課題研究報告書

LLM エッジ推論の FPGA 実装及び最適化に関する研究

HE Jiaxiang

主指導教員 田中 清史 教授

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和 07 年 9 月

Abstract

Recent advancements in large language models (LLMs) such as ChatGPT have demonstrated remarkable capabilities in natural language understanding and generation. However, these models demand substantial computational resources and energy, limiting their deployment to data centers equipped with high-end GPUs. Deploying LLMs on edge devices introduces additional challenges, including limited memory, constrained computational power, and strict energy budgets.

To address these issues, field-programmable gate arrays (FPGAs) have emerged as promising platforms due to their parallelism, low power consumption, and reconfigurability. While existing studies such as DFX and FlightLLM have shown significant improvements in inference performance using multi-FPGA systems for data centers. Building upon insights from existing studies, this work explores a practical approach to adapting large language model inference to the specific constraints of edge computing environments—namely, limited logic cell resources and memory bandwidth. It aims to complement existing high-performance-oriented research and extend the applicability of FPGA-based LLM inference to edge scenarios.

To effectively implement large language models (LLMs) on FPGAs, it is essential to first establish a solid computational foundation. This work adopts a top-down approach, it begins with an overview of LLM architecture, focusing particularly on the Transformer and LLaMA models, followed by a discussion of key hardware concepts, including FPGAs, SoC FPGAs, OpenCL, and High-Level Synthesis (HLS).

The related work section reviews recent advancements in accelerating Large Language Model (LLM) inference using FPGAs, specifically focusing on two prominent works: DFX and FlightLLM. Both DFX and FlightLLM conclusively demonstrate the viability and significant advantages of FPGA acceleration for LLM inference, particularly within data center environments where high-end FPGAs with abundant logic and HBM resources are available. However, their specific architectural choices and resource requirements make them less suitable for deployment on resource-constrained edge devices. Building upon the valuable insights gleaned from these pioneering works, this study aims to tailor the approach for edge-specific constraints.

The primary purpose is to create a fully standalone edge AI device for LLM inference. This capability is crucial for various intended use cases where real-time, on-device processing is paramount. To meet these demands, the core performance specifications for the device are: Response Time (less than 1 second), Context Length (200 characters or more) and Inference Speed (2 words per second).

The tinyllamas model is selected for its lightweight architecture, which aligns

well with our requirements in terms of model size and input sequence length, as well as its high implementability on resource-constrained edge devices. The AMD Xilinx Kria™ KV260 development board is chosen as the hardware platform. The KV260's SoC FPGA integrates an Arm CPU-based Processing System (PS) and Programmable Logic (PL). To manage diverse tasks and ensure flexible development, a Linux OS (PetaLinux) is employed.

Instead of a full model implementation on the FPGA, the approach involves subdividing computational processes into fundamental, reusable tasks. These basic operations, such as addition, scalar multiplication, matrix multiplication, normalization, Rotary Positional Embeddings (ROPE), and Softmax function, are implemented as independent accelerator kernels. Specifically, these are: add_kernel, mul_kernel, matmul_kernel, rmsnorm_kernel, rope_kernel, and softmax_kernel.

The system operates in three distinct phases after the execution of the host application. Initialization Phase involves loading the tinyllamas model weights, configuring OpenCL settings, loading and initializing the accelerator kernels, and allocating necessary memory resources. In the Text Generation Loop phase, the model predicts and outputs the next token based on the input. It repeatedly calls the DecoderLayer function, converts tokens to characters, prints them, and checks if the generated string has reached the specified length. Upon completion, Termination Process phase outputs log information (e.g., processing time), releases all used memory, and gracefully terminates the program.

The evaluation focuses on two key aspects: hardware resource usage and character generation time. The performance of individual accelerator kernels is meticulously analyzed to identify areas for improvement, after which targeted optimizations are applied. The system is then re-tested post-optimization to validate the effectiveness of these methods and discuss the results.

The results of the initial version of implementation revealed that the inference latency is approximately 1.5 times the target system specification. For instance, generating 32 output tokens took 24209.4 ms, translating to an average latency of approximately 756 ms per token, well above the target 500ms per word. This clearly indicated that significant performance optimization for speed-up is essential. Analysis of individual accelerator kernel usage times highlighted a critical bottleneck: the matmul_kernel (matrix multiplication) alone accounted for over 80% of the total execution time. Consequently, optimization efforts should be primarily focused on improving the performance of the matmul_kernel.

The matmul_kernel is decomposed into four internal functions: stream_input_vector and stream_input_matrix for data input, compute_matrix_vector_multiply for computation, and write_result_to_memory for data output. In the initial implementation, the DATAFLOW pragma is applied to these functions to enable function-

level parallelization. Additionally, loop unrolling (UNROLL) is attempted on the inner loop of the compute_matrix_vector_multiply function's nested loop (matrix_multiply_execute). However, a critical limitation is identified: due to the matmul_kernel's design for versatility across different vector and matrix sizes, the HLS tool could not determine the exact number of loop iterations at compile time. This often resulted in the UNROLL pragma being ignored, hindering the desired performance gains.

Two optimization approaches are explored for the matmul_kernel. The first approach, static dedicated kernelization, aimed to accelerate performance through loop unrolling. However, this proved impractical as its on-chip memory (BRAM) usage exceeded the FPGA's constraints.The second approach, pipelining, involved applying #pragma HLS PIPELINE to the internal loops of each function. This significantly improved the throughput of data transfer and computation. Furthermore, the QKV calculation within the DecoderLayer function is parallelized, and the matmul_kernel is divided into separate Q, K, and V operations, enabling parallel execution and enhancing the overall efficiency of the accelerator.

Test results after optimization showed a maximum of 76.5% improvement in token generation speed and a 43.3% reduction in average latency. These achievements meet the initial system performance goals and demonstrated the effectiveness of accelerating LLM inference processing on an embedded FPGA.

