für jede Operation, also jeden Knoten im Graphen, wie viele FLOPs diese benötigt. Dies geschieht abhängig von dem Input-Tensor, auf den der Knoten zugreifen kann. Um die Größen der Tensoren zu berechnen, wurde die Methode `compute_output_shape` hinzugefügt. Nach jeder durchgeführten Operation wird dort die Größe des Ergebnistensors angepasst. Die Visualisierung des Graphen zeigt dadurch drei Ergebnisse: Zum einen zeigt jede Kante die veränderte Tensorgröße nach jeder Operation. In jedem Knoten wird die benötigte FLOP-Zahl für diese Operation mit den jeweiligen Input-Tensoren angezeigt. Schließlich wird am Ende die Anzahl benötigter FLOPs für das gesamte Layer dargestellt. Das erstellte Python-Tool hat gegenüber einem händisch erstellten Schaubild drei entscheidende Vorteile: Zum einen lassen sich die Formeln einfach anpassen. In den Überlegungen zu den FLOP-Zahlen der Kernel mussten für aufwändige mathematische Operationen, wie z. B. die Exponentialfunktion, FLOP-Werte auf der jeweiligen Hardware angenommen werden. Sollten sich diese ändern, kann die Formel im Programm einfach angepasst und die komplette FLOP-Zahl neu berechnet werden.

[31] `https://networkx.org/`

Ein großer Vorteil des Programms ist es, dass sich die Input-Größe des Tensors dynamisch ändern lässt. Die FLOP-Zahl lässt sich so einfach für mehrere Szenarien durchrechnen, beispielsweise für die Prefill-Phase mit einer großen Input-Matrix oder die Decode-Phase mit nur einem Token. In späteren Tests ließen sich dadurch Messungen einfach validieren und mit den theoretischen Ergebnissen auf Korrektheit prüfen. Ein dritter Vorteil für zukünftige Arbeiten ist, dass sich neue Knoten einfach in das Netz einfügen oder entfernen lassen. Viele andere LLM-Familien, die ebenfalls Decode-Only-Architekturen nutzen, ähneln sich im Aufbau zu Qwen3. Das Programm lässt sich somit in Zukunft auch auf andere LLMs anwenden. Zusammenfassend wurden die FLOPs der relevanten Tensoroperationen eines Layers berechnet.
Das Ergebnis zeigte, dass die theoretischen FLOPs für ein Layer eine ähnliche Größenordnung wie das der LIKWID Messung eines Layers hatte, vorrausgesetzt, die finale Logit-Berechnung wurde nicht inkludiert. Die hohe Anzahl der FLOPs die für die Logit-Berechnungen nötig sind traten in der LIKWID Messung der Inferenz nicht auf.

3.2.4 Messungen

Um die theoretischen Berechnungen der benötiten FLOPs eines Layers zu validieren sollte die Berechnung eines gesamten Layers mit Likwid getestet werden. Hier traten jedoch Probleme mit der verwendeten Hardware auf.

Voraussetzungen für Likwid Messung Der Likwid-perfctr zur Messung von FLOPs lässt sich mit NVIDIA und AMD Grafikkarten verwenden. Für AMD Grafikkarten wird die ROCm (Radeon Open Compute Platform) Schnittstelle vorrausgesetzt. Eine vorhandene AMD R9 390 war zu alt und unterstütze diese Schnittstelle nicht. Auf einer integrierten Intel Grafikkarte liessen sich die FLOPs mit Likwid nicht messen.

Cloud Comuputing Eine verbreitete Möglichkeit für den Betrieb von LLMs ist das Mieten einer virtuellen Maschine. Verschiedene Anbieter stellen dafür VMs mit leistungsstarken Grafikkarten bereit, auf denen sich auch große Modelle ausführen lassen. Für Testzwecke, etwa in wissenschaftlichen Arbeiten, ist dies oft realistischer als der Neukauf von Hardware. Hochleistungs-GPUs von NVIDIA, die primär für das LLM-Training entwickelt wurden, können bei VM-Anbietern bereits zu niedrigen Stundensätzen gemietet werden, während der Neupreis solcher Karten bei über 25.000 US-Dollar liegt. Im Vergleich dazu sind Consumer-GPUs für den privaten Gebrauch deutlich günstiger: Eine NVIDIA RTX 5090 mit 32 GB VRAM wird bei einem Cloud-Anbieter mit 0.69 US-Dollar pro Stunde berechnet[32], während der Kaufpreis aktuell über 2000 Euro liegt. Für FLOP-Messungen in einer virtuellen Maschine müssen jedoch mehrere Voraussetzungen erfüllt sein:

- **GPU-Virtualisierung:** In virtuellen Machinen kann der Zugriff auf eine GPU virtualisiert werden (vGPU). Dabei teilen sich mehrere virtuelle Maschinen eine Grafikkarte. Für die direkte Messung der FLOPs ist ein GPU-Passthrough notwendig bei dem die GPU exklusiv von der eigenen VM genutzt wird. Für die VM muss die Grafikkarte wie eine physische Karte erscheinen.
- **Zugriffsrechte:** Der Zugriff auf Hardware-Counter kann eingeschränkt und nur mit Administratorrechten möglich sein. Solche Adminrechte werden für die Messung benötigt.
- **Treiber:** Likwid benötigt für die FLOP Messung bestimmte Schnittstellen der Grafikkarten, bei AMD Karten beispielsweise ROCm. Aktuelle Treiber sind bei den virtuellen Maschinen wohl standardmässig vorhanden und müssen nicht extra installiert werden.

Aufgrund dieser Einschränkungen wurden die Messungen nicht in einer virtuellen Maschine durchgeführt.

NHR Cluster Die FAU betreibt mit NHR@FAU [33] ein eigenes Hochleistungsrechenzentrum für wissenschaftliche Anwendungen. Auf mehreren Clustern steht dort leistungsstarke Hardware für High Performance Computing zur Verfügung. Beispielsweise das Cluster Fritz mit mehreren hundert NVIDIA-A100-GPUs.
Die Leistungsstärksten Cluster stehen jedoch nur Nutzern mit erweiterten Zugangsrechten zur Verfügung. Im Rahmen dieser Arbeit konnte mit einem Tier-3-Zugang das TinyGPU-Cluster verwendet werden, das neben einer NVIDIA A100 auch aktuelle Consumer-GPUs wie etwa eine NVIDIA RTX 3080 verwendet.

[32] `https://www.runpod.io/pricing?utm_source=getdeploying.com&utm_medium=referral&utm_content=nvidia-rtx5090`
[33] `https://doc.nhr.fau.de/`

Dieses Cluster wurde genutzt, um LIKWID-Messungen auf der GPU durchzuführen und die Ergebnisse mit bisherigen Messungen auf einem Laptop mit Intel-CPU zu vergleichen.
Die Verwendung der modernere GPU des TinyGPU Clusters bietet zusätzliche Messmöglichkeiten mit LIKWID, die auf der Intel-CPU nicht verfügbar waren. Diese weitere Messgruppen sollten zusätzliche Einsichten in das Inferenzverhalten geben.
Es war möglich, eine Verbindung zum TinyGPU-Cluster herzustellen, die benötigten Daten zu übertragen und das llama-cli Tool dort zu kompilieren. Der Aufruf des CLI-Tools sowie die LIKWID-Messung erfolgen auf dem NHR-Cluster nicht interaktiv auf der Kommandozeile, sondern über den Slurm Workload Manager. Hierzu wurde ein Batch-Skript erstellt, das die Messung der Inferenz mit Likwid startet und die Ergebnisse in eine Textdatei schreibt.
Im ersten Teil des Skripts werden der auszuführende Clusternode, die gewünschte Hardware und weitere Konfigurationsparameter für das Programm festgelegt. Für die Messung wurden folgende Ausführungsoptionen angegeben:

```
#SBATCH --partition=rtx3080
#SBATCH --gres=gpu:1
#SBATCH --job-name=qwen_flops
...
#SBATCH --nodes=1
#SBATCH --time=00:01:00
#SBATCH --exclusive
#SBATCH --constraint=hwperf
```

Für die Messungen wurde eine RTX-3080-Grafikkarte verwendet, wobei für die isolierte Messung der FLOPS ein exklusiver Zugriff auf die GPU sowie der Constraint hwperf erforderlich sind.
Anschließend wird der Job in die Warteschlange eingereiht und automatisch gestartet, sobald ein geeigneter Rechenknoten verfügbar ist.
Messungen konnte in die Warteschlange eingefügt und erfolgreich Ausgeführt werden. Die Ergebnisse beinhalteten sowohl die Likwid-Messung als auch Statistiken der GPU-Nutzunhgen.
Bei mehreren Messungen traten jedoch lange Wartezeiten auf, bis der Job auf dem Cluster ausgeführt werden konnte. Beispielsweise:

```
SubmitTime=2025-11-26T09:29:28
StartTime=2025-11-26T23:41:55
```

Aufgrund dieser langen Wartezeiten wurden die weiteren Messungen nicht auf dem NHR-Cluster, sondern lokal auf dem eigenen System durchgeführt. Ein weiteres Problem bestand darin, dass die OpenCL-Implementierung von `llama.cpp` nicht für NVIDIA-Grafikkarten optimiert ist. Für NVIDIA-GPUs verwendet `llama.cpp` standardmäßig die CUDA-Implementierung. Diese konnte zwar ebenfalls auf dem Cluster ausgeführt werden, lässt sich aufgrund von spezifischen Optimierungen jedoch nicht direkt mit den OpenCL-Kernels vergleichen.

`llama.cpp` auf CPU Da nicht die Performance sondern nur die totale ausgeführte FLOP Zahl wichtig ist, wurde die Likwid Messung auf der CPU durchgeführt. OpenCL Kernel lassen sich generell Plattformunabhängig ausführen. Sowohl auf CPU und GPU. Im Host Code muss dafür das Device auf dem der Kernel ausgeführt wird geändert werden. Im Hostcode von `llama.cpp` wird das verwendete Device in der Methode: `ggml_cl2_init` angegeben. Die Ausführung lässt sich einfach auf die CPU schieben indem in der Auswahl der verfügbaren Geräte nur CPUs gewählt werden:

```
//cl_int clGetDeviceIDsError = clGetDeviceIDs(p->id, CL_DEVICE_TYPE_ALL, NDEV,
    device_ids, &p->n_devices);

zu:

cl_int clGetDeviceIDsError = clGetDeviceIDs(p->id, CL_DEVICE_TYPE_CPU, NDEV,
    device_ids, &p->n_devices);
```

Für ein Ziel eines späteren Kapitels wurde der Code von `llama.cpp` so angepasst, dass er auf einem FPGA lauffähig ist. Dafür wurde eine Framework verwendet das im Rahmen einer Doktorarbeit am Lehrstuhl 3 entwickelt wird. Mit dieser Bibliothek ließ sich `llama.cpp` jedoch nicht auf der CPU ausführen, da OpenCL-Kontextkonflikte auftraten. Geladene Modellparameter bzw. Kernel-Eingabedaten befanden sich in einem anderen Kontext und führten beim Aufruf zu einem Segmentation Fault. Für CPU-Messungen wurde deshalb das ursprüngliche Projekt erneut geklont.

LIKWID Im Projektstand zum Commit 814f795e [34] war es möglich, die OpenCL-Kernel auf der CPU auszuführen und eine LIKWID-Messung des gesamten Layers durchzuführen.

Mess-Setup Der Decode-Aufruf für einen Berechnungsgraphen erfolgt in der Klasse `llama_context` des Projekts `llama.cpp`. Die Methode `decode` wird dort definiert und für einen Input-Batch aufgerufen. Innerhalb von `decode` wird der Batch in kleinere Microbatches aufgeteilt. Für jeden Microbatch wird `graph_compute` aufgerufen und der Berechnungsgraph entsprechend abgearbeitet. In der Decode-Methode kann der Berechnungsgraph zudem ausprinted werden, um zu prüfen, dass die Messung auf der Annahme des korrekten Graphen basiert.
Für die Messung wurde LIKWID in der Datei `llama_context` inkludiert und in der CMake-Konfiguration von `llama.cpp` verlinkt. Um die benötigten FLOPs für die Berechnung des Operatorbaums pro Token zu erfassen, wird die LIKWID-Messung um die while-Schleife eines Mikrobatches gelegt. Nach Abschluss der Messung wird das Programm beendet.

```
decode(llama_batch & inp_batch) {
    #ifdef LIKWID_PERFMON
        LIKWID_MARKER_INIT;
        LIKWID_MARKER_THREADINIT;
        LIKWID_MARKER_START("layer");
    #endif
    while (sbatch.n_tokens > 0) {
        const auto compute_status = graph_compute(gf, ubatch.n_tokens > 1);
    }
    #ifdef LIKWID_PERFMON
        LIKWID_MARKER_STOP("layer");
        LIKWID_MARKER_CLOSE;
        std::exit(1);
    #endif
}
```

Um die Inferenzkosten nach der Prefill-Phase zu messen, wurde ein Counter verwendet, der die Anzahl der Decode Schritte zählt. Gemessen werden die FLOPs mit dem Aufruf:

```
likwid-perfctr -C 0 -g FLOPS_SP ./build/bin/llama-cli -m ./models/Qwen3-0.6B-Q4_0.gguf
--file longprompt.txt --no-warmup
```

Die Option `-C` pinnt die Codeausführung an einen bestimmten Thread, um die Messung gezielt auf diesem Kern durchzuführen. Um zu überprüfen, dass die Berechnung tatsächlich auf Kern 0 ausgeführt wird, wurde zusätzlich `likwid-pin` verwendet. Mit dem Aufruf

```
likwid-pin -c 0 ./build/bin/llama-cli ...
```

lässt sich das Programm explizit auf Kern 0 ausführen und anschließend mit `likwid-perfctr` messen. Die Ergebnisse stimmten mit denen des reinen `likwid-perfctr`-Aufrufs überein.

Warmup

In `llama.cpp` wird vor der ersten Inferenz standardmäßig eine Warmup-Phase durchgeführt. Dabei erfolgt eine Dummy-Berechnung mit reduzierter Batchgröße (1–2 Tokens), um die GPU-Kerne aus dem Ruhemodus zu aktivieren, Speicherbereiche für den KV-Cache und Zwischenergebnisse zu allokieren sowie Treiberinitialisierungen auszulösen. Diese Maßnahmen verringern den Einfluss von Start-Overheads und führen zu stabileren Laufzeiten in nachfolgenden Durchläufen.
Für FLOP-Messungen der Prefill-Phase größerer Prompts muss die Warmup-Phase mit der Option:

```
--no-warmup
```

deaktiviert werden. Der Grund liegt in der abweichenden Batchgröße während des Warmups, die nicht der tatsächlichen Ausführungscharakteristik entspricht. Bei einem Prompt von beispielsweise 800 Tokens würde das Warmup lediglich eine Batchgröße von 2 Tokens verarbeiten, während die eigentliche Prefill-Phase den gesamten Prompt in einem Durchlauf behandelt. Für eine korrekte Analyse der Floating-Point-Operationen und der Vektorisierungsanteile (Skalar-, 128-, 256- und 512-Bit-Operationen) ist daher die Messung der realen Batch-Verarbeitung des ersten Inferenzdurchlaufs erforderlich.

[34] https://github.com/ggml-org/llama.cpp/commit/814f795e063c257f33b921eab4073484238a151a

3.3 Roofline Analyse

Eine zentrale Fragestellung dieser Arbeit ist die Analyse der Rechenintensität und des Speicherverbrauchs der Inferenz. Für kleine LLM-Modelle soll bewertet werden, wann sich die Auslagerung von Berechnungen bzw. Layern auf eine GPU oder ein FPGA lohnt. Programme können grob in *memory-bound* oder *compute-bound* klassifiziert werden: Bei *memory-bound* Programmen ist die Laufzeit durch Wartezeiten auf Daten limitiert .Zusätzliche Rechenleistung verkürzt die Laufzeit nicht.
Für *memory-bound* Berechnungen kann sich eine Auslagerung auf die GPU aufgrund des zusätzlichen Datentransfers unter Umständen nicht lohnen, da die schnellere Hardware durch das Warten auf Daten nicht ausgelastet wird. Eine Übersicht über Rechen- und Speicherintensität einer Berechnung (bzw. eines Kernels) auf einer konkreten Hardware liefert das Roofline-Modell. Dazu werden die maximale Rechenleistung und die verfügbare Speicherbandbreite des Systems sowie die operationelle Intensität $I = \frac{\text{FLOPs}}{\text{Byte}}$ der Anwendung betrachtet.
Zwei Leistungsobergrenzen (Rooflines), diejenige der Spitzenrechenleistung und diejenige der Speicherbandbreite, begrenzen die erreichbare Ausführungsrate. Durch die Darstellung lässt sich für eine Anwendung, die im Roofline-Modell eingeordnet ist, schnell erkennen, ob sie rechendominant oder speicherdominant ist. Als Vorlage für die Erstellung des Roofline-Modells diente ein Tutorial aus der Dokumentation des Likwid-Tools. [35].

Peak Performance und Bandwith Für die Leistungsobergrenzen wurden die maximal erreichbaren FLOPs und die maximal verfügbare Speicherbandbreite auf dem in dieser Arbeit verwendeten Laptop mit Intel-i7-Prozessor bestimmt.

- **Peak Performance:** `likwid-bench -t peakflops_avx_fma -W N:80kB:8`
  - **MFlops/s: 255812.36**
- **Peak Bandwith:** `likwid-bench -t load -W N:2GB:36`
  - **MByte/s: 44853.02**

3.3.1 Szenarien

Die Auslagerung der Berechnung kann auf zwei Arten stattfinden. Zum einen können unterschiedliche Layer ausgelagert werden. Falls manche Layer deutlich rechenintensiver sind als andere, können diese auf dem schnelleren FPGA ausgeführt werden. Ausserdem können auch Teile eines Layer ausgelagert werden. Ein Layer besteht in den verwendeten Architekturen aus einem Attention Mechanismus und einem FFN-Block. Es soll dafür untersucht werden welcher Teil der beiden rechenintensiver ist und sich für die Auslagerung lohnen kann. Beide Szenarien sollen im folgenden gemessen und auf ihre Operationale Intensität untersucht werden.

Prefill und Decode Die beiden grundlegenden Phasen der LLM-Inferenz, die Prefill und die Decode Phase, wurden in Abschnitt 2.2.6 beschrieben. Für die Bewertung des Rechenaufwandes ist wichtig, dass in der Prefill-Phase eine größere Anzahl an Tokens als Input verarbeitet wird, während in der Decode-Phase jeweils nur ein neues Token hinzukommt. Aufgrund der größeren Eingabe und der damit verbundenen größeren Matrixdimensionen ist anzunehmen, dass die Prefill-Phase deutlich rechenintensiver ist als die Decode-Phase. Zu Beginn der Inferenz ist der KV-Cache noch leer, wodurch die erforderlichen Speicherzugriffe geringer ausfallen.
Ziel ist es, die konkreten Messwerte beider Szenarien zu bestimmen und den genauen Unterschied in ihrer operationellen Intensität herauzufinden und zu visualisieren.

Attention und FFN Eine zweite Form der Auslagerung von Berechnungen kann innerhalb eines Layers erfolgen. Ein LLM-Layer besteht im Wesentlichen aus dem Teil zur Berechnung des Attention-Mechanismus und einem Feed- Forward-Netzwerk (FFN). Anstatt ein gesamtes, besonders rechenintensives Layer auf einen FPGA auszulagern, soll untersucht werden, ob sich die Ausführung einzelner Komponenten eines Layers auf einem FPGA lohnt.
Zunächst wurde eine Literaturrecherche durchgeführt, um zu prüfen, ob bereits Überlegungen oder Messungen zum FLOP-Anteil von Attention- und FFN-Layern in LLMs vorliegen. Es fanden sich nur wenige

[35] `https://github.com/RRZE-HPC/likwid/wiki/Tutorial:-Empirical-Roofline-Model`

wissenschaftliche Arbeiten. Viele Überlegungen und Messungen sind in Blogbeiträgen privater Anwender dokumentiert.
Eine Arbeit zum FLOP-Aufwand einzelner Layer stammt von Forschenden der Johns Hopkins University und OpenAI [18]. Die Autoren geben dort den FLOP-Bedarf der Komponenten eines LLM-Layers in Abhängigkeit folgender zentraler Modellgrößen an:

- n-layer: Anzahl der LLM-Layer
- d model: Dimension des 'residual streams'
- d attn: Outputdimension der Multi-Head-Attention
- d ff: Outputdimension des FFN

In der Tabelle 10 sind die Ergebnisse für jeden Teil des Layers dargestellt. Aus den Formeln geht her-

Operation	**FLOPs per Token**
Attention: QKV	$2n_{\text{layer}}d_{\text{model}}3d_{\text{attn}}$
Attention: Mask	$2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$
Attention: Project	$2n_{\text{layer}}d_{\text{attn}}d_{\text{embd}}$
Feedforward	$2n_{\text{layer}}2d_{\text{model}}d_{\text{ff}}$
Total (Non-Embedding)	$N = 2d_{\text{model}}n_{\text{layer}}(2d_{\text{attn}} + d_{\text{ff}})$, $C_{\text{forward}} = 2N + 2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$

Tabelle 10: Rechenaufwand der verschiedenen Teile eines Transformer-Modells. Quelle: [18]

vor, dass der Rechenaufwand des FFN stark von der Modellgröße abhängt, während der Aufwand der Attention mit der Kontextlänge bzw. der Größe der KVQ-Dimension wächst.
Neben dieser theoretischen Analyse untersuchen zwei gefundene Blogbeiträge das Verhältnis des Rechenaufwands von FFN und Attention und visualisieren dieses grafisch. Einmal in Abhängigkeit von steigender Kontextlänge und einmal in Abhängigkeit von steigender Modellgröße.
Ein Beitrag [6] analysiert das Verhältnis des Rechenaufwands für steigende Kontextlängen am Beispiel von LLaMA 2 7B , siehe Abbildung 27. Es zeigt sich, dass bei einem kleinem Kontext das FFN die Laufzeit dominiert. Ab einer hinreichend großen Kontextlänge überschreitet jedoch der Aufwand der Attention diesen Wert und die Attention wird zur dominierenden Komponente. In einem weiteren Blogbeitrag [3]

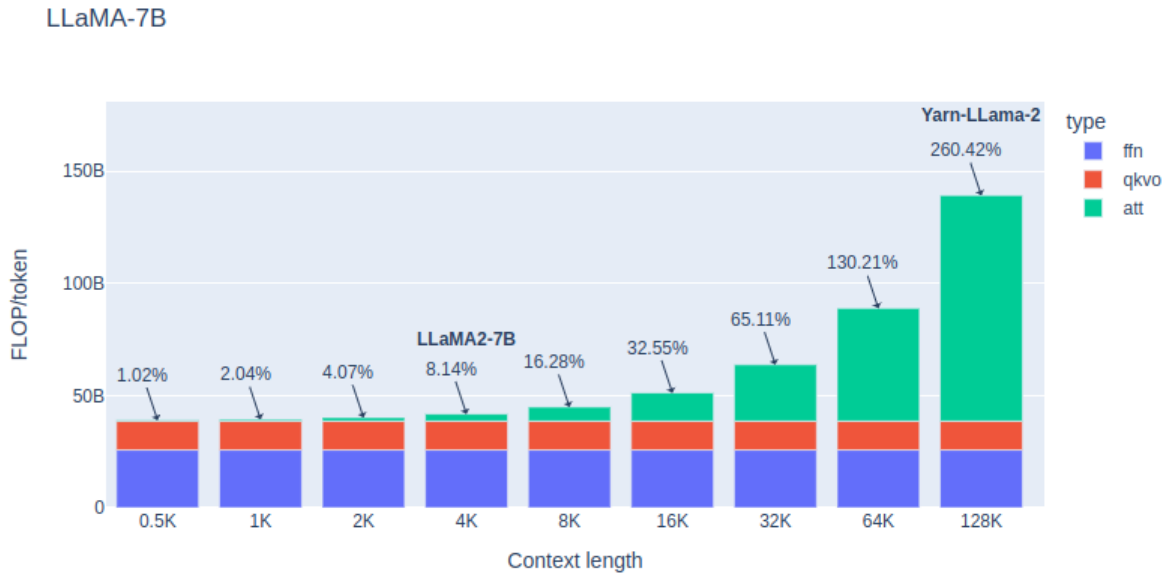


Abbildung 27: Veränderung der FLOP-Anteile bei steigendem Kontext. Quelle: [6]

wird bei fester Sequenzlänge die Modellgröße variiert und die FLOP-Verteilung visualisiert 28. Dabei zeigt sich, dass der FFN-Block bei sehr großen Modellen den Rechenaufwand dominiert, während bei kleineren

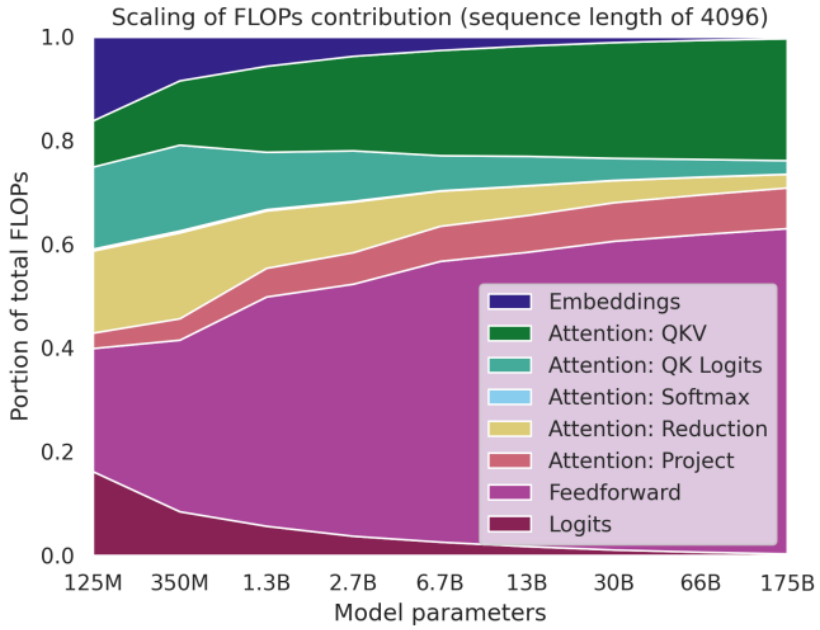


Abbildung 28: Veränderung der FLOP-Anteile bei steigender Modellgröße. Quelle [3]

Modellen wie Qwen3 0.6B der Attention-Block anteilig rechenaufwändiger ist. Auch wenn intuitiv vermutet werden könnte, dass der Attention-Mechanismus aufgrund der quadratischen Abhängigkeit von der Kontextlänge stets dominiert, zeigen diese Ergebnisse, dass insbesondere bei kleinen Kontexten der FFN-Block den größten Anteil am Rechenaufwand hat. Ursache sind die *FFN up*- und *FFN down*-Projektionen, die den Vektorraum zwischen d_{model} und d_{ff} erweitern bzw. zurückprojizieren.
Es ist zu erwarten, dass in den Messungen ein ähnliches Verhalten auftritt. Unklar bleibt jedoch, wie sich das Verhältnis der Anteile von FFN und Attention mit steigender Kontextlänge für ein sehr kleines Modell wie Qwen3 0.6B konkret verschiebt. Im Folgenden wurden hierfür Messungen mit kleinem und großem Kontext in der Prefill-Phase durchgeführt.

3.3.2 FLOP-Messungen

Mit dem bereits verwendeten Likwid Tool wurden die FLOPs für die beiden Szenarien gemessen.

Prefill und Decode Um einen realistischen Vergleich zwischen Prefill- und Decode-Phase zu erhalten, wurde ein mittellanger Eingabetext mit 832 Tokens (Beschreibung der Friedrich-Alexander-Universität) verwendet. Mit der Option `--verbose` lässt sich beim Aufruf von `llama.cpp` die Tokenisierung einschließlich der Gesamtzahl der Tokens ausgeben. Für die Messung der Prefill-Phase wurde die Inferenz nach dem ersten erzeugten Token abgebrochen.
Bei der Messung des Decode-Schritts traten Schwierigkeiten auf. Obwohl die LIKWID-Messung erst gestartet wurde, nachdem bereits das erste Token generiert war, wurde nicht ausschließlich die Decode-Phase erfasst. Die gemessenen FLOPs waren deutlich zu hoch, und die von LIKWID ausgewiesene Zeit entsprach der Gesamtlaufzeit, nicht nur der Decode-Dauer. Anhand der theoretischen FLOP-Abschätzungen pro Layer und dem erstellten Python Tool ließen sich diese unrealistischen Ergebnisse identifizieren. Zur Bestimmung der Decode-FLOPs wurden schließlich die in der Prefill-Phase gemessenen FLOPs von den FLOPs der Gesamtlaufzeit bis einschließlich des ersten Decode-Schritts subtrahiert. Auf diese Weise ergibt sich eine Näherung für den FLOP-Anteil der Decode-Phase unter ansonsten identischen Rahmenbedingungen.