This work establishes a foundation for deploying LLMs in resource-constrained environments and identifies key optimization strategies for future research. The results indicate strong potential for FPGA-based LLM inference in applications requiring local processing, power efficiency, and cost-effectiveness, while highlighting opportunities for further improvements through quantization, mixed-precision arithmetic, and advanced memory management techniques.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景

近年、ChatGPT のような大型言語モデル（Large Language Model、以下 LLM という）は、テキスト生成や自然言語生成において顕著な性能を達成している。しかしながら、その強力な性能の背後に膨大な計算資源使用量と極めて高額な運用コストの現状がある。エッジ側のローカル LLM 応用は、ハードウェアリソースの制約、消費電力、モデルサイズとユーザー体験のトレードオフなどの課題に直面している。

一方、FPGA(Field Programmable Gate Array) は、高速な並列処理と消費電力の面で優れており、内部の回路構造を何度も再構成できるため、AI モデルのエッジコンピューティングに適している。

LLM を FPGA に実装すること [1][2][3] によって、現行の GPU 実装より高いスループット、エネルギー効率向上が可能になった。既存研究はデータセンター向けの FPGA を用いて、パラメータ数の多い LLM の実装例を示した。ただし、データセンター向けハイエンドの FPGA と比べた結果、エッジデバイスはハードウェアリソース、メモリ容量、消費電力などの制限があり、エッジ向けの LLM の FPGA 実装は性能確保と品質保証という点で未解決である。

1.2　研究目的

エッジデバイスへの LLM 実装は、主にリソース制限、エネルギー効率、およびレイテンシ削減という課題を伴う。本研究では、これら三つの課題に対し、LLM の FPGA 実装、最適化、および性能評価の全工程を探究する。

エッジデバイス上では利用可能なメモリと計算能力に制約があるため、LLM のリソース利用効率を最大化するための工夫が不可欠である。限られたリソースを有効活用しつつ、モデルの性能と生成内容の品質を維持する方法を模索する。また、エッジデバイスはバッテリー駆動である場合が多いため、消費電力の削減は極めて重要である。本研究では、ハードウェア全体のエネルギー消費削減を目指し、エネルギー効率の向上を図る。レイテンシの改善に関しては、エッジデバイス上での LLM の応答時間を可能な限り削減し、レスポンス性能を向上させること

を目標とする。これにより、ユーザーエクスペリエンスを改善し、リアルタイムアプリケーションにおけるLLMの利用を可能にする。

1.3 報告書構成

本報告書は、LLMのFPGA実装と最適化に関する研究成果を以下の構成で記述する。

- 第2章では、本研究に関連するLLMの仕組みおよびFPGAハードウェアの前提知識について述べ、その後、高位合成（HLS）による開発手法を説明する。
- 第3章では、FPGAを用いたLLM用ハードウェアアクセラレータに関する先行研究について述べる。
- 第4章では、本研究の対象システムの仕様と目標を定義し、その実装手法について述べる。
- 第5章では、システムの実験方法、最適化前後の実験結果、およびそれらの考察について述べる。
- 第6章では、本報告書のまとめと今後の課題について述べる。

第2章　予備知識

本章では、本研究の基礎となる予備知識をまとめる。まず、2.1 節から 2.4 節では Transformer アーキテクチャ、大規模言語モデル（LLM）の概要、LLaMA の内部構造、およびテキスト生成の仕組みについて説明する。次に、2.5 節から 2.9 節では、FPGA と SoC FPGA の定義、設計開発フロー、OpenCL、および高位合成（HLS）の詳細について述べる。

2.1　Transformer

Transformer は、2017 年に Vaswani らによって発表された論文「Attention Is All You Need」[4] において提案された。それ以前の主流であった RNN（Recurrent Neural Networks）や LSTM（Long Short-Term Memory）は、逐次的なデータ処理を行うため、計算の並列化が困難であり、長期的な文脈情報の保持が難しいという課題を抱えていた。Transformer は「注意機構（Attention）」を活用することで、全ての単語（トークン）を並列に処理することを可能にし、より長い文脈を効果的に学習できる構造を有している。

Transformer は図 2.1 に示すように、エンコーダ・デコーダ (Encoder-Decoder) 構造を持つニューラルネットワークであり、以下の主要コンポーネントで構成される。

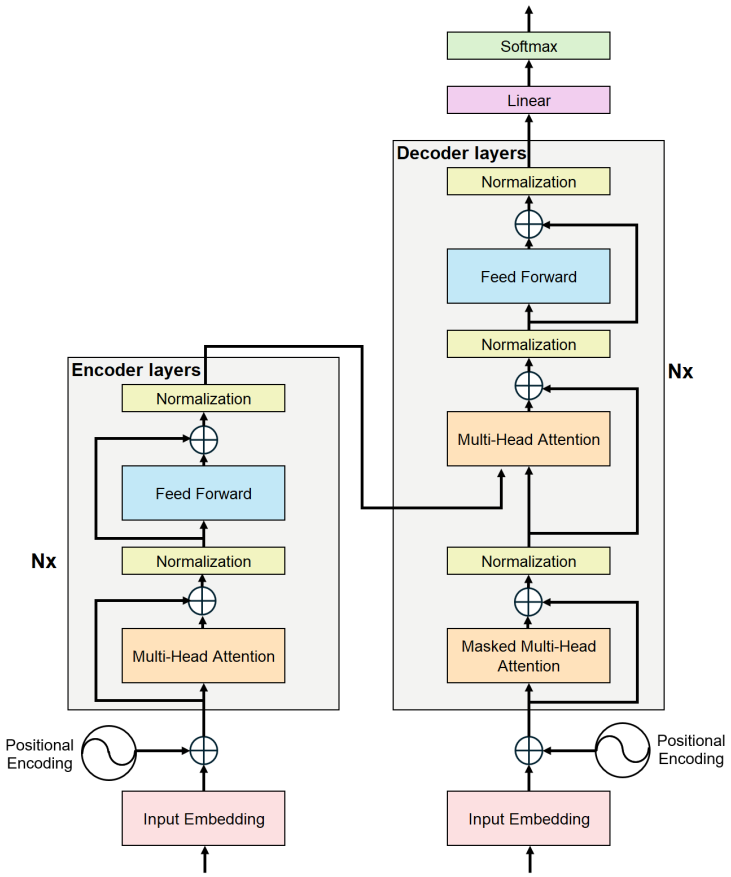

図 2.1: Transformer アーキテクチャ構造図 [4]

入力埋め込み（Input Embedding） テキスト中の単語（または文字）をベクトル空間に変換する機能であり、単語（または文字）間の意味的関係を反映した密な表現を提供する。

位置エンコーディング（Positional Encoding） Transformer は系列の順序情報を直接扱わないため、各単語の位置情報を数値的に埋め込む。具体的には、正弦関数や余弦関数を用いた周期的な関数によって位置エンコーディングが生成され、入力埋め込みに加算される。

注意機構（Attention） 各単語の入力系列内の他の全ての単語との関連性を学習することで、文脈情報を獲得するメカニズムである。これにより、モデルは文中のどの部分に「注意」を払うべきかを動的に判断できる。

フィードフォワードネットワーク（FFN） 各トークンの表現を非線形に変換し、強化するための全結合型ニューラルネットワークである。各トークンに対して独立に適用される。

正規化と残差接続（Layer Normalization & Residual Connections） モデルの学習を安定化させ、勾配消失問題を軽減するために、各サブレイヤーの出力にレイヤー正規化と残差接続が適用される。これにより、深いネットワークでの情報伝播が改善される。

2.2 Large Language Model（LLM）

Large Language Model（大規模言語モデル、略称：LLM) は、ディープラーニングを基盤とし、人間のようなテキスト生成能力を持つ人工知能モデルである。これらのモデルは、与えられたプロンプト（初期テキスト）に基づいて、その文脈を継続する形で新たなテキストを生成する。LLM は、テキストの理解と生成において人間レベルの能力を発揮することを目指して設計されている。具体的には、文脈からの推論能力、文脈に即した首尾一貫した応答生成能力、多言語翻訳能力、文章要約能力、質問応答能力（一般的な会話や FAQ への対応)、さらには創造的なライティングやコード生成タスクの支援能力などを備えている [5]。現在広く利用されている高性能な LLM の大部分は、BERT、GPT ファミリ、PaLM、LLaMA ファミリに代表されるように、Transformer アーキテクチャを採用しており、その中でも特にデコーダのみ（Decoder-Only）の構造が主流となっている。

LLM は、そのライセンス形態により、クローズドソース（プロプライエタリ）モデルとオープンソースモデルの二つに大別される。

クローズドソース LLM は、モデルの重みや学習データが非公開であり、通常は OpenAI の ChatGPT や Anthropic の Claude のように、ウェブサイトや API を通じてのみ利用が提供される。これらは企業によって開発され、高い性能を持つ一方で、利用には特定の制約が課される場合が多い。

オープンソース LLM は、モデルのアーキテクチャ、重み（パラメータ)、学習データの一部、および推論・学習コードが公開されているモデルを指す。主に学術機関や企業が開発し、研究利用や商用利用を目的として提供されている。

近年では、Meta の LLaMA や Google の Gemma など、商用利用が可能な強力なオープンソースモデルが登場しており、特に LLaMA は多くのカスタマイズされた LLM の基盤モデルとして広く利用されている。代表的なオープンソース LLM を表 2.1 に示す。

表 2.1: 代表的なオープンソース LLM 一覧表

ベンダー	モデル名	パラメータ数※
Meta	LLaMA2	7B/13B/70B
	LLaMA3.1	8B/70B/405B
Mistral AI	Mistral	7B/8X7B(MoE)/8X22B(MoE)
Google	Gemma	2B/7B
	Gemma2	2B/9B/27B
OpenAI	GPT-2	117M/355M/774M/1.5B
Microsoft	Phi-3	4B/7B/14B
	Phi-4	15B

※ M は million（百万）、B は billion（十億）を示す。以下、M、B とする。

2.3 LLaMA ファミリ

2017 年に提案されたオリジナルの Transformer アーキテクチャに基づき、各企業は独自の LLM および関連サービスを相次いで発表している。しかしながら、クローズドソースの LLM はその学習手法や内部構造が非公開であるため、本節では、重みおよびモデル構造が公開されているオープンソース LLM である「LLaMA ファミリ」を対象として詳細な説明を行う。修正を適用した後の構造を図 2.2 に示す。

LLaMA ファミリは、Transformer の原始構造に対して以下の変更を加えている。

層正規化（Layer Normalization）の位置を、各層の入力直前に移動。フィードフォワードネットワーク（FFN）における活性化関数を ReLU から SwiGLU に変更。位置エンコーディング（Positional Encoding）を廃止し、代わりにロータリーポジション埋め込み（Rotary Positional Embeddings: RoPE）を導入。RoPE は各デコーダ層の内部に組み込まれる。