Prefill Für die Prefill Phase wurde Qwen3 0.6B mit dem Inputprompt bestehend aus einem und 832 Tokens aufgerufen. Der Vergleich zeigt, dass die Prefill Phase mit einem Langen Input Prompt eine sehr hohe Rechenintensität aufweist.

Decode Die Decode-Phase lässt sich im Code dadurch identifizieren, dass die Input Grösse gleich Eins ist. Unterschiede entstehen jedoch durch die Größen der K- und V-Matrizen, die bei einem langen

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	473 600	1 894 400
FP_ARITH_SCALAR	7 868 353	7 868 353
FP_ARITH_256B	7 034 112	56 272 896
FP_ARITH_512B	411 328	6 581 248
Gesamt		**72 616 897**

Tabelle 11: Gemessene Anzahl an FLOPs für 1 Token, 28 Layer während der Prefill Phase

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	618 501 706	2 474 006 824
FP_ARITH_SCALAR	1 700 186 134	1 700 186 134
FP_ARITH_256B	930 892 768	7 447 142 144
FP_ARITH_512B	10 038 530 112	160 616 481 792
Gesamt		**172 237 816 894**

Tabelle 12: Gemessene Anzahl an FLOPs für 832 Token, 28 Layer während der Prefill Phase

vorherigen Prompt stärker gefüllt sind. Im Folgenden wurde die Decode Phase mit allen 832 Input Tokens für ein und alle 28 Layer verglichen

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	473 619	1 894 476
FP_ARITH_SCALAR	4 082 173	4 082 173
FP_ARITH_256B	7 034 112	56 272 896
FP_ARITH_512B	411 328	6 581 248
Gesamt		**68 830 793**

Tabelle 13: Gemessene Anzahl an FLOPs für 1 Token, 28 Layer während der Decode Phase (Differenz)

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	1 265 664	5 062 656
FP_ARITH_SCALAR	4 850 623	4 850 623
FP_ARITH_256B	7 523 328	60 186 624
FP_ARITH_512B	14 118 336	225 893 376
Gesamt		**295 993 279**

Tabelle 14: Gemessene Anzahl an FLOPs für 832 Token, 28 Layer während der Decode Phase (Differenz)

Die Messungen zeigen das charakteristische Verhalten des Rechenaufwands in der Prefill- und der Decode-Phase. Wird nur ein einzelnes Token verarbeitet, ist der Rechenaufwand beider Phasen nahezu identisch. Eine Prefill-Phase mit einem langen Eingabetext erfordert dagegen einen deutlich höheren Rechenaufwand. Ursache dafür sind die zahlreichen Token-Multiplikationen, die gleichzeitig durchgeführt werden. Der Vergleich der Decode-Phase mit einem langen Eingabetext und einem Eingabetext der Länge 1 verdeutlicht den Einfluss der K- und V-Matrizen. Obwohl in beiden Fällen nur ein neues Token als Eingabe dient, muss dieses bei einem gefüllten K- und V-Cache mit deutlich mehr zuvor gespeicherten Tokens multipliziert werden. Dadurch steigen die insgesamt benötigten FLOPs erheblich an.

Attention vs FFN Für den Vergleich der Attention und des FFN Block wurde analysiert, wie groß der FLOP-Anteil beider Komponenten an der Gesamtrechnung ist. Für die Messung wurde das Modell mit allen Layer erstellt. Verglichen wurden die FLOP Zahlen mit einem und 832 Input-Token während der Prefill-Phase.

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	458,496	1,833,984
FP_ARITH_SCALAR	6,934,989	6,934,989
FP_ARITH_256B	3,405,440	27,243,520
FP_ARITH_512B	296,640	4,746,240
Gesamt		**40,758,733**

Tabelle 15: Gemessene Anzahl an FLOPs für 1 Token, 28 Layer für den Attention-Block

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	473,600	1,894,400
FP_ARITH_SCALAR	7,868,354	7,868,354
FP_ARITH_256B	7,034,112	56,272,896
FP_ARITH_512B	411,328	6,581,248
Gesamt		**72,616,898**

Tabelle 16: Gemessene Anzahl an FLOPs für 1 Token, 28 Layer für das ganze Layer

1 Token

$$\text{Verhältnis der Gesamt-FLOPs:} \quad \frac{40\,758\,733}{72\,616\,898} \times 100 = 56.1\%$$

Das Attention Layer macht bei der Inferenz mit 832 Token somit rund 56 Prozent der Rechenintensität aus.

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	859,144,192	3,436,576,768
FP_ARITH_SCALAR	1,557,033,600	1,557,033,600
FP_ARITH_256B	1,407,811,584	11,262,492,672
FP_ARITH_512B	8,750,403,584	140,006,457,344
Gesamt		**156,262,560,384**

Tabelle 17: Gemessene Anzahl an FLOPs für 832 Token, 28 Layer für den Attention-Block

Instruktionstyp	Events	FLOPs
FP_ARITH_128B	618,501,696	2,474,006,784
FP_ARITH_SCALAR	1,700,185,925	1,700,185,925
FP_ARITH_256B	930,892,768	7,447,142,144
FP_ARITH_512B	10,038,530,112	160,616,481,792
Gesamt		**172,237,816,645**

Tabelle 18: Gemessene Anzahl an FLOPs für 832 Token, 28 Layer für das ganze Layer

832 token

$$\text{Verhältnis der Gesamt-FLOPs:} \quad \frac{156\,262\,560\,384}{172\,237\,816\,645} \approx 0.9073 \approx 90.73\%$$

Das Attention Layer macht bei der Inferenz mit einem Token somit rund 90 Prozent der Rechenintensität aus.

3.3.3 Speicherverbrauch

Für den Nenner der Operational Intensity wird der Speicherverbrauch des Programms während dem Durchlauf gemessen. Hier traten jedoch wieder Schwierigkeiten der Messung im Zusammenspiel mit der in dieser Arbeit verwendeten Hardware auf. Im Folgenden soll darauf eingegangen und die schließlich verwendeten Speicherflüsse der Szenarien gezeigt werden.

Listing 2: Durchführung der Intel VTune Messung

```
if (late_decode) {
    flush_cpu_cache();
    #ifdef INTEL_VTUNE
    __itt_resume();
    #endif
}
...
if(late_decode){
    #ifdef INTEL_VTUNE
    __itt_detach();
    std::exit(0);
    #endif
}
```

Vorgehen

Likwid Zuerst wurde versucht das Likwid Tool für die Speichermessung zu verwenden. Die Messung des Speicherflusses war in der verwendeten Hardware mit Likwid nicht möglich. Die benötigte Messgruppe MEM war auf dem Intel i7 Prozessor nicht verfügbar. Alternativ kann die LIKWID Messgruppe DATA verwendet werden. Die verfügbare Messgruppe DATA war für die Berechnung der Operational Intensity (OI) jedoch nicht ausreichend. DATA misst alle Datenzugriffe und nicht nur das Laden aus dem RAM. Für die OI sollen die Daten gemessen werden die aus dem Speicher geladen werden müssen.

Intel VTUNE Alternativ wurde versucht, das Intel-Analyse-Tool Intel VTune zu verwenden. Mit Intel VTune lassen sich auf Intel-Prozessoren Analysedaten über die Laufzeit und Performance auslesen, sowohl über ein CLI-Tool als auch über eine grafische GUI. Intel VTune kann proprietäre Intel-Counter lesen und so die Zugriffe auf den RAM messen, auch wenn das mit anderen Tools nicht möglich ist. Eine VTune-Messung wird, anders als beim LIKWID-Tool, nicht erst im Code gestartet und beendet, sondern misst standardmäßig den gesamten Programmdurchlauf. Um einen bestimmten Bereich zu messen, wie in diesem Fall die Inferenzberechnung, lässt sich entweder ein Filter im Code definieren, nach dem in den Messergebnissen gesucht werden kann, oder die Messung wird beim Start pausiert und erst im relevanten Codeteil mit `resume()` fortgesetzt. Das Messen mit einem Filter war jedoch nicht möglich. Der nach „prefill“ gefilterte Messreport enthielt keine Daten über alle Speicherzugriffe. Die Memory-Zugriffe liessen sich nicht klar einer Task zuordnen. Die Messung wurde deshalb beim Programmstart mit der Option `-start-paused` gestoppt und im Code im zu messenden Bereich mit `__itt_resume()` fortgesetzt. Für die Decode-Phase wurde die Messung im Code nur gestartet, wenn alle Token der Prefill-Phase verarbeitet wurden. Listing 2 zeigt die Befehle mit denen die Intel VTUNE Messung durchgeführt wurde.

Gestartet wurde die Messung mit der Messgruppe memory-access:

```
vtune -collect memory-access -start-paused -knob dram-bandwidth-limits=true
-result-dir vtune_results/decode500_29  -app-working-dir .
-- ./build/bin/llama-cli -m ./models/Qwen3-0.6B-Q4_0.gguf --file longprompt.txt
-b 2048 --no-warmup
```

Zur Abschätzung der Speicherzugriffe wurden Last-Level-Cache-(LLC)-Misses verwendet. LLC-Misses verursachen einen Hauptspeicherzugriff und können daher als Metrik für die effektive Datenbewegung dienen. Die Anzahl der Speicherzugriffe in Byte wurde aus den LLC-Misses durch Multiplikation mit 64 abgeleitet, da pro Miss eine Cache-Line (64 Byte) nachgeladen wird. Um sicherzustellen, dass vor Messbeginn keine wiederverwendbaren Daten im Cache verbleiben und Zugriffe tatsächlich in LLC-Misses münden, wurde der Cache durch das Laden von Testdaten zuvor geflusht. Die Messungen mit dem Intel VTUNE Tool zeigten jedoch, dass die LLC-Cachemisses nicht den realistischen Speicherverbrauch wiederspiegeln.
Grund dafür ist vor allem die zu kurze Laufzeit der Inferenz. In den Messergebnisse taucht die Meldung:

```
Recommendations:
Increase execution time:
| Application execution time is too short. Metrics data may be unreliable.
```

```
| Consider reducing the sampling interval or increasing your application
| execution time.
```

auf. Die deutlich zu niedrigen Cache-Misses deuteten darauf hin, dass Intel VTune nicht genug Zeit hat, die Hardware-Counter auszulesen. Es wurden deshalb zwei Maßnahmen ausprobiert. Zum einen wurde mit der Option `-knob sampling-interval` das Messintervall erhöht. Zum anderen wurde der Codeblock, in dem die Inferenz durchgeführt wird, in einer Schleife wiederholt gemessen und anschließend der Durchschnitt der Messergebnisse verwendet. Mit beiden Änderungen lieferte VTune dennoch weiterhin unrealistisch niedrige Ergebnisse. Stattdessen wurde der Speicherverbrauch theoretisch analysiert.

Theoretischer Speicherverbrauch Da die Messung mit Intel VTune unrealistische Werte lieferte, wurde der benötigte Speicherverbrauch für die Inferenz in verschiedenen Szenarien theoretisch ermittelt. Dies ist möglich, da die Größe der benötigten Parameter sowie der KV-Caches des Modells bekannt sind. Der restliche Datenverbrauch ist im Vergleich zu den Gewichten und Caches vernachlässigbar.

KV Cache `llama.cpp` printed bei der Inferenzausführung die Größe des KV Cache auf der Konsole aus. Diese Größe ist jedoch der für den maximalen Kontext benötigte Platz und spiegelt nicht die wirklich verwendeten Daten wieder. Um die verwendeten Daten des KV Caches für einen bestimmten Decode Befehl zu bestimmen, werden die benötigten Bytes für einen Eintrag mit der Anzahl der darin gespeicherten Embeddings und der Anzahl Layer mutlipliziert. Die Größe des K und V Caches sind dabei gleich. Aus den Prints von `llama.cpp` geht hervor, dass die Daten im F16 Format gespeichert werden und somit 2 byte brauchen: `type_k = 'f16', type_v = 'f16'`

$$S_{\text{KV,token}} = 2 \times n_{\text{layers}} \times n_{\text{embd\_kv}} \times b_{\text{dtype}}$$

Für das verwendete Qwen3-0.6B-Modell mit $n_{\text{layers}} = 28$, $n_{\text{embd\_kv}} = 1024$ und FP16-Präzision ergibt sich:

$$S_{\text{KV,token}} = 2 \times 28 \times 1024 \times 2 \text{ Bytes} = 114,688 \text{ Bytes} \approx 112 \text{ KB}$$

Bei einer initialen Kontextlänge von $L = 832$ Tokens nach der Prefill-Phase beträgt die Gesamtgröße des KV-Cache:

$$S_{\text{KV,832}} = L \times S_{\text{KV,token}} = 832 \times 114,688 \text{ Bytes} = 95,420,416 \text{ Bytes} \approx 91 \text{ MB}$$

Die KV-Cache-Größe pro Layer errechnet sich zu:

$$S_{\text{KV,layer}} = \frac{S_{\text{KV,total}}}{n_{\text{layers}}} = \frac{91 \text{ MB}}{28} \approx 3,25 \text{ MB}$$

Embeddings Das verwendete Qwen3-0.6B Modell ist 382,2 MB groß. Neben den Gewichten benötigt vor allem das Vocabulary, aus dem die ersten Embedding-Werte ausgelesen werden, viel Platz. Für Qwen3-0.6B wird eine Vokabulargröße von 151.936 Einträgen angenommen. Dies lässt sich aus dem Operatorbaum auslesen. Die Werte sind Q6_K-quantisiert und benötigen effektiv etwa 6.5 Bit pro Gewicht. Bei einer Embedding-Dimension von 1024 für jedes Token aus dem Vocabulary ergibt sich dafür ein Speicherbedarf von:

$$S_{\text{embed}} = n_{\text{vocab}} \times n_{\text{embd}} \times \frac{6,125}{8} \text{ Bytes}$$

$$S_{\text{embed}} = 151.936 \times 1024 \times \frac{6,125}{8} \approx 113,5 \text{ MB}$$

Ausgehend von der Gesamtgröße der Modelldatei von ca. 358,8 MB abzüglich der Embedding-Schicht verbleiben für die Transformer-Layer:

$$S_{\text{layers}} = S_{\text{total}} - S_{\text{embed}} = 358,8 \text{ MB} - 113,5 \text{ MB} = 245,3 \text{ MB}$$

Logits Im letzten Layer eines LLM wird nach der Verarbeitung der Token-Embeddings durch die Attention- und FFN-Schichten der finale Hidden-State jedes Tokens mit der Output-Embedding-Matrix multipliziert, um für jedes Token Wahrscheinlichkeiten über das gesamte Vokabular zu berechnen. Die Größe der entsprechenden Matrix ist [Anzahl der zu verarbeitenden Tokens, Vokabulargröße].
Aus dem Operatorbaum von Qwen3 0.6B ergibt sich, dass die Werte in F16 Genauigkeit gespeichert

werden. Jeder Wert belegt somit 2 Bytes. Damit ergibt sich für die resultierende Logit-Matrix bei einem Token:

$$1\,\text{Token} \cdot 151.936\,(\text{Vokabulargröße}) \cdot 2\,\frac{\text{Bytes}}{\text{Wert}} = 303.872\,\text{Bytes} \approx 296,75\,\text{KiB} \approx 0,30\,\text{MB}$$

Berechnung für 832 Token:

$$832\,\text{Token} \cdot 151.936\,(\text{Vokabulargröße}) \cdot 2\,\frac{\text{Bytes}}{\text{Wert}} = 252.821.504\,\text{Bytes} \approx 241,11\,\text{MiB} \approx 253\,\text{MB}$$

Speicherverbrauch Für den Input von nur einem Token werden die Grössen des KV Cache und der Logit Berechnungen als vernachlässigbar angenommen. Für das Verhältnis der Gewichte von der Attention-Schicht zu FFN-Schicht wird von einer Verteilung von 60 Prozent für das FFN Netz und 40 für den Attention Mechanismus ausgegangen.