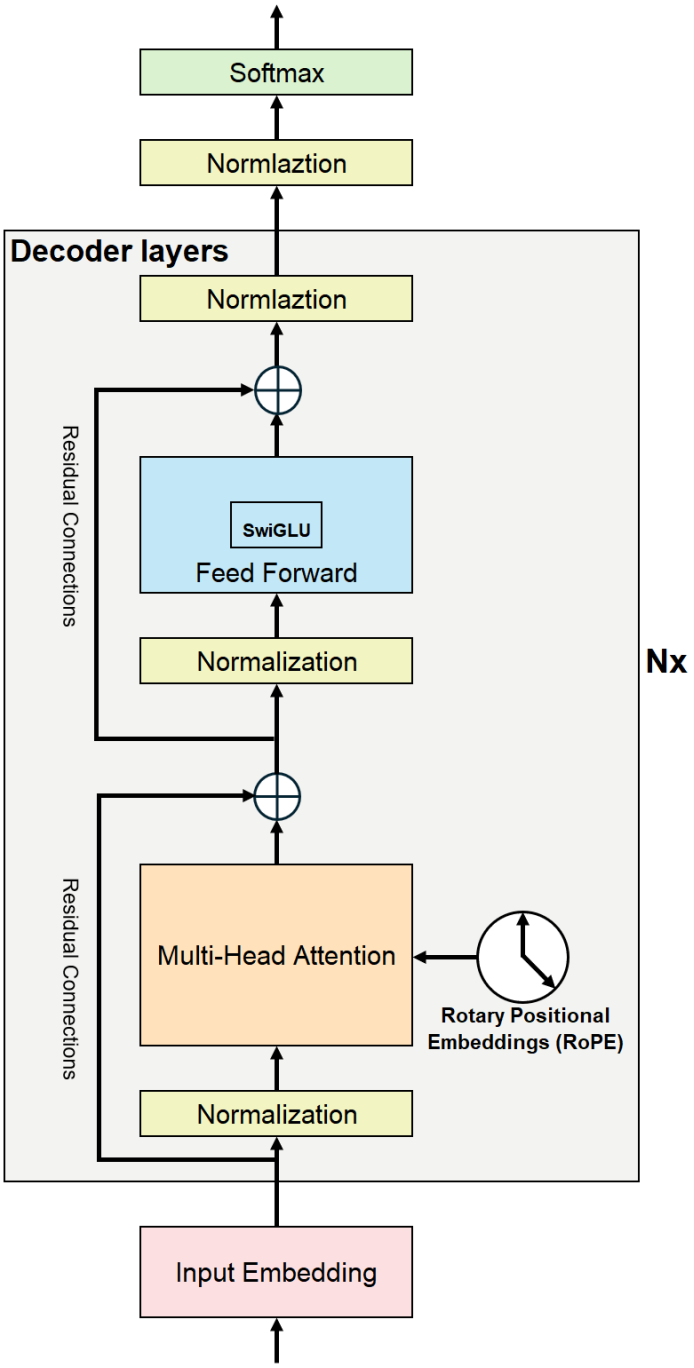

図 2.2: LLaMA ファミリの構造図 (Decoder-Only)

2.4 LLaMA の仕組み

LLM を FPGA に実装し、ハードウェアアクセラレータを設計するにあたっては、まず LLM の原理および計算処理の本質的理解が不可欠である。以下では、LLaMA ファミリの LLM を例にとり、モデルの動作原理と処理フローを詳述する。

具体的には、重みファイルの内部構造を解析し、モデルが各ステップにおいてどのように重みコンポーネントを用いて計算を実行し、最終的にテキストを生成するに至るかを段階的に説明する。

2.4.1 重みファイル

説明の便宜上、LLaMA ファミリを基盤とする LLM を LLaMATemp と呼称する。LLaMATemp のモデルパラメータは以下の通りである。

・モデル次元（Dim_model）：d
・デコーダレイヤーの数（N_layer）：n
・マルチヘッドアテンションのヘッド数（Att_heads）：ah
・語彙数（Vocab_size）：v
・最大コンテキスト：ctx
・精度：$FP32$

モデル重みファイルの構造を図 2.3 に示す。入力埋め込みテーブル（Input Embedding）、デコーダレイヤー（Decoder Layer）、および三角関数 LUT（Look Up Table）から構成されている。各デコーダレイヤーは共通の構造を有し、アテンションヘッド（Attention Head）、フィードフォワードネットワーク（FFN）、および RMS 正規化レイヤー（Input RMSNorm）で構成されている。図 2.3 の各矩形ブロックは各コンポーネントの重みを表しており、それぞれの行列の行数および列数はモデルのサイズに依存する。例として、Input Embedding の重み行列は、語彙数 v を行数、モデル次元 d を列数とする $v \times d$ の行列である。

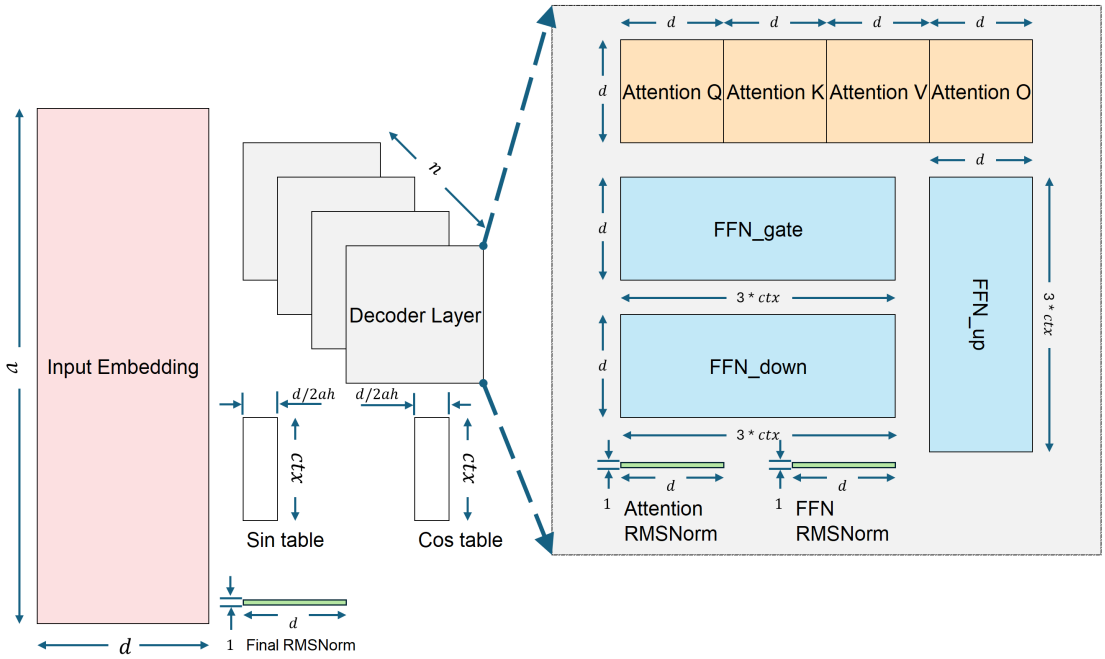

図 2.3: LLaMATemp の重みファイル内部構造図

入力埋め込みテーブル（Input Embedding） 語彙トークンをベクトル表現に写像するためのトークン埋め込み行列である。

デコーダレイヤー（Decoder Layer） Decoder_L0 から Decoder_Ln-1 まで、全体で n 層からなるデコーダアーキテクチャであることを示している。デコーダレイヤーは、マルチヘッドアテンション（MHA）、フィードフォワードネットワーク（FFN）および正規化レイヤー（RMS Norm）から構成される。

マルチヘッドアテンション（MHA）

- Attention_Q：クエリ（Query）の行列
- Attention_K：キー（Key）の行列
- Attention_V：バリュー（Value）の行列
- Attention_O：アテンション最終出力（Output）のための行列

フィードフォワードネットワーク（FFN）

- FFN_gate：FFN におけるゲーティング機構を実現する行列
- FFN_down：内部次元へのダウンサンプリングを行う行列
- FFN_up：モデル次元へのアップサンプリングを行う行列

RMS 正規化レイヤー

- Attention_RMSNorm：アテンションに入力する前の計算を安定化するための RMS（Root Mean Square）正規化パラメータ
- FFN_RMSNorm：フィードフォワードネットワークに入力する前の計算を安定化するための RMS（Root Mean Square）正規化パラメータ

2.4.2 重みとの計算

図 2.4 は LLM の全体的な動作フローを示す。図の流れのように、次の出力 Token は今まで処理した Token（初期入力の Token ＋生成済みの Token）を積み重ねた計算結果となり、自己回帰的（Auto-Regressive）な生成を行う。

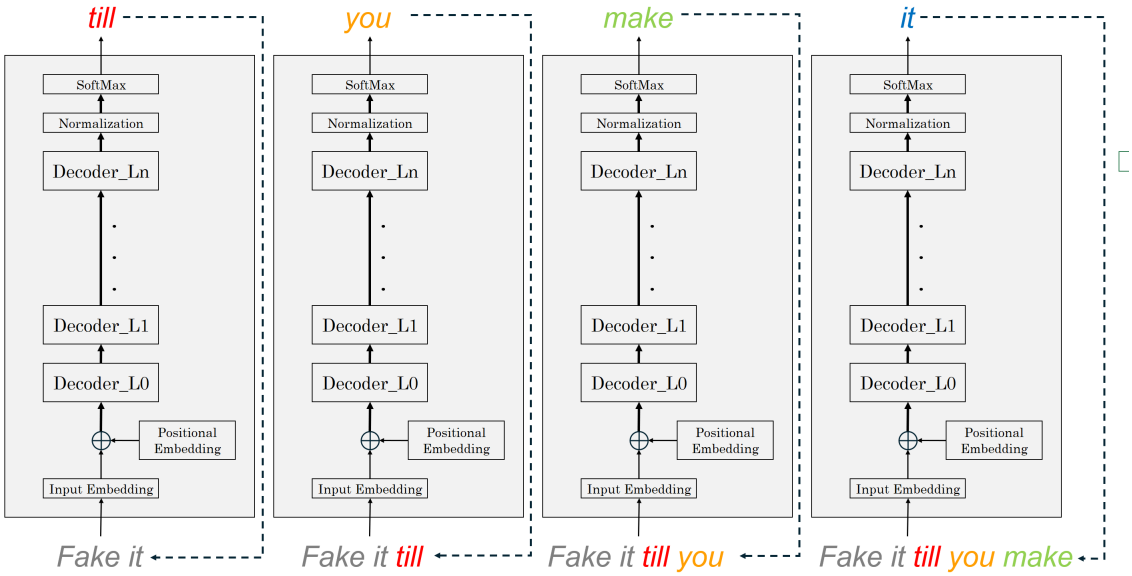

図 2.4: 重みファイル内部構造のイメージ図

LLaMATemp への最初テキストの入力後、主に三種類の変換処理が施される。入力埋め込み（Input Embedding）、デコーダ層（Decoder layer）、そして後処理（最終正規化や Softmax など）である。前述の LLaMATemp の重みファイルの構造を踏まえて、以下ではテキストが LLM に入力された後にどのような変換が行われるのか、どのような計算処理が段階的に実施されるのかを詳細に展開する。各ステップにおける計算は特定のアルゴリズムに基づいて進行し、また、各重みコンポーネントがどのように演算に関与するかについても明確にする。

Step1. 入力埋め込み（Input Embedding）

テキストがモデルに入力される際、まずトークン化（tokenization）が行われ、テキストは一連のトークンへと分割される。各トークンは、事前に定義された語彙

（vocabulary）における一つの索引に対応付けられる。その後、モデルはこの索引に基づき、予め用意された埋め込み行列（embedding table、またはルックアップテーブル lookup table とも称される）から対応するベクトル表現を検索する。このプロセスは、本質的には事前定義されたルックアップテーブルからのベクトル検索操作であり、複雑な数学的演算は伴わず、単純な配列のインデックス処理とアクセスによって実現される。

アルゴリズムの擬似コードを Algorithm 1 に示す。

Algorithm 1 Input Embedding

```
function InputEmbedding(inputText):
    tokens = Tokenize(inputText)
    embeddings = EmptyArray[len(tokens)]
for i=0; i < len(tokens) - 1; i++ do
    tokenID = FindTheIndex(tokens[i])
    embeddings[i] = embedding_weight[tokenID]
end for
return embeddings
```

テキストが Token に分割され、それぞれの Token id を検索し、取得する。重みファイル中の Input embeddings コンポーネントから、Token id と一致する行を読み出し、要素数 d のベクトルに変換される。入力埋め込み処理された最初のベクトル配列を x と記す。x が Decoder 層に入力される。

Step2. デコーダ層（Decoder layer）

図 2.2 のように、入力埋め込み（Input Embedding）から出力されるベクトル配列は以下の一連の処理を順に受ける：

2-1 正規化（Normalization）
2-2 多頭注意（Multi-head Attention）
2-3 残差接続（Residual Connection）
2-4 正規化（Normalization）
2-5 フィードフォワードネットワーク（FFN）
2-6 残差接続（Residual Connection）