Prefill vs Decode

- 1 Token, 28 Layer Prefill:
  Gewichte: 261,9 MB
  Gesamt: 261,9 MB
- 832 Token, 28 Layer Prefill:
  Gewichte: 261,9 MB
  KV Cache: 91 MB
  Logits: 253 MB
  Gesamt: 605,9 MB
- 1 Token, 28 Layer Decode:
  Gewichte: 261,9 MB
  Gesamt: 261,9 MB
- 832 Token, 28 Layer Decode:
  Gewichte: 261,9 MB
  KV Cache: 91 MB
  Logits: 253 MB
  Gesamt: 605,9 MB

Attention vs FFN

- 1 Token 28 Layer Prefill Attention:
  Gewichte: 104.76 MB
  Gesamt: 104.76 MB
- 832 Token 28 Layer Prefill Attention:
  Gewichte: 104.76 MB
  KV Cache: 91 MB
  Logits: 253 MB
  Gesamt: 448.76 MB

3.3.4 Ergebnisse

Der Roofline Graph plottet für eine Anwendung auf der X-Achse die Operational Intensity, gemessen in FLOPS pro verwendetem Byte und auf der Y-Achse die FLOPs pro Zeit (in Millionen). Im resultierenden Graph ist eine Anwendung rechengebunden, falls sie rechts vom Knickpunkt (Ridge Point) liegt. Liegt sie links davon ist sie Speichergebunden. Die MFLOPS/s lassen sich direkt aus der LIKWID Messung auslesen. Bei der Messung des ersten Decode Schrittes musste wie bei der Messung der FLOPs die Differenz zur Prefill Phase genommen werden. Wichtig ist zudem, dass die Laufzeiten der Prefillphase deutlich länger ist als die der späteren Decode Phasen. Der Grund dafür ist der Overhead für das Erstellen der Kernel und anderer Initialisierungen. Anders als die FLOPs sind diese Messungen auch nicht deterministisch und verändern sich bei mehreren Durchläufen. Dies muss bei einer Reproduktion der Ergebnisse beachtet werden. Für die FLOPs/Byte wurden die Ergebnisse aus der theoretischen Überlegung zum Speicherverbrauch verwendet.

Prefill vs Decode Der Vergleich der Prefill- und Decode-Phase entspricht dem erwarteten Verhalten. Bei einem kleinen Input von nur einem Token bleibt die Rechenintensität sehr niedrig, da nur kleine Matrizen multipliziert werden müssen und trotzdem alle Modellgewichte geladen werden. Vor allem die Prefill-Phase mit einem langen Input-Prompt ist rechenintensiv und durch die Geschwindigkeit der Hardware limitiert.

Performance:

- **Prefill Phase**

- **1 Token:** SP [MFLOP/s] : 10.6585
- **832 Token:** SP [MFLOP/s] : 11666.0821

- **Decode Phase**
  - **1 Token:** SP [MFLOP/s] : 10.0118 (Differenz)
  - **832 Token:** SP [MFLOP/s] : 186.5863 (Differenz)

Operational Intensity:

- **Prefill Phase**
  - **1 Token:**
$$\frac{72.616.897 \text{ FLOPs}}{274.614.845 \text{ Bytes}} \approx 0,26 \text{ FLOPs/Byte}$$
  - **832 Token:**
$$\frac{172.237.816.894 \text{ FLOPs}}{635.364.884 \text{ Bytes}} \approx 271,08 \text{ FLOPs/Byte}$$

- **Decode Phase**
  - **1 Token:**
$$\frac{68.830.793 \text{ FLOPs}}{274.614.845 \text{ Bytes}} \approx 0,25 \text{ FLOPs/Byte}$$
  - **832 Token:**
$$\frac{295.993.279 \text{ FLOPs}}{635.364.884 \text{ Bytes}} \approx 0,47 \text{ FLOPs/Byte}$$

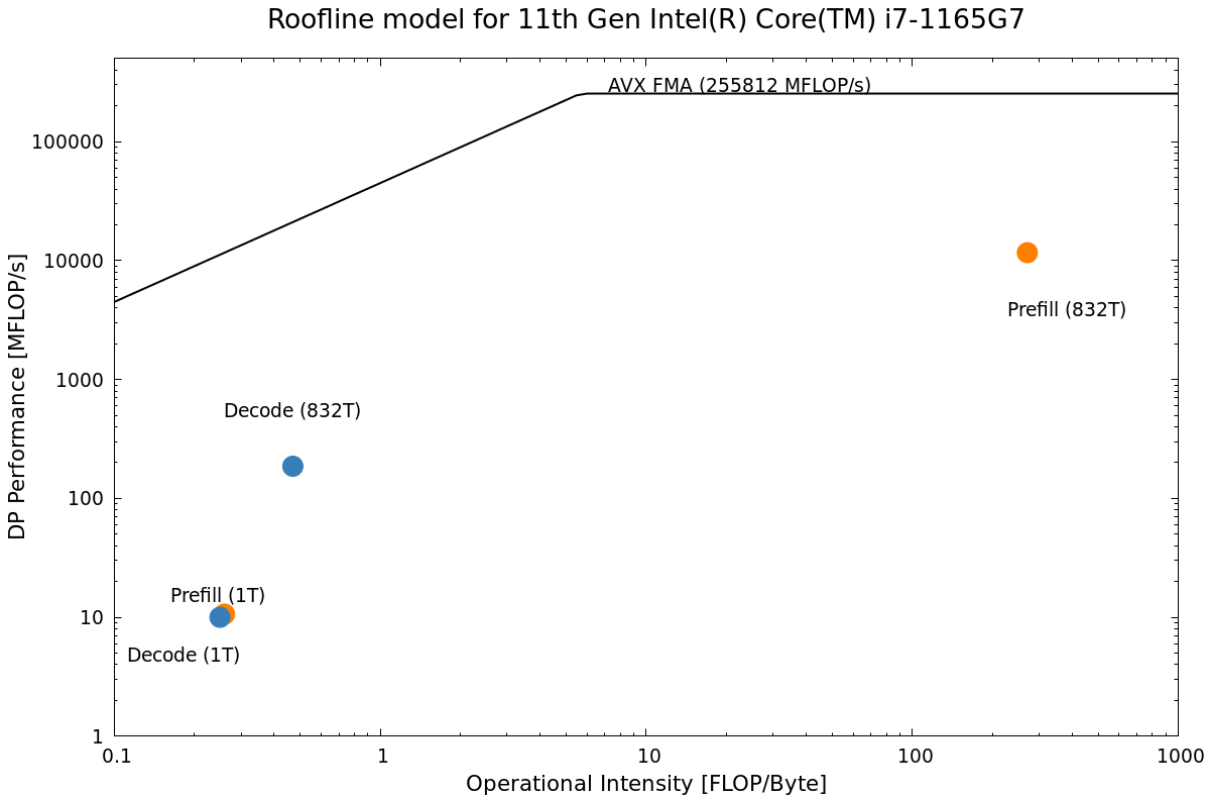


Abbildung 29: Roofline-Modell für die Prefill- und Decode-Phasen mit einem und 832 Token

Attention vs FFN Literaturrecherchen zeigten, dass das FFN bis zu einer bestimmten Kontextlänge den größten Rechenaufwand verursacht. Ab einer größeren Kontextgröße übersteigt jedoch der Aufwand des Attention-Mechanismus den des FFN, wobei der genaue Kipppunkt vom Modell und der Architektur abhängt. Zudem nimmt der relative Anteil des FFN-Aufwands bei kleinen Modellen ab, während er bei großen Modellen mit vielen Gewichten höher ausfällt. Die Messung der Anzahl an Flops für das Qwen3 0.6 Modell zeigte, dass der relativ Anteil des FFN am Gesamtaufwand bereits bei kleinen Kontextgrößen vergleichsweise gering ist. Für den konkreten Fall der Auslagerung einzelner Layer-Komponenten auf ein FPGA ergibt sich daraus, dass sich die Auslagerung des Attention-Mechanismus auch bei kleinen Kontextgrößen lohnt. Der FFN-Teil dominiert die Berechnung in diesem Modell nicht in dem Maße wie bei größeren Modellen.

- **1 Token**
  - **Ganzes Layer:** SP [MFLOP/s] : 10.6585
  - **Attention Block:** SP [MFLOP/s] : 6.1451
- **832 Token**
  - **Ganzes Layer:** SP [MFLOP/s] : 11666.0821
  - **Attention Block:** SP [MFLOP/s] : 8592.0133

Operational Intensity:

- **1 Token**
  - **Ganzes Layer:**
$$\frac{72.616.898 \text{ FLOPs}}{274.614.845 \text{ Bytes}} \approx 0,26 \text{ FLOPs/Byte}$$
  - **Attention Block:**
$$\frac{40.758.733 \text{ FLOPs}}{104.760.000 \text{ Bytes}} \approx 0,39 \text{ FLOPs/Byte}$$
- **832 Token**
  - **Ganzes Layer:**
$$\frac{172.237.816.645 \text{ FLOPs}}{635.364.884 \text{ Bytes}} \approx 271,08 \text{ FLOPs/Byte}$$
  - **Attention Block:**
$$\frac{156.262.560.384 \text{ FLOPs}}{449.520.000 \text{ Bytes}} \approx 347,62 \text{ FLOPs/Byte}$$

4 FPGA-Implementierung

Ein wichtiger Teil dieser Arbeit besteht aus theoretischen Überlegungen zum Ablauf und zur Rechenintensität kleiner Large Language Models. Um die erarbeiteten Ergebnisse praktisch anzuwenden, wurde versucht, ein LLM auf einem FPGA lauffähig zu machen. Die theoretischen Ergebnisse können dabei helfen, die rechenintensiven Teile der Inferenz auf das FPGA auszulagern.
Dafür wurde das *LLaMa.cpp*-Projekt angepasst, um es auf einem FPGA auszuführen. Grundlage war die Ausführung des Qwen3-0.6B-Modells. Verwendet wurden nur die Teile von *LLaMa.cpp*, die für die Textinferenz mit diesem Modell nötig sind. Zum Einsatz kamen zwei am Lehrstuhl 3 für Informatik der FAU entwickelte Tools. Mit dem Tool *easyocl*[36] lassen sich Kernel mit einer vereinfachten OpenCL-Abstraktion starten. Um die OpenCL-Kernel für das FPGA zu kompilieren, wurde das *brig-tool* verwendet. Folgende Schritte wurden dabei umgesetzt:

[36] `gitlab.rrze.fau.de/cs3/private/projects/povos/easy-opencl-lib/`

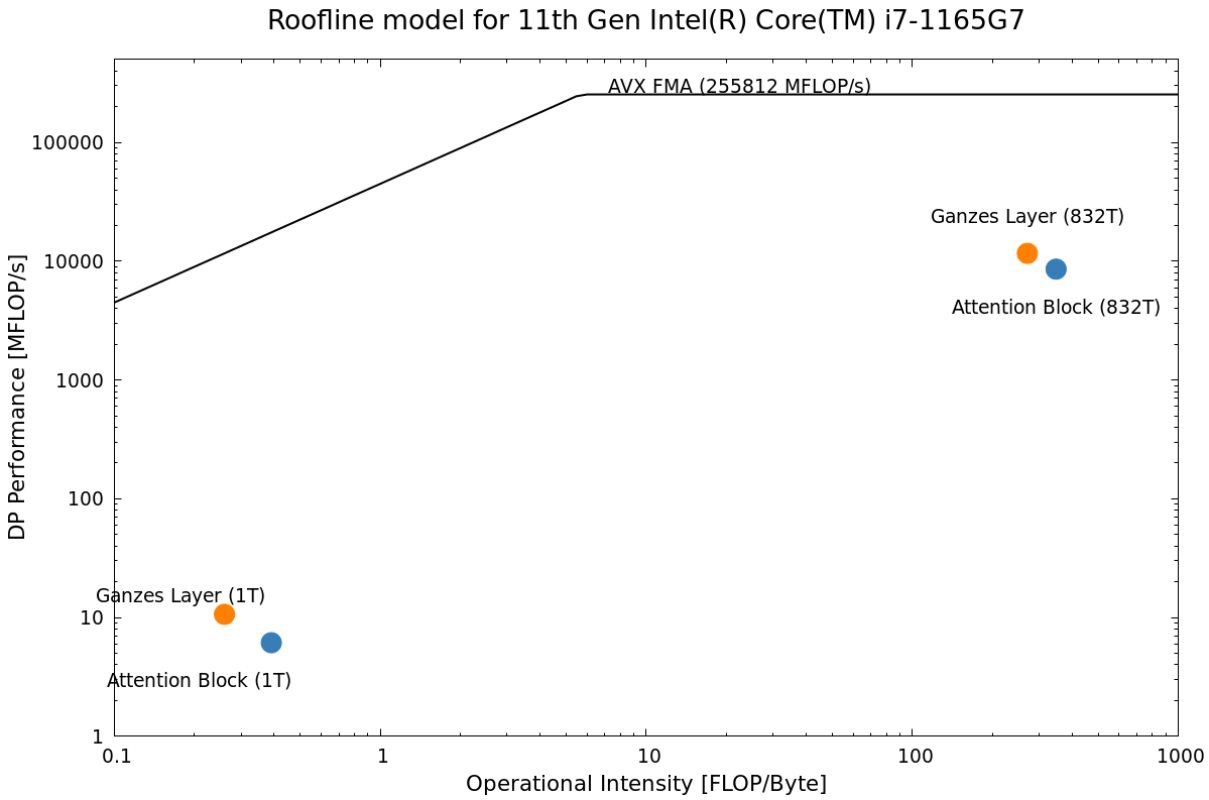


Abbildung 30: Roofline-Modell für den Attention Block und das komplette Layer mit einem und 832 Token

1. `llama.cpp` soll auf diejenigen Funktionen reduziert werden, die für die Ausführung von Qwen3 0.6B benötigt werden
2. OpenCL Code im HostCode soll durch Aufrufe des easyOCL Tool ersetzt werden
3. Die OpenCL-Kernel werden so modifiziert, dass sie sich mit dem BRIG-Compiler hardwarenah übersetzen lassen

Im Folgenden werden die Anpassungen beschrieben, die am `llama.cpp` Projekt vorgenommen wurden um mit diesen zwei Tools die Ausführung auf einem FPGA zu ermöglichen.