2-1. 正規化（Normalization）

モデル内部のデータ分布を安定化させ、学習の収束性と効率を向上させるため、LLaMA ファミリではオリジナルの Transformer アーキテクチャで使用されていた Layer Normalization に代えて、RMS Normalization（Root Mean Square Normalization）を採用している。

RMSNorm は、入力ベクトルのスケールを調整することで、極端に大きな値や小さな値が出現するのを防ぎ、すべてのサブレイヤーにおいて一定範囲の数値分布を保つことを目的としている。入力ベクトル x に対して、RMSNorm は以下の式で定義される：

$$RMSNorm(x) = \frac{x}{RMS(x)} \cdot w$$

$$RMS(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}$$

ここで w はスケーリング係数のパラメータ (図 2.3 中の Attention RMSNorm/FFN RMSNorm/Final RMSNorm) であり、ϵ は 0 で割るエラーを防ぐために加えられる微小定数である。RMSNorm により、各層に入力されるデータはそのスケールが適切に制御され、以降の Attention 処理や FFN の演算効率および勾配伝播の安定性が向上する。

2-2 多頭注意（Multi-head Attention）

RMS Normalization の処理を終えたベクトル x は、多頭注意機構に入力される。Multi-Head Attention（以下、MHA という）は、入力情報中の各要素間の依存関係を抽出し、文脈的な表現を生成するための中核的な機構である。

（1）Query, Key, Value の生成入力ベクトル x はまず、3 つの異なる重み行列によって線形変換され、それぞれ Query（Q)、Key（K)、Value（V）ベクトルが得られる。

$$Q = xW^Q,\ K = xW^K,\ V = xW^V$$

W^Q、W^K と W^V（サイズ：$d \times d$）はそれぞれ AttentionQ、AttentionK、AttentionV の重み行列である。Q、K と V は、行列乗算変換後のベクトルである。

（2）RoPE（Rotary Positional Embedding）による位置情報の付加

LLaMA においては、従来の Positional Encoding ではなく、RoPE（Rotary Positional Embedding）による位置情報の付与が行われる。これは、Query および Key に対して周期関数（Sin/Cos）の回転変換を適用することで実現される。具体的には、入力系列中の各トークンの相対位置関係を保持したまま、線形変換の後に各次元のペアに対し回転操作を施す。この RoPE 処理により、Transformer の自己注意機構は位置の相対性を効率的に扱えるようになる。

入力埋め込みベクトルの $2i$ 番目と $2i+1$ 番目 ($i \geq 0$) の要素に対して、位置 m の単語に適用される回転は以下のように定義される。

$$RoPE(x_{2i}) = x_{2i}cos(\theta_m) - x_{2i+1}sin(\theta_m)$$

$$RoPE(x_{2i+1}) = x_{2i}sin(\theta_m) + x_{2i+1}cos(\theta_m)$$

ここで、$\theta_m = m \cdot \theta$ であり、m は単語の位置、θ は事前に定義された回転の基本周波数に関わるパラメータである。一般的には、$\theta_i = 10000^{-2i/d}$ (d は埋め込みベクトルの次元数) のように設定される。

（3）注意スコアの計算とソフトマックス正規化

RoPE 処理を施された Q および K を用いて、スカラーの注意スコアが以下のように計算される：

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V$$

上記 $d_k = d/ah$ は、1 つのヘッドあたりの次元数であり、ah はヘッド数を示す。スコア行列は Softmax 関数によって正規化され、各位置に対する注意の重みが決定される。

（4）各ヘッドの並列処理と統合

上記の処理は、ヘッド数 ah に対して別々で実施される。すなわち、Query、Key、Value のベクトルはそれぞれ ah 個に分割され、各ヘッドで独立に注意処理が行われる。全ヘッドから得られた注意出力は、次のように連結された後、出力線形変換が適用される：

$$MergeHeads(x) = [head_1; ...; head_{ah}]W^O$$

W^O（サイズ：$d \times d$）は AttentionO の重み行列である。$1 \times d$ ベクトルと $d \times d$ 行列の乗算となり、$1 \times d$ ベクトルが出力される。

2-3 残差接続（Residual Connection）

2-2 多頭注意処理の出力を $MHA(x)$ とする、第 1 回目の残差接続は単に最初に入力されたベクトル x との加算となる。以下のように定義される：

$$x_1 = x + MHA(x)$$

このような残差接続は、深層ネットワークにおける勾配消失問題の緩和、および情報の伝播を安定化する目的で導入されている。

2-4 正規化（Normalization）

2-1 と同様、LLaMA では LayerNorm の代替として RMSNorm が採用されている。残差接続後のベクトル x_1 に対して、再度 RMSNorm を適用することで、入力値のスケーリングを行う。

$$x_2 = RMSNorm(x_1)$$

これにより、次の FFN 層に渡す前に、値のスケールを一定の範囲に保つことが可能となる。

2-5 フィードフォワードネットワーク（FFN）

FFN は、Transformer 各層の中に独立して存在する 2 層の全結合層（Fully Connected Layer）から構成されているが、LLaMA ではさらに SwiGLU 活性関数を採用することで性能を最適化している。

LLaMA の FFN は以下のような計算を行う：

$$\mathrm{z} = (Swish(x_2 W_{gate}) \odot x_2 W_{down}) W_{up}$$

上記 $\odot$ は要素ごとの積（Hadamard product）、活性化関数 Swish は $Swish(x) = x \cdot \frac{1}{1+e^{-x}}$ である。

2-6 残差接続（Residual Connection）

2-5 FFN の出力ベクトルを $FFN(x_2)$ とすると、再度残差接続によって元の 2-4 の x_2 と加算され、次の Decoder 層への入力として出力される：

$$x_{out} = x_2 + FFN(x_2)$$

この x_{out} は次の Decoder 層（または出力層）に受け渡され、モデル全体で情報が深層に伝播されていく。

Decoder 層全体の擬似コードを Algorithm 2 に示す。

Algorithm 2 Decoder Layer

```
function DecoderLayer(x, AttentionQ_weight, AttentionK_weight, AttentionV_weight, AttentionO_weight, FFN_gate_weight, FFN_down_weight, FFN_up_weight, Attention_RMSNorm_weight, FFN_RMSNorm_weight):
# 2-1 Normalization
x1 = RMSNorm(x, Attention_RMSNorm_weight)
# 2-2 Multi-head Attention
Q = x1 * AttentionQ_weight
K = x1 * AttentionK_weight
V = x1 * AttentionV_weight
Q_rotary = RoPE(Q)
K_rotary = RoPE(K)
attention_output = MultiHeadAttention(Q_rotary, K_rotary, V)
MAH_output = MergeHeads(attention_output) * AttentionO_weight
# 2-3 Residual Connection
x2 = x + MAH_output
# 2-4 Normalization
x3 = RMSNorm(x2, FFN_RMSNorm_weight)
# 2-5 FFN
gate = Swish(x3 * FFN_gate_weight)
down = x3 * FFN_down_weight
ffn_output = (gate ⊙ down) * FFN_up_weight
# 2-6 Residual Connection
x_out = x2 + ffn_output
return x_out
```

以上が、第 1 層の Decoder 層における処理の全体像である。LLaMA の構造では、このような Decoder 層が N 層連続して積層されることで、深い文脈理解と生成能力を獲得している。

Step3. 後処理

全ての Decoder 層（0 層〜N-1 層）を通過した後の最終出力ベクトル x_{final} は、モデルが最終的に予測を行うための語彙空間（Vocabulary Space）に写像される。この変換により、語彙サイズ v を持つ分類問題として次トークンの確率分布が得られる。

3-1 正規化（Normalization）

2-1 及び 2-4 と同様、Decoder 最終出力に対して再度 RMS Normalization を適用

し、語彙空間への投影前に数値スケールを調整する。

$$x_{norm} = RMSNorm(x_{final})$$

3-2 Logits の算出

次に、正規化された出力ベクトル x_{norm} を用いて、語彙空間への線形射影を行う。これは、モデルが語彙のどの単語を次に出力すべきかを判断するためのスコア（logits）を生成する操作である：

$$logits = x_{norm} \cdot W_E^T$$

W_E は Input Embedding と同じ重みを転置した行列である。（重み共有、Weight tying）

logits は各語彙トークンに対するスコアのベクトルである。

3-3 Softmax による確率分布化

logits はスコアであるため、直接使用することはできない。各トークンに対する確率的な解釈を得るために、Softmax 関数が用いられる：

$$P = Softmax(logtis)$$

この処理により、全語彙に対して次に出現するトークンの確率分布が得られる。通常、この中で最も高い確率を持つトークンが出力トークンとして選ばれる。

3-4 次トークンの生成

最終的に得られた確率分布に基づき、次のトークン $token_{next}$ が決定される。モデルはこのトークンを新たな入力とし、次回のデコードステップに再帰的に入力して、文全体を逐次生成していく。

後処理の擬似コードを Algorithm 3 に示す。

Algorithm 3 Post Processing

```
function PostProcessing(x_final, Final_RMSNorm_weight, Input_Embedding_weight):
# 3-1 Final Normalization
x_norm = RMSNorm(x_final, Final_RMSNorm_weight)
# 3-2 Logits
logits = x_norm * TRANSPOSE(Input_Embedding_weight)
# 3-3 Softmax
P = Softmax(logits)
# 3-4 Sampling the next token
token_next = ArgMax(P)
return token_next
```

これらの各ステップにおける計算処理、特に大規模な行列演算や要素ごとの演算を本節で明らかにした。LLM の動作原理と計算フローを基盤として、これらの複雑な演算を FPGA 上で効率的に実装するためのハードウェアアクセラレータ設計に着手できるようになる。

2.5 FPGA

FPGA (Field Programmable Gate Array) は、プログラム可能なハードウェアデバイスであり、ユーザーがハードウェアロジックを自分の設計構想に合わせて自由に構築できる。汎用 CPU/MCU に比べ、並列化の活用による低消費電力で高い処理能力の実現が可能である。データ並列化とパイプライン並列化を工夫することで、汎用 CPU/MCU と比較して数十倍から数百倍以上の性能向上が得られる。

一般的な GPU や CPU と比較して消費電力あたりの演算性能が高く、カスタムハードウェアアクセラレーションにより、汎用プロセッサと比べてレイテンシを大幅に削減しつつ、特定の演算処理に特化した回路を設計できるため、FPGA はエッジ推論向けの LLM 応用に適している。

2.6 SoC FPGA

SoC FPGA（System on Chip FPGA）は、従来の FPGA とプロセッサを 1 つのチップ上に統合したデバイスである。FPGA の柔軟なハードウェア構成能力と、CPU のソフトウェア処理能力を組み合わせることで、効率的なシステム設計が可能になる。ARM や RISC-V などのプロセッサを搭載し、OS（Linux、RTOS など）

の実行が可能で、ソフトウェアによるハードウェアモジュールの起動等の制御が容易である。従来のディスクリート CPU ＋ FPGA よりも低消費電力・高性能・小型化が可能になる。本研究は AMD の Zynq™ UltraScale+™ MPSoC ファミリの SoC FPGA をターゲットとして、LLM を実装する。

2.7 設計パラダイムと開発フロー

SoC FPGA に対しては、ヘテロジニアス・システム・アーキテクチャに適した開発手法が求められる。本報告書では、Crockett らによる SoC FPGA 向けの設計フロー [6] を参考にして、LLM エッジ推論システムを実装する。基本的なフローを図 2.5 に示す。