Abschließend wird in einem Ausblick gezeigt, welche weiteren Änderungen nötig währen um anderen Modelle auszuführen.

4.1 LLaMA.cpp Reduktion

Das `llama.cpp`-Projekt sowie die intern verwendete GGML-Bibliothek unterstützen eine Vielzahl an Modellen und mehrere Hardware-Backends. Im ersten Schritt wurden alle Teile entfernt, die für die Ausführung von Qwen3 über OpenCL-Kernel nicht benötigt werden. In der GGML-Implementierung wurde dazu der Code für andere Backends wie Vulkan oder CUDA gelöscht.
In `llama.cpp` wurde der Code ebenfalls stark reduziert, sodass nur noch das Qwen3-Modell erzeugt wird. In der Datei `llama-model.cpp` befindet sich der Code, der für alle unterstützten Modelle den Berechnungsgraphen erstellt. Diese Datei kann deutlich verkleinert werden, indem ausschließlich der Code für das gewünschte Modell beibehalten wird.

Graph Export Anstatt den Code für die nicht benötigten Modelle zu löschen, wurde versucht, den gesamten Schritt der Grapherstellung zu überspringen. In einem Github-Beitrag von Georgi Gerganov[37] wird die Möglichkeit erwähnt, den GGML-Berechnungsgraphen zu exportieren und später wieder einzulesen, sodass die Berechnungen direkt ausgeführt werden können.

[37] `https://github.com/ggml-org/llama.cpp/discussions/915`

Es sollte untersucht werden, ob sich der Berechnungsgraph einer LLM-Architektur einmalig erstellen und anschließend auf dem FPGA nur noch mit geänderten Eingabedaten laden und ausführen lässt. Dadurch wäre deutlich weniger Code aus `llama.cpp` notwendig.
In einem ersten Schritt wurden die Funktionen zum Ex- und Importieren von Berechnungsgraphen gesucht. Diese waren in einem Commit vom November 2024 entfernt worden. Die Funktionen konnten jedoch wiedergefunden und erfolgreich genutzt werden, um einen Graphen zu exportieren und erneut zu importieren. Der Graph wird dabei im GGML-Format gespeichert.
Die Wiederverwendung des Berechnungsgraphen konnte jedoch nicht umgesetzt werden. In GGML lässt sich ein Berechnungsbaum erstellen, der nur Berechnungsknoten enthält und später mit Input-Daten befüllt werden kann. In `llama.cpp` wird jedoch bei jedem Aufruf von `decode` ein neuer Graph erzeugt: Für jedes neue Token werden Embeddings und Positionskodierungen erneut eingebaut und vollständig berechnet, anstatt lediglich die Eingabedaten zu aktualisieren.
Das damit verbundene Problem beziehungsweise der Performancenachteil durch das wiederholte Erstellen des Graphen wird in mehreren Diskussionsbeiträgen im GitHub-Repository von `llama.cpp` beschrieben. [38] [39] [40] In einer Diskussion wird erläutert, warum die Wiederverwendung des Graphen grundsätzlich nicht möglich ist.[41].
Das Hauptproblem betrifft die KV-Caches und die darin enthaltenen Positionsinformationen. Diese können bei einem erneuten Aufruf nicht einfach gelöscht oder überschrieben werden. Selbst wenn die Input-Daten verändert werden, bleiben die Positionsdaten im KV-Cache erhalten. Georgi Gerganov hat dieses Problem ebenfalls angesprochen und arbeitet nach aktuellem Stand an einer Lösung, um den Berechnungsbaum anzupassen und wiederverwendbar zu machen.[42]

4.2 EasyOCL

Die EasyOCL-Bibliothek wurde entwickelt, um die OpenCL-Programmierung zu vereinfachen. Sie bietet Funktionen, die komplexe OpenCL-Befehle abstrahieren. Da sich die vereinfachten Aufrufe von EasyOCL besser auf einem FPGA ausführen lassen, sollten im `llama.cpp`-Projekt die regulären OpenCL-Aufrufe durch die EasyOCL-Methoden ersetzt werden.
EasyOCL konnte in `llama.cpp` als dynamische Bibliothek eingebunden werden.
llama.cpp und die GGML-Bibliothek mussten an mehreren Stellen für EasyOCL angepasst werden.
EasyOCL verwendet die ältere OpenCL-Version 1.2, die einige der in `llama.cpp` genutzten Befehle nicht unterstützt. Besonders betroffen waren Kernel, die explizite SIMD-Instruktionen nutzen. In den meisten Fällen waren keine weiteren Änderungen nötig, da die SIMD-Größe bereits der Vorgabe entsprach. Nur im Kernel der `rms-norm` mussten die SIMD-Größe und einige Codeabschnitte angepasst werden.
Bei Tests auf einem Intel-Laptop nutzte `llama.cpp` die eingebetteten Kernel. Der Kernel-Code wird dabei während des Build-Prozesses eingebunden und nicht erst zur Laufzeit geladen. Bei der Verwendung mit EasyOCL kam es anfangs zu doppelten Definitionen von Structs, die über Präprozessoroptionen aufgelöst werden mussten.
Die EasyOCL-Methode zum Erstellen aller Kernel eines Verzeichnisses konnte nicht verwendet werden, da ein Fehler auftrat. Stattdessen wurden die Kernel manuell erstellt und wie in `llama.cpp` in Variablen gespeichert.

4.3 Brig Compiler

Um die Kernel auf dem FPGA auszuführen, standen zwei Tools vom Lehrstuhl 3 zur Verfügung: Das Brig-Tool, ein Compiler, der den OpenCL-Code hardwarenah kompiliert und in einer BRIG-Datei abspeichert, sowie das pas-tool, das den kompilierten Code analysiert und überprüft, ob sich der OpenCL-Code hardwarenah abbilden lässt.
Es wurde angenommen, dass Kernel, die vom Brig-Tool kompiliert wurden und die vom pas-tool erfolgreich verarbeitet werden, auch auf einem FPGA lauffähig sind. Damit ein Kernel vom pas-tool akzeptiert wird, mussten die OpenCL-Kernel jedoch an mehreren Stellen angepasst und vereinfacht werden.
Die Hauptprobleme beim Kompilieren des OpenCL-Codes aus `llama.cpp` mit dem Tool waren:

- Die Subgroup-Extension

[38] `https://github.com/ggml-org/llama.cpp/discussions/1548`
[39] `https://github.com/ggml-org/llama.cpp/pull/8366`
[40] `https://github.com/ggml-org/llama.cpp/discussions/1548#discussioncomment-6010244`
[41] `https://github.com/ggml-org/llama.cpp/issues/71#issuecomment-1475276375`
[42] `https://github.com/ggml-org/llama.cpp/pull/8366#issuecomment-2987224974`

- Integer-Division
- Komplexe Schleifen und Kontrollstrukturen
- Mathematische Funktionen
- Konvertierungsmethoden
- Nutzung von lokalem Speicher

Funktionalität zum Auslesen der lokalen Id und Workgruppen-Id war zunächst nicht vorhanden, wurde jedoch zu dem Brig-Compiler ergänzt.
Das Debugging bzw Verändern des Originalcode war in Teilen aufwändig, da das Pas-Tool problematische Codezeilen nicht benennt. Dadurch blieben eigentliche Fehlerquellen oft unentdeckt, oder andere, unproblematische Aufrufe wurden fälschlicherweise als fehlerhaft markiert. Die Anpassungen wurden ausschließlich für die Kernel des Modells Qwen3 0.6B durchgeführt. Außerdem wurden manche Änderungen speziell für dieses Modell umgesetzt, beispielsweise durch fest einprogrammierte Parameter.

Validierung Um zu überprüfen, ob die für den Brig-Compiler angepassten Kernel die gleichen korrekten Ergebnisse wie die Originale liefern, wurden zunächst separate Testprogramme implementiert. Dazu wurden OpenCL-Host-Dateien erstellt, die sowohl den Original-Kernel als auch den angepassten Kernel mit identischen Eingabedaten ausführen und die resultierenden Ausgabetensoren miteinander vergleichen. Die Aufrufparameter wurden aus dem Modell Qwen3 0.6B übernommen. Durch diese Methodik konnte festgestellt werden, ob und an welchen Positionen innerhalb des Ausgabetensors Abweichungen auftreten sowie in welchem Ausmaß sich die Ergebnisse unterscheiden. Als zweite Validierungsmethode wurden die angepassten Kernel testweise in die LLM-Berechnung integriert und die erzeugten Textausgaben mit denen der Original-Kernel verglichen. Im Abschnitt 2.2.4, in dem die Funktionsweise von LLMs beschrieben wird, wurde bereits die Sampling-Technik zur Auswahl des nächsten Tokens sowie die Rolle der Temperaturparameter erläutert. Wird die Temperatur auf null gesetzt, wird deterministisch das wahrscheinlichste nächste Token gewählt. Bei wiederholtem Aufruf mit identischem Eingabeprompt erzeugt das LLM dadurch reproduzierbare Ergebnisse. Auf diese Weise konnten die angepassten Kernel in `llama.cpp` integriert und hinsichtlich ihrer Korrektheit überprüft werden. Nach der Integration zeigte sich jedoch, dass angepasste Kernel, deren Ausgabetensoren zwar nur geringfügig vom Original abwichen, trotzdem deutlich unterschiedliche Textausgaben erzeugten. Die Ursache hierfür liegt vermutlich im dynamischen Verhalten von LLMs. Parameter wie die Tensorgröße verändern sich während der fortlaufenden Token-Generierung, wodurch bereits kleine numerische Abweichungen zu unterschiedlichen Ergebnissen führen können. Solche Effekte werden durch statische Tests nicht vollständig erfasst. Der Aufwand zum Erstellen separater Testdateien für jeden Kernel erwies sich zudem als deutlich höher als das direkte Einsetzen in `llama.cpp`. Für zukünftige Arbeiten bietet es sich daher an, die für den Brig-Compiler angepassten Kernel zunächst direkt in `llama.cpp` zu testen, bevor separate Testprogramme implementiert werden.

Reduktion Die OpenCL-Kernel in `llama.cpp` verwenden an vielen Stellen die Subgroup-Erweiterung, insbesondere `sub_group_reduce`-Funktionen zur effizienten Aggregation von Zwischenergebnissen. Da der Brig-Compiler diese nicht unterstützt, musste der Code angepasst werden.
Als Ersatz schreiben die Work-Items ihre Werte in ein lokales Array, das vom ersten Work-Item aufsummiert wird. Zum Zeitpunkt der Implementierung konnte der Brig-Compiler jedoch keine lokalen Arrays deklarieren, die für alle Work-Items einer Gruppe sichtbar sind. Deshalb wurden diese Arrays als Kernel-Parameter übergeben.
Diese Anpassung wurde in folgenden Kerneln durchgeführt:

- Mul_Mv_f16_f32_l4
- mul_mv_q4_0_f32_8x_flat
- RMS-Norm (hier wurde kein lokaler Speicher übergeben)
- Softmax

```
if (local_id == 0) {
        for (int i = 0; i < local_size; i++) {
            workgroup_sum += sum[i];
        }
        sum = workgroup_sum;
        for (int i = 4 * (ne00 / 4); i < ne00; i++) {
            sum += x_scalar[i];
        }
        sum /= ne00;
    }
```

Komplexe Schleifen In mehreren Kerneln traten Fehler auf, wenn Schleifen mit variabler Abbruchbedingung komplexe Berechnungen im Schleifenrumpf enthielten. Dieses Problem betraf die Kernel RMS-Norm, Softmax und die quantisierte Matrix-Vektor-Multiplikation.

Die Fehlermeldungen die auftraten waren:

- *could not find FSM path between branch and loop out state*
- *could not find corresponding loop of register*

Die Meldung *could not find corresponding loop of register* trat im RMS-Norm-Kernel während der Reduktion der Work-Item-Ergebnisse auf: Dieser Fehler tritt nur auf, wenn `ne00` als Schleifenvariable verwendet wird. Daher wurde die Größe von `ne00` für die zwei möglichen Werte fest in den Kernel einprogrammiert.

Float Division Zur Berechnung der Speicheradresse eines Elements dividieren die Kernel die globale Work-Item-ID durch die Tensorlängen der Dimensionen. Beispiel im Copy-Kernel:

```
int i3 = n / (ne2ne1ne0);
int i2 = (n - i3ne2ne1ne0) / (ne1ne0);
int i1 = (n - i3ne2ne1ne0 - i2ne1ne0) / ne0;
int i0 = (n - i3ne2ne1ne0 - i2ne1ne0 - i1*ne0);
```

Da der Brig-Compiler keine direkte Integer-Division unterstützt, wurden Divisionen über Fließkommazahlen realisiert:

```
int i3 = (int)((float)n / (float)(ne2ne1ne0));
int i2 = (int)((float)(n - i3ne2ne1ne0) / (float)(ne1ne0));
int i1 = (int)((float)(n - i3ne2ne1ne0 - i2ne1ne0) / (float)ne0);
int i0 = (n - i3ne2ne1ne0 - i2ne1ne0 - i1*ne0);
```

Diese Anpassung führte jedoch zu fehlerhaften Ausgaben. Die Ursache sind Rundungsfehler: Ein Wert, der mathematisch exakt 822 sein sollte, wird als 821.999997 dargestellt. Nach dem Cast zu int ergibt dies 821 statt 822. Der Fehler in der Adressberechnung führte zu verfälschten Ausgaben, die nur noch zufällige Zeichen lieferten. Diese Rundungsfehler konnten behoben werdenn indem ein epsilon auf die Werte addiert wird. Zusätzlich besteht bei großen Tensoren ein Präzisionsproblem: Ein 32-Bit-Float kann nur bis 2^{24} (16,7 Mio.) exakt darstellen. Werden größere Werte verwendet, treten zwangsläufig Rundungsfehler auf. Dies ist bei großen Matrizen, etwa mit `ne0 = 1024`, `ne1 = 1024` und `ne2 = 16`, der Fall.

Sonstige Probleme Bei der Berechnung der Attention-Maske in der Softmax-Funktion:

```
global float4 * pmask = src1 != src0 ? (global float4 *)(src1 + i01*ne00) : 0;
```

gibt der Brig-Compiler den Fehler: *extracted CMP instruction is not constant* aus.

4.3.1 Requantisierung

Das von Hugginface heruntergeladenen Qwen3 0.6B Modell enthielt größtenteils Tensoren im FP32 und Q4_0 quantisierten Format, jedoch auch wenige Tensoren im Q4_1 und Q6 quantisiertem Format:

```
llama_model_loader: - type  f32:  113 tensors
llama_model_loader: - type q4_0:  193 tensors
llama_model_loader: - type q4_1:    3 tensors
llama_model_loader: - type q6_K:    1 tensors
```

Für die Tensormultiplikation existieren in der GGML-Library mehrere Kernel-Implementierungen, die je nach Datentyp der Tensoren ausgewählt werden. Die Multiplikations-Kernel für Q6- und Q4_1-quantisierte Tensoren sind aufwändiger als der für Q4_0-Tensoren. Um die Komplexität zu reduzieren und weniger Kernel anpassen zu müssen, wurden die Q4_1 und Q6_K Tensoren im Modell in Q4_0-Tensoren umgewandelt. Dafür kam das Tool llama-quantize zum Einsatz.

Grund der Quantisierung Das quantisierte Qwen-Modell von Huggingface verwendet für manche Berechnungen FP32-genaue Tensoren und für andere Q4-quantisierte Tensoren. Dies liegt an den unterschiedlichen Genauigkeitsanforderungen verschiedener Berechnungen. Besonders in komplexen Operationen, wie etwa dem Ziehen einer Wurzel, können geringe Abweichungen in der Eingangspräzision einen großen Einfluss auf das Ergebnis haben. Für solche Berechnungen sollten daher FP32-Tensoren genutzt werden.

Die Input- und Output-Tokens des Qwen3 0.6B werden in FP32 verarbeitet. Aus mehreren Arbeiten zur Quantisierungsgenauigkeit geht jedoch hervor, dass 4-Bit-quantisierte Gewichte in der Regel ausreichend Genauigkeit bieten. Erst bei Quantisierungen unter 4 Bit kommt es zu deutlichen Qualitätseinbußen. Dumitru et al. [8] zeigen, dass das Komprimieren der Tensoren von BF16 auf eine 4-Bit-Kodierung nur einen geringen Performanceverlust zur Folge hat. Die Autoren verweisen zudem auf mehrere andere Arbeiten, die ebenfalls zu dem Schluss kommen, dass 4-Bit-Quantisierung robuste Ergebnisse liefert. Auch Jin et al. [17] zeigen, dass 4-Bit-Quantisierung im Vergleich zu nicht quantisierten Tensoren nur einen sehr geringen Performanceverlust verursacht.

LLaMa-Quantize Das `llama.cpp`-Projekt stellt mit llama-quantize ein Tool zur Quantisierung von Modellen bereit. [43] Dabei lassen sich nicht nur komplette Modelle, sondern auch einzelne Layer oder spezifische Tensor-Typen quantisieren.
Die Option *–allow-requantize* ermöglicht es, bereits bestehende Quantisierungen zu überschreiben. Mit der Option *–tensor-type* können bestimmte Tensor-Typen eines Modells in eine gewünschte Quantisierungsform umgewandelt werden. Diese Option unterstützt Regex-Syntax, womit sich mehrere Tensor-Typen gleichzeitig anpassen lassen. Die Option *–output-tensor-type* erlaubt die Änderung des Output-Tensors eines Modells.
Die Dokumentation des Tools enthält jedoch keine Angaben zur genauen Syntax der Tensor-Namen. Mithilfe eines Blogposts [44], in dem ein Nutzer die Quantisierung einzelner Tensoren demonstriert und Messungen durchführt, konnten die ursprünglich Q6-quantisierten Tensoren erfolgreich in Q4-Form gebracht werden. Dadurch ist der rechnerisch aufwändige Kernel zur Multiplikation von Q6-Quantisierten Tensoren nicht mehr nötig. Durch aufwändige mathematische Operationen wäre es schwer gewesen, diesen Kernel für die Verwendung mit dem brig-tool anzupassen.
Das verwendete Basismodell, das nahezu ausschließlich aus Q4- und FP32-Tensoren besteht, hatte eine Größe von 469,1 MB. Nach dem Requantisieren der Q6- und Q4_1-Tensoren mit dem Quantize-Tool verringerte sich die Modellgröße auf 429 MB, also um fast 10 Prozent. Das Modell besteht dadurch aus folgenden Tensortypen:

```
llama_model_loader: - type  f32:  113 tensors
llama_model_loader: - type q4_0:  198 tensors
```

4.3.2 Kernelspezifische Anpassungen

Für die spezifischen Berechnungen in den einzelnen Kerneln waren weitere Änderungen nötig. Im Folgenden werden diese Anpassungen beschrieben und erklärt, ob und wie sich die Kernel auf dem FPGA ausführen lassen.

Silu Der SiLU-Kernel, der in Qwen3 als Aktivierungsfunktion verwendet wird, nutzt im Original die `exp()`-Funktion:

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

Da der Brig-Compiler `exp()` nicht unterstützt, wurde zunächst eine schnelle Sigmoid-Approximation mit `fabs(x)` getestet. Diese lieferte jedoch unbrauchbare Ergebnisse. Als Lösung wurde eine eigene Exponentialfunktion integriert, die den Term mithilfe einer Taylor-Reihe approximiert. Diese Implementierung konnte erfolgreich kompiliert werden, ist für FPGA-Hardware jedoch wahrscheinlich zu komplex.

[43] `https://github.com/ggml-org/llama.cpp/blob/master/tools/quantize/README.md`
[44] `https://github.com/ggml-org/llama.cpp/discussions/12741`

Rms Im RMS-Norm-Kernel trat erneut das FSM-Problem auf. Dieses wurde durch Hardcoding von `ne00` umgangen. Die Tensorgröße `ne00` ist fix und verändert sich nicht über die Tokens hinweg. Für Qwen3 0.6B ergeben sich feste Werte von `ne00=128` und `ne00=1024`. Zwei Kernelvarianten wurden erstellt, und im Host-Code *ggml-opencl.cpp* wird je nach Größe die passende Version aufgerufen.
Da die Subgroup-Extension nicht unterstützt wurde, werden die Werte aller Workitems nicht mehr per `sub_group_reduce_add` summiert, sondern vom Workitem mit *Id=0* über ein Array aggregiert. Der ursprüngliche Reduktionscode:

```
for (uint i = get_local_size(0) / get_max_sub_group_size() / 2; i > 0; i /= 2) {
    if (get_local_id(0) < i) {
        sum[get_local_id(0)] += sum[get_local_id(0) + i];
    }
}
```

führte zum Fehler: *ASAP SCHEDULER: cannot find loop condition CFG node for replication.* Die Reduktion wurde daher so angepasst, dass ausschließlich das erste WorkItem die Summe berechnet.

Softmax Auch die Softmax-Kernel verwenden Subgroup-Funktionen, die durch manuelle Reduktionen ersetzt wurden.
Zusätzlich enthält der Kernel Code für ALiBi. Der Brig-Compiler konnte die `pow()`-Funktion, die in ALiBi genutzt wird, nicht übersetzen. Da Qwen3 RoPE als Positionskodierung nutzt, ist ALiBi jedoch überflüssig. Der entsprechende Code wurde auskommentiert.
Für die Exponentialberechnung wurde eine eigene Approximationsfunktion implementiert, die Taylorreihen und Bit-Shifting für 2^n nutzt.
Das FSM-Problem blieb jedoch bestehen, da im Kernel drei Schleifen über die Tensorlänge laufen (Maximum, Softmax-Berechnung, Rückschreiben). Funktionen mit `float4`-Inputs erzeugten dabei den Fehler *could not find FSM path between branch and loop out state.*
Ein Hardcoding von `ne00` war hier nicht möglich, da die Größe der Attention-Scores mit jedem neuen Token wächst. Somit war der Softmax-Kernel nicht für den Brig-Compiler nutzbar und muss auf der CPU berechnet werden.

Mul F16 Im Kernel `mul_mv_f16_f32_l4` werden die F16-Werte im Original mit `convert_float()` in F32 konvertiert. Da diese Funktion nicht unterstützt wird, wurde eine eigene Umwandlung entwickelt: Bit-Shifts extrahieren Vorzeichen, Mantisse und Exponent und setzen sie ins F32-Format ein.
Modulo-Operationen und Integer-Divisionen für Adressberechnungen wurden durch `while`-Schleifen ersetzt. Zur Reduktion wurden die Zwischensummen aller Workitems in einem lokalen Array `agg0` gespeichert, das als Parameter übergeben wird. Aggregiert werden die Werte wieder vom Workitem mit *Id=0*.

Mul Q4 Im quantisierten Multiplikationskernel konnte der Brig-Compiler den Datentyp `half` nicht verarbeiten. Der Skalierungsfaktor für die Dequantisierung wurde stattdessen als `uint16_t` übergeben und anschließend in Float umgerechnet.
Im Original werden 16 Werte von `y` summiert. Dies überforderte den Brig-Compiler. Stattdessen wurden die Berechnungen auf Blöcke von 8 Werte reduziert und Zwischenergebnisse gespeichert. Die Hauptschleife ruft im Original 8-mal nacheinander die Multiplikationsmethode auf. Der Brig-Compiler konnte dies nicht verarbeiten und erzeugte den Fehler *could not find FSM path between branch and loop out state.*
Als Lösung werden nun jeweils 3 Werte multipliziert und in einem `float3` gespeichert. Dafür musste `N_DST` im Kernel und Host-Code geändert werden, um Startzeilen und Workitems korrekt anzupassen.

Mul Die Berechnungen im `Mul.cl`-Kernel sind vergleichsweise einfach. Nur eine Modulo-Operation konnte nicht übersetzt werden und wurde durch einen Float Cast umgesetzt.

RoPE Der RoPE-Kernel scheiterte aufgrund der benötigten mathematischen Funktionen. `pow()` und `floor()` konnten durch eigene Implementierungen ersetzt werden. Bei den Sinusfunktionen war dies nicht möglich, sodass die Ausgaben in `llama.cpp` deutlich von den erwarteten Werten abwichen.

Cpy Im Copy-Kernel mit F16-Genauigkeit (`kernel_cpy_f32_f16`) musste wieder eine eigene F16-zu-Float-Konvertierung implementiert werden.
Im Gegensatz zu anderen Kernen konnten die Integer-Divisionen für die Startposition hier nicht durch Fließkommadivisionen ersetzt werden, da die Ergebnisse sonst falsch waren.

4.3.3 Zusammenfassung

Die in Qwen 0.6B verwendeten OpenCL-Kernel konnten mit folgenden Einschränkungen mit dem Brig-Tool kompiliert werden:

- **add:** Keine Änderungen
- **cpy:**
  - Int-Division wude zu float gecastet
  - ushort statt half → Konvertierungsmethode hinzugefügt
- **get_rows:** Keine Änderungen
- **mul_mv_f16_f32_l4**
  - Int-Division wude zu float gecastet
  - convert_float wurde duch eine eigene Konvertierungsfunktion ersetzt
  - Subroup Funktion durch Reduktion ersetzt → überegebenes Array als Zwischenspeicher
  - ushort statt half
  - Speicherberechnung für dst: Multiplikation als loop umgesetzt
- **mul_mv_q4_0_f32:**
  - ushort statt half → Konvertierungsmethode hinzugefügt. Die Konvertierungsmethode ist komplexer als in cpy.cl und eventuell nicht auf Hardware umzusetzen
  - Int-Division wude zu float gecastet
  - Aufteilen der Berechnung von sumy für die Multiplikation
  - N_DST = 8 zu N_DST = 3 geändert
- **mul:** Int-Division wude zu float gecastet
- **rms_norm:** ne00 konnte nicht als Schleifenbedingung verwendet werden -¿ Aufgeteilt in zwei Kernel mit festen Werten für ne00 (128,1024)
- **rope:** Nicht mit dem Brig-Compiler umsetzbar. Die Sinusfunktionen konnten nicht approximiert werden. Die Approximation von anderen Methoden(pow,log) ist zu komplex.
- **silu:** Die exp-Funktion musste approximiert werden. Die Implementierung ist wahrscheinlich zu komplex
- **softmax_4_f32**
  - Die ALiBi Berechnung wurde entfernt, da sie wegen der Rope Positionscodierung nicht benötigt wird
  - Exp musste durch komplexe Funktion approximiert werden
  - ne00 lässt sich in Schleifen durch den Brig-Compiler nicht verarbeiten -¿ Kernel kompiliert nicht

Ein Bestandteil zukünftiger Arbeiten am Tool ist die Implementierung komplexer mathematischer- und Trigonometrischer Funktionen. Diese werden für zahlreiche Berechnungen in LLMs, wie z.B bei Positionsencodings, Aktivierungsfunktionen und der Softmax-Operation, benötigt.

4.4 Umsetzung

Es war möglich, die Kernel, die sich mit dem Brig-Tool erfolgreich kompilieren ließen, in VHDL-Code zu überführen. Eine anschließende Synthese mit Vivendo und weitere notwendige Schritte, um den Code auf das FPGA zu synthetisieren, waren jedoch nicht erfolgreich. Diese Probleme lagen nicht am fehlerhaften OpenCL-Code, sondern unter anderem an Kompatibilitätsproblemen der OpenCL-Version des FPGA. Diese Probleme konnten im Zeitrahmen der Masterarbeit nicht behoben werden. Die Synthese des erstellten VHDL-Codes und die Ausführung der Inferenz auf dem FPGA ist somit Ziel zukünftiger Arbeit.
Es war jedoch möglich, die Inferenz des `llama.cpp` Projektes auf der CPU des FPGA auszuführen. Durch Zeitmessungen lässt sich erkennen, welche Auswirkungen ein langsamerer Prozessor auf die Ausführungszeit der Inferenz hat. Die zentrale CPU des verwendeten `Ultra96-V2` Boards ist eine Quad-Core ARM Cortex-A53 CPU mit einer Taktfrequenz von 1,5 GHz. Im Vergleich zu dem in den vorherigen Messungen verwendeten Intel-i7-Prozessor mit 2,8 GHz bietet der ARM-Prozessor weniger parallele Kerne, eine niedrigere Taktrate, deutlich kleinere Caches und kleinere SIMD-Vektoreinheiten. Der sehr parallele und speicheraufwändige Code der LLM-Inferenz hat auf dieser CPU Nachteile.
Gemessen wurde im Folgenden die Inferenz der Prefill Phase im Vergleich unterschiedlicher Hardware. Als Input wurde der gleiche 832 Token lange Prompt wie in den Messungen des Roofline-Vergleichs verwendet.
Abbildung 31 zeigt den Vergleich der Messungen auf der NVIDIA RTX 3080 GPU aus dem NHR@FAU Cluster, dem Intel i7 Prozessor und dem ARM-Prozessor des FPGA. Es fällt auf, dass die rechenintensive Prefill-Phase auf dem ARM-Prozessor eine deutlich längere Laufzeit hat. Eine zweite Messung auf dem

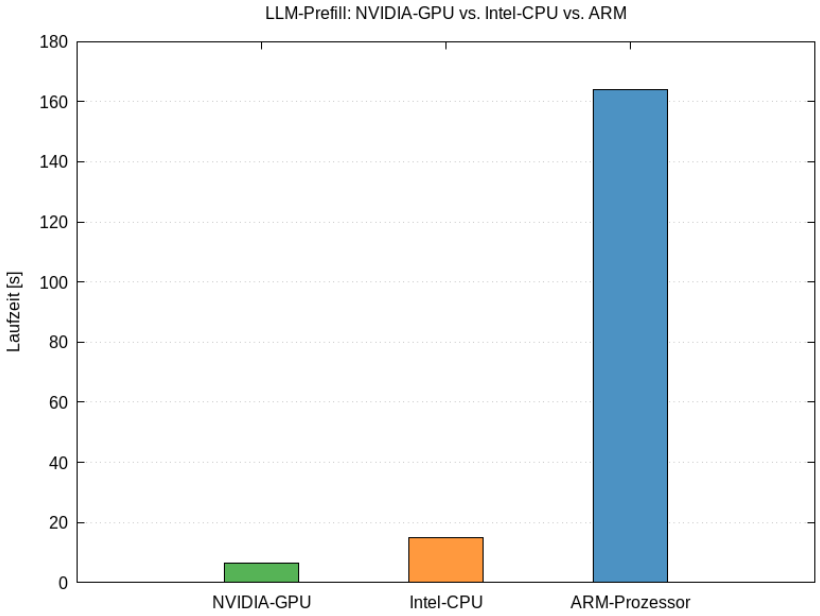


Abbildung 31: Messvergleich der Prefill-Laufzeiten auf den Intel- und ARM-Prozessoren

ARM-Prozessor (Abbildung 32) zeigt den Verlauf der Decode-Laufzeit. Mit einer steigenden Tokenzahl wächst der KV-Cache und somit der Rechenaufwand für den Attention-Mechanismus. Die Laufzeit wächst mit 869 zusätzlichen Tokens um rund 20 Prozent.

4.5 Ausblick

Um das Qwen3-0.6B-Modell mit dem Brig-Compiler auf einem FPGA lauffähig zu machen, waren mehrere Anpassungen an `llama.cpp` sowie an den verwendeten OpenCL-Kerneln erforderlich. Im Folgenden wird untersucht, welche weiteren Änderungen bzw. Kernelanpassungen notwendig wären, um zusätzliche Modelle zu unterstützen.
Betrachtet werden dabei Modellfamilien, die ähnlich wie Qwen3 kompakte Varianten mit weniger als drei Milliarden Parametern (<3B) anbieten und sich somit grundsätzlich für eine Implementierung auf einem FPGA eignen.
Zunächst erfolgt ein theoretischer Vergleich der Architekturunterschiede zwischen den betrachteten Mo-

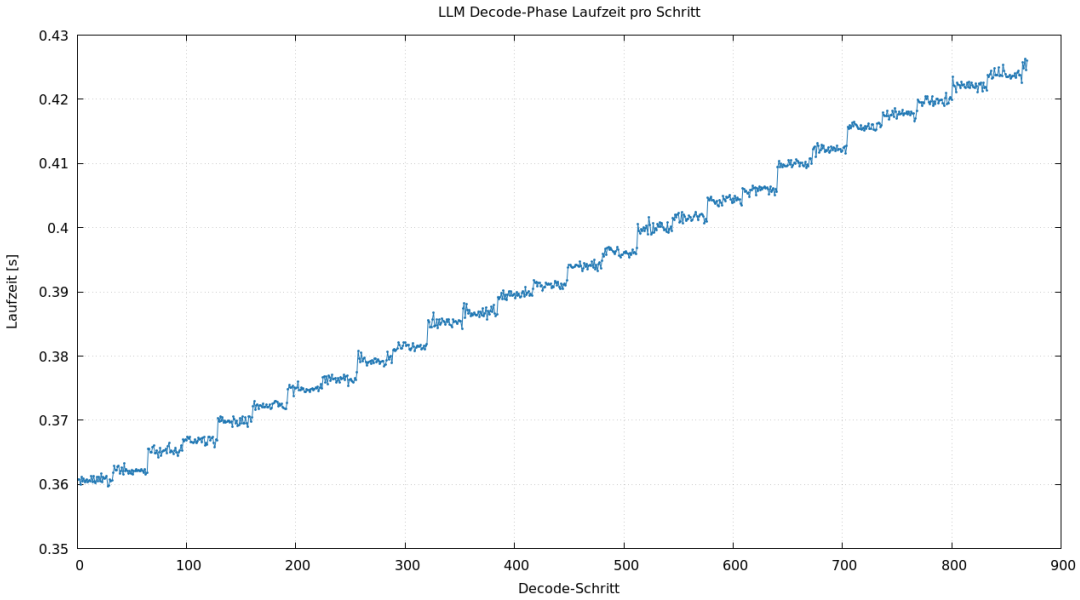


Abbildung 32: Messvergleich der Decode-Laufzeiten auf dem Arm-Prozessor

dellfamilien, um mögliche grundlegende Abweichungen zu identifizieren. Anschließend werden die Modelle mit `llama.cpp` für die Inferenz ausgeführt, und die dabei verwendeten OpenCL-Kernel werden analysiert. Dabei wird, analog zu Qwen3, von Datenformaten in FP16, FP32 sowie Q4-Quantisierung ausgegangen. Die betrachteten Modellfamilien sind:

- Gemma3 (270M) Google
- Phi3 Microsoft
- Deep Seek Coder (1.3B)
- LLaMA3 (1B)

4.5.1 Vergleichbare Modelle

Gemma3 Gemma3 wurde im Jahr 2025 von Google veröffentlicht [25]. Die Variante mit 270 Millionen Parametern ist besonders kompakt und primär auf Textverarbeitung optimiert. Neben der Textinferenz unterstützt Gemma3 auch Bildverarbeitungsaufgaben. Im technischen Bericht wird beispielsweise ein Anwendungsfall beschrieben, in dem ein Nutzer eine Restaurantrechnung abfotografiert und das Modell die Kosten pro Bestellung sowie den jeweiligen Anteil des Trinkgelds berechnet.
Gemma3 basiert, so wie Qwen3, auf einer reinen Decoder-Architektur. Die Modellarchitektur hat mehrere Gemeinsamkeiten mit Qwen3, darunter die Verwendung von Grouped Query Attention und der RMS-Norm. Als Aktivierungsfunktion wird jedoch GELU anstelle von SiLU eingesetzt.
Zwei zentrale Merkmale von Gemma3 sind Interleaved Attention und QK-Norm. Beide Mechanismen werden im Host-Code implementiert und erfordern daher keine eigenen Kernel. Beim Interleaving wird abwechselnd eine lokale Sliding-Window-Attention und eine globale Self-Attention ausgeführt. In Gemma3 folgen fünf lokale auf eine globale Attention-Schicht. Dieses Verfahren reduziert den Speicherbedarf für den KV-Cache, insbesondere bei langen Kontexten, und macht das Modell somit für ressourcenbeschränkte Umgebungen wie FPGAs relevant.
Die QK-Norm normalisiert die Query- und Key-Matrizen vor der Berechnung der Attention. Dabei wird, wie bei anderen Komponenten des Modells, ebenfalls die RMS-Norm verwendet.
Zusammenfassend sind für die Features von Geamm3 folgende Anpassungen notwendig:

- GELU: Neuer Kernel notwendig
- Interleaving Attention: Keine Kernelnänderung notwendig
- QK- Norm: Keine Kerneländerung notwendig

Phi 3 und Phi 4 Die Modellfamilien Phi-3 und Phi-4 von Microsoft basieren auf derselben Architektur. Die Hauptunterschiede in Phi-4 liegen in verbesserten Trainingsdaten und optimierten Datensätzen [9]. Laut dem technischen Bericht orientiert sich die Phi-Architektur stark an LLaMA-2 und verwendet somit ebenfalls eine reine Decoder-Architektur.
Phi nutzt denselben Tokenizer und dieselbe Vokabulargröße wie LLaMA-2, wodurch für LLaMA entwickelte Softwarepakete und Werkzeuge weitgehend kompatibel mit Phi-3-Mini sind.
Die technischen Berichte zu Phi-3 und Phi-4 enthalten nur begrenzte Informationen über die genaue Modellarchitektur. Aus der Implementierung des Berechnungsgraphen in `llama.cpp` lässt sich jedoch erkennen, dass die Architektur grundsätzlich ähnlich zu Qwen3 aufgebaut ist. Phi-3 verwendet, wie Qwen3, RoPE zur Positionskodierung, wobei die RoPE-Frequenzen schichtabhängig skaliert werden.Als Aktivierungsfunktion kommt SwiGLU zum Einsatz.
Das Modell Phi-4-Mini erweitert die reine Textinferenz zusätzlich um zwei weitere Encoder für Audio- und Bilddaten, wodurch es multimodale Eingaben verarbeiten kann.

Für die Verwendung von Phi-3 sind insbesondere folgende Anpassungen erforderlich:

- SwiGLU: Neuer Kernel notwendig

Mistral Mistral-Modelle werden in `llama.cpp` derzeit nicht unterstützt. In der Datei `llama-model.cpp` existiert zum aktuellen Zeitpunkt keine Methode zur Erstellung eines Berechnungsgraphen für ein Mistral-Modell.

4.5.2 Nicht angepasste Kernel

Um das Qwen3-Modell auf dem FPGA zu synthetisieren, musste LlamaCpp angepasst werden. Dabei wurden ausschließlich die für Qwen3 relevanten Kernel modifiziert. Nicht benötigte Kernel blieben unverändert. Es soll untersucht werden, welche Bedeutung die bisher nicht angepassten Kernel für die LLM-Inferenz haben und welcher Aufwand erforderlich wäre, um diese für das Brig-Tool anzupassen.

Die bisher unveränderten Kernel sind:

- **clamp:** Die `clamp`-Operation begrenzt Werte auf ein definiertes Minimum und Maximum. Im Kernel wird mittels einer if-Abfrage überprüft, ob die Eingabewerte innerhalb des zulässigen Wertebereichs liegen. Werte außerhalb dieses Bereichs werden auf den Minimal- bzw. Maximalwert gesetzt. Der Kernel ist einfach aufgebaut und müsste, falls erforderlich, nur minimal angepasst werden.

- **diag-mask:** Diese Maske wird auf die Attention-Werte im Attention-Mechanismus angewandt, um sicherzustellen, dass bei der Verarbeitung eines Tokens in der Prefill-Phase nicht auf zukünftige Tokens zugegriffen wird. Die entsprechenden Werte werden hierfür auf unendlich gesetzt. In LLaMA ist diese Funktionalität sowohl in einem eigenen Kernel als auch in der Softmax-Funktion integriert. Da der Softmax-Kernel auf die Attention-Scores zugreift, kann die Maskierung dort über einen zusätzlichen Maskeninput erfolgen. Der Vorteil dieser Integration liegt in der Reduktion von Speicherzugriffen auf große Attention-Matrizen. Ein separater `diag-mask inf`-Kernel ist daher nicht erforderlich.

- **GELU:** Die GELU-Aktivierungsfunktion:

$$\mathrm{GELU}(x) = 0.5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

  verwendet die `tanh`-Funktion. Für die Nutzung von GELU durch den Brig-Compiler müsste diese Implementiert werden.

- **gemv:** Der `gemv_noshuffle`-Kernel implementiert eine Matrix-Vektor-Multiplikation. Die Kernel `gemv_noshuffle` und `gemv_noshuffle_general` werden in der `load_kernel`-Funktion zu Q4-Multiplikationskernen kompiliert: `CL_mul_mat_vec_q4_0_f32_1d_4x_flat_general`. Diese Multiplikationskernel werden im Host-Code ausschließlich für Adreno-Hardware verwendet und sind somit für andere Hardware nicht erforderlich.

- **Mul Mat:** Der Multiplikationskernel `kernel_mul_mat_Ab_Bi_8x4` wird ausschließlich für Matrixmultiplikationen auf Adreno-Hardware verwendet und muss daher nicht angepasst werden. Weitere bislang ungenutzte Matrixmultiplikationskernel dienen primär hardwareabhängigen Optimierungen und sind für die Grundfunktionalität nicht erforderlich.

- **scale:** Dieser Kernel multipliziert den Eingabevektor mit einem Skalierungsfaktor. Er führt keine komplexe Berechnung durch, und kann einfach für den Brig-Compiler angepasst werden.
- **transpose:** Die Transpositionsoperation tritt zwar in den Operatorbäumen der verwendeten LLMs auf, wird jedoch in der tatsächlichen Berechnung nicht ausgeführt. In der Methode `compute_forward` wird für Transpose-Operationen ein `nop` (No Operation) eingefügt. Die vorhandenen Transpose-Kernel werden für große gebatchte Matrizen verwendet und im Host-Code ausschließlich für Adreno-Hardware geladen. Für andere Hardwaretypen werden diese Kernel nicht benötigt, weshalb keine Anpassung an den Brig-Compiler erforderlich ist. Es ist anzunehmen, dass die Gewichte bereits in transponierter Form vorliegen.

4.5.3 Datentypen

GGML stellt insbesondere für die Matrixmultiplikation und die Softmax-Berechnung eine Vielzahl von Kernelimplementierungen bereit, die in Abhängigkeit vom Datenformat der Tensoren ausgewählt werden. Für das Qwen3-Modell wurden die ursprünglich in Q6 quantisierten Tensoren in das Q4-Format konvertiert, um die Anzahl der benötigten Kernel zu reduzieren. Die OpenCL-Host-Datei von GGML wählt für Qwen3 zudem Kernel aus, die mit im SOA-Format (Sequence of Arrays) gespeicherten Daten arbeiten. Es soll untersucht werden, ob für andere Modelle ebenfalls ausschließlich Kernel für Q4- und FP32/FP16-Datentypen verwendet werden können oder ob diese Modelle von weiteren Datentypen abhängen. Dazu wird geprüft, welche Quantisierungen in den jeweiligen Modellen eingesetzt werden und ob gegebenenfalls Anpassungen erforderlich sind.

Gemma3 Gemma3 verwendet für die Gewichtsmatrizen das in vielen LLMs verbreitete BF16-Format. BF16 nutzt, so wie FP16, 16 Bit, weist jedoch eine verkürzte Mantisse und einen vergrößerten Exponenten auf. Dadurch kann ein größerer Wertebereich dargestellt werden, allerdings mit reduzierter numerischer Genauigkeit. Unterstützung für BF16 wurde in GGML hinzugefügt[45] . In den OpenCL-Kernelimplementierungen existiert jedoch keine Variante, die direkt mit BF16-Werten arbeitet. Vor dem Aufbau des Operatorbaums werden die Gewichte daher nach FP16 konvertiert. Um dies zu prüfen wurde das Gemma3-Modell mit 270 M Parametern[46] heruntergeladen, der Operatorbaum erstellt und ausgegeben.
Die kleinste auf der Google-Huggingface-Seite verfügbare Gemma-Version umfasst 1 B Parameter und verwendet Q4_0-quantisierte Gewichte[47] .

Phi 4 Das Modell Phi4-mini-instruct mit 4 B Parametern[48] nutzt Gewichte im BF16-Format.

4.5.4 Fazit

In diesem Kapitel wurde ein Ausblick auf weitere erforderliche Anpassungen gegeben, um neben Qwen3 0.6B auch Sprachmodelle anderer Modellfamilien zu unterstützen. Hierzu wurde eine Architekturübersicht über vergleichbar kleine und populäre Modelle, nämlich Gemma von Google und Phi von Microsoft, erstellt. Dabei wurden sowohl die technischen Berichte als auch die in `llama.cpp` generierten Operatorbäume dieser Modelle analysiert.
Aufgrund unterschiedlicher Berechnungskernel für verschiedene Datenformate wurden auch die verwendeten Daten- bzw. Quantisierungsformate der Modellgewichte untersucht. Es zeigte sich, dass der grundlegende Architekturaufbau dieser Modelle dem von Qwen3 sehr ähnlich ist. Beide verwenden reine Decoder-Architekturen mit Attention- und FFN-Blöcken. Aus den Operatorbäumen lässt sich erkennen, dass sowohl der Aufbau als auch die verwendeten Operationen weitgehend mit denen von Qwen3 übereinstimmen. Die wesentlichen Unterschiede und notwendigen Anpassungen betreffen zum einen die verwendeten Aktivierungsfunktionen. Im Gegensatz zu Qwen3 setzen vergleichbare LLMs häufig das Datenformat BF16 ein. Es konnte jedoch festgestellt werden, dass `llama.cpp` dieses Datenformat unterstützt und die BF16-Gewichte vor der Erstellung des Operatorbaums sowie vor der Inferenzausführung in FP16 konvertiert. Dadurch können die vorhandenen FP16-Kernel weiterhin genutzt werden.
Eine abschließende Analyse der bisher nicht angepassten OpenCL-Kernel zeigte, dass diese entweder nicht erforderlich sind, da sie lediglich für spezifische Hardware wie Adreno-Geräte vorgesehen sind, oder dass

[45] `https://github.com/ggml-org/llama.cpp/pull/6412`
[46] `https://huggingface.co/google/gemma-3-270m`
[47] `https://huggingface.co/google/gemma-3-1b-it-qat-q4_0-gguf`
[48] `https://huggingface.co/microsoft/Phi-4-mini-instruct`

deren Anpassung an den BRIG-Compiler mit geringem Aufwand möglich ist. Lediglich die Anpassung weiterer Aktivierungsfunktionen, wie beispielsweise GELU, erfordert zukünftig zusätzliche mathematische Funktionen, etwa die Tangensfunktion.

5 Abschluss

In dieser Masterarbeit wurde die Ausführung großer Sprachmodelle auf ressourcenbeschränkter Hardware untersucht. Der Schwerpunkt lag auf der Analyse und Umsetzung der Inferenz eines kleinen, öffentlich verfügbaren Sprachmodells auf einem System mit begrenzter Speicher- und Rechenkapazität. Die Arbeit teilt sich in drei Hauptteile. Zunächst wurden Aufbau und grundlegende Mechanismen von LLMs beschrieben. Anschließend wurde der Rechen- und Speicherbedarf des ausgewählten Modells theoretisch analysiert und mithilfe eines Roofline-Modells visualisiert. Abschließend wurden erste Schritte zur Implementierung und Ausführung des Modells auf einer FPGA-Plattform vorgestellt und ein Ausblick auf zukünftige mögliche Erweiterungen gegeben.
Im ersten Teil der Arbeit wurden die wichtigsten Architekturmerkmale von LLMs beschrieben, mit Fokus auf den für die Inferenz relevanten Komponenten. Ergänzend wurde eine Literaturrecherche zu Optimierungsmethoden durchgeführt, die zeigt, mit welchen Techniken sich Speicherbedarf reduzieren und Rechenzeit verkürzen lässt. Techniken zum Trainieren von LLMs werden nicht beschrieben, da die Arbeit von bereits vortrainierten Modellen ausgeht.
Neben der generellen LLM-Architektur wurde das in dieser Arbeit verwendete Inferenz-Framework `llama.cpp` beschrieben. Dazu wurde das Projekt in seine Hauptkomponenten zerlegt und der Kontrollfluss visualisiert. Für die Analyse des Rechenaufwands der einzelnen Operationen der LLM-Inferenz wurden die in `llama.cpp` verwendete Bibliothek GGML und deren OpenCL-Implementierung untersucht. Die Beschreibung zeigt, welche Teile von `llama.cpp` für Messungen und Anpassungen der LLM-Inferenz relevant sind, und senkt damit die durch die begrenzte Dokumentation bestehende Einstiegshürde in das Projekt.
Ein zentraler Teil der Arbeit befasst sich mit der Analyse des für die LLM-Inferenz erforderlichen Rechenaufwands. Für jeden eingesetzten OpenCL-Kernel wurde die auf die Zielhardware optimierte Berechnung beschrieben. Der theoretisch erwartete Rechenaufwand in FLOPs wird anschließend mithilfe des Messtools LIKWID validiert. Für einen vollständigen Layer wurde zudem ein Python-Programm erstellt, mit dem sich die FLOP-Zahl eines Qwen3-Layers in Abhängigkeit von der Eingabegröße berechnen lässt.
Mithilfe einer Roofline-Analyse wurde untersucht, welche Teile der Architektur und in welchen Phasen die Inferenz besonders rechenintensiv ist und sich für eine Auslagerung eignet. Betrachtet wurden zwei Szenarien. Zum einen wurde der Unterschied zwischen den beiden Inferenzphasen eines autoregressiven Decode-Modells analysiert. Hier zeigte sich, dass die Prefill-Phase bei langen Eingabesequenzen rechendominant ist und von einer Auslagerung auf schnellere Hardware profitieren kann. Zum anderen wurde verglichen, welcher Teil eines LLM-Layers, der Attention-Mechanismus oder das FFN, den höheren Rechenaufwand verursacht. Bestehende Arbeiten verglichen diese Anteile bereits für größere Modelle. Die Auswertung in dieser Arbeit zeigte, dass bei einem kompakten Modell wie Qwen3 0.6B das FFN-Layer nicht dominant ist, sondern sich die Rechenlast vergleichsweise gleichmäßig auf FFN- und Attention-Layer verteilte und der FFN-Anteil mit zunehmender Kontextlänge weiter abnahm.
Diese Erkenntnisse können künftig genutzt werden, um rechenintensive Teile der Inferenz gezielt zu identifizieren und auf geeignete Hardware auszulagern.
Neben der Implementierung der Inferenz beschreibt diese Arbeit auch verschiedene Möglichkeiten und Herausforderungen bei der Messung von LLMs auf ressourcenbeschränkten Systemen. Für die Messungen kamen mehrere Mess-Tool sowohl auf einem Laptop mit Intel-CPU als auch auf dem NHR@FAU-Rechencluster zum Einsatz. Dabei traten an mehreren Stellen Einschränkungen auf. Auf dem Intel-Laptop konnten beispielsweise mit LIKWID keine Messungen der Speicherverwendung durchgeführt werden, da die dafür erforderliche Messgruppe auf dem System nicht verfügbar war. Als Alternative wurde daher das Intel-Tool VTune verwendet. Zudem waren FLOP-Messungen nur auf der CPU, nicht jedoch auf der GPU möglich.
Die Messung der Inferenz kann ebenfalls in virtuellen Maschinen durchgeführt werden. Dafür wurden die dafür erforderlichen Voraussetzungen und Zugriffsrechte beschrieben. Ein Rechnencluster auf dem die Inferenz erfolgreich ausgeführt und gemessen werden konnte ist das NHR (National High Performance Computing Center) der FAU. Aufgrund der für die Messungen erforderlichen exklusiven Zugriffsrechte gestaltete sich die Durchführung dort jedoch teilweise als sehr zeitaufwändig. Diese Probleme zeigen, dass zwar kleine LLM-Modelle grundsätzlich auf ressourcenbeschränkten Systemen ausgeführt werden können, deren präzise Messung jedoch moderne Hardware erfordert.

Ein weiterer Schwerpunkt der Arbeit lag auf der Anpassung des Projekts `llama.cpp` und insbesondere der in GGML verwendeten OpenCL-Kernel, sodass diese mit dem BRIG-Tool kompiliert und anschließend auf einem FPGA eingesetzt werden konnten. Hierfür wurde der OpenCL-Code an mehreren Stellen überarbeitet und an die Anforderungen des BRIG-Compilers angepasst. Für jeden betrachteten Kernel wurde dokumentiert, welche Änderungen erforderlich waren. Durch diese Anpassungen ließ sich korrekter VHDL-Code erzeugen, der sich allerdings aufgrund fehlender Kompatibilität des FPGAs und begrenzter Funktionalität des BRIG-Compilers noch nicht erfolgreich synthetisieren ließ. Die Inferenz konnte daher im Rahmen dieser Arbeit nur auf der CPU des FPGA-Systems ausgeführt und gemessen werden. Sobald das FPGA die benötigte OpenCL-Version unterstützt und der BRIG-Compiler erweitert wird, können die angepassten Kernel in zukünftigen Arbeiten auf dem FPGA genutzt werden.
Ein Ausblick zeigte, dass für den Einsatz weiterer LLM-Familien nur wenige zusätzlich angepasste Kernel erforderlich sind. Dazu wurden die Architekturen der Modelle Gemma3 und Phi-4 mit der von Qwen3 verglichen. Der Aufwand der dafür nötigen Änderungen ist gering und betrifft hauptsächlich unterschiedliche Aktivierungsfunktionen.
Die vorgenommenen Anpassungen an den OpenCL-Kernen bilden eine Grundlage für zukünftige Ausführungen und Messungen der LLM-Inferenz auf einem FPGA-Systemen. Die Projektbeschreibung soll zudem die zukünftige Weiterarbeit erleichtern.

List of Abbreviations

ALiBi Attention with Linear Biases.

FPGA Field Programmable Gate Arrays.

GQA Group-Query-Attention.

LLM Large Language Model.

MHA Multi-Head-Attention.

MQA Multi-Query-Attention.

RNN Recurrent neural network.

RoPE Rotary Positional Encoding.

Literatur

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[3] Adam Casson. Transformer flops, 2023.

[4] Peizhuang Cong, Qizhi Chen, Haochen Zhao, and Tong Yang. Baton: Enhancing batch-wise inference efficiency for large language models via dynamic re-batching, 2024.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[6] Harm de Vries. In the long (context) run, 2023.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[8] Razvan-Gabriel Dumitru, Vikas Yadav, Rishabh Maheshwary, Paul-Ioan Clotan, Sathwik Tejaswi Madhusudhan, and Mihai Surdeanu. Layer-wise quantization: A pragmatic and effective method for quantizing llms beyond integer bit-levels, 2024.

[9] Marah Abdin et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[10] Mukhammed Garifulla, Juncheol Shin, Chanho Kim, Won Kim, Hye Kim, Jaeil Kim, and Seokin Hong. A case study of quantizing convolutional neural networks for fast disease diagnosis on portable medical devices. *Sensors*, 22:219, 12 2021.

[11] Xuan-Son Nguyen Georgi Gerganov. Introduction to ggml, 2024.

[12] Alicia Golden, Samuel Hsia, Fei Sun, Bilge Acun, Basil Hosmer, Yejin Lee, Zachary DeVito, Jeff Johnson, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Is flash attention stable?, 2024.

[13] Ozgur Guldogan, Jackson Kunde, Kangwook Lee, and Ramtin Pedarsani. Multi-bin batching for increasing llm inference throughput, 2024.

[14] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, page 1–15. ACM, June 2023.

[15] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[16] imrimallis. Understanding how llm inference works with llama.cpp, 2023.

[17] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models, 2024.

[18] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[19] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024.

[20] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021.

[21] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation, 2022.

[22] Qwen Team. Qwen3 technical report, 2025.

[23] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[24] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[25] Gemma Team. Gemma 3 technical report, 2025.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[27] Cheng Zhang, Xingyu Zhu, Longhao Chen, Tingjie Yang, Evens Pan, Guosheng Yu, Yang Zhao, Xiguang Wu, Bo Li, Wei Mao, and Genquan Han. Enhancing llm inference performance on arm cpus through software and hardware co-optimization strategies. *Integrated Circuits and Systems*, PP:1–9, 01 2025.