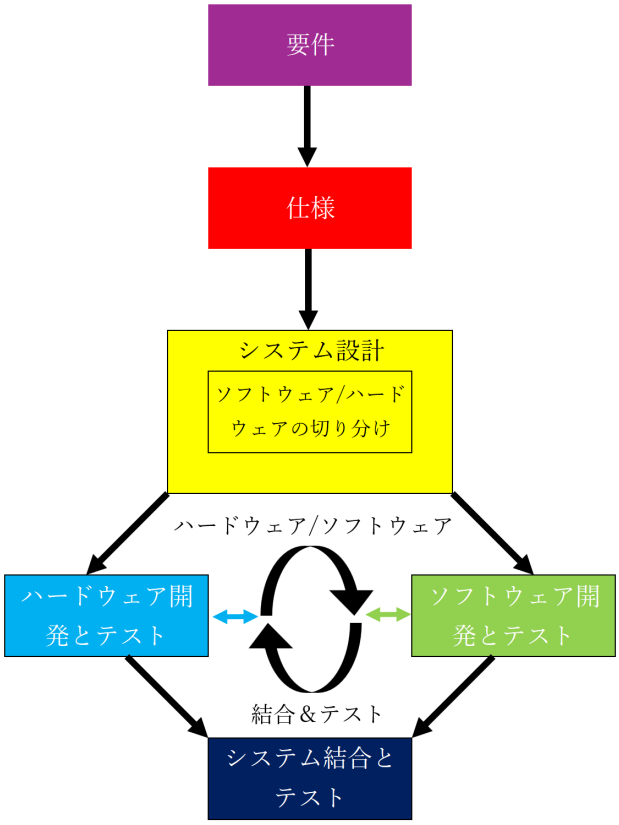

図 2.5: 設計フローの基本モデル [6]

設計の最初の段階は、システムの望ましい動作を定義すること、すなわち一連の要件から適切な仕様を作成することである。これは図の上部に開始点として示されており、その後に開発されるシステム設計の基礎となる。ARM CPU と FPGA を組み合わせた SoC FPGA 上での実装を前提とする。次に続くシステム設計段階の重要な要素は、意図した機能をソフトウェアとハードウェア間で適切に分割し、両セクション間のインターフェースを定義することである。このシステム分割により、ソフトウェア開発とハードウェア開発をほぼ並行して進めることが可能となる [6]。

LLM システム全体には、LLM 推論演算タスク以外にも、初期化や文字プリントアウトなどのタスクが存在する。ターゲットとする SoC FPGA においては、一般的なソフトウェア処理が適したタスク（ARM CPU）と、ハードウェアアクセラレーションが必要なタスク（FPGA PL）を、表 2.2 に示すように切り分ける。

表 2.2: タスク分配一覧

タスク内容	実装先	備考
初期化処理	ARM CPU	メモリ管理、初期化
モデルロード	ARM CPU	モデル重みファイルをメモリにロードする
推論演算	FPGA PL	重みとの加算/乗算など
生成文字出力	ARM CPU	生成された文字のプリントアウト
その他	ARM CPU	処理時間計測、エラー処理など

開発は、ソフトウェア開発（ARM CPU）とハードウェア設計（FPGA PL）を並行して行い、最適なバランスを探る。

・タスク分割を明確にし、FPGA の並列処理を活かす
・メモリ帯域を考慮し、モデル重み/キャッシュのデータ転送を効率化する
・HLS を活用して、アルゴリズムの実装難易度を下げる

2.8 OpenCL

OpenCL（Open Computing Language）は、スーパーコンピュータ、クラウドサーバー、パーソナルコンピュータ、モバイルデバイス、組み込みプラットフォームなど、多様なアクセラレータにおけるクロスプラットフォーム並列プログラミングを可能にする、オープンかつロイヤリティフリーの標準規格である。OpenCL は、プロフェッショナル向けクリエイティブツール、科学・医療ソフトウェア、ビジョン処理、ニューラルネットワークのトレーニングと推論など、多岐にわたる市場カテゴリーにおいて、幅広いアプリケーションの速度と応答性を大幅に向上させる [7]。

OpenCL の役割は、「CPU や GPU、FPGA など、異なるハードウェア上で並列計算を効率よく実行するための共通 API（プログラミングインターフェース）を提供すること」である。SoC FPGA 開発においては、OpenCL を用いることで、FPGA 上に実装されたハードウェアアクセラレータをソフトウェアから容易に呼び出すことが可能となる。

OpenCL は、ヘテロジニアス（Heterogeneous）なハードウェア環境における並列計算を効率的に管理するための基盤を提供し、主に以下の役割を果たす。

共通インターフェース提供 OpenCL は、CPU、GPU、FPGA といった異なるタイプのハードウェアを抽象化された「デバイス」として統一的に扱えるインターフェースを提供する。これにより、単一の API セットを用いて FPGA 上に実装されたカーネル（並列計算ユニット）との通信が可能となる。

カーネルとホストアプリケーションの分離 OpenCL は、並列計算を行うカーネル（Kernel）と、その計算を制御しデータ転送を司るホストアプリケーション（Host Application）の明確な分離を提唱する。

ホストアプリケーション: カーネルの起動、データ転送、実行フローの制御を行う部分であり、CPU などの汎用プロセッサ上で動作する。

カーネル: 実際の演算処理を実行する部分であり、FPGA などのアクセラレータ上で動作する。

データ転送と実行制御の抽象化 OpenCL は、デバイス間のデータ転送とカーネルの実行制御に関する操作を抽象化された API として提供する。これにより、開発者は低レベルなハードウェアの詳細を意識することなく、並列処理を記述できる。API の例を表 2.3 に示す。

表 2.3: OpenCL API の例

機能	API の例	説明
デバイス選択	clGetDeviceIDs()	目的の FPGA デバイスを選択する
メモリ確保	clCreateBuffer()	FPGA デバイス上のメモリ領域を確保する
カーネルロード	clCreateProgramWithBinary()	コンパイル済みカーネルバイナリを FPGA にロードする
カーネル起動	clEnqueueNDRangeKernel()	FPGA 上のカーネル処理を開始する
結果取得	clEnqueueReadBuffer()	処理結果を CPU メモリに転送する

FPGA 設計の高位抽象化 通常、FPGA を設計するには HDL（Hardware Description Language）である VHDL や Verilog HDL を用いた記述が必要となる。しかし、OpenCL を用いることで、C 言語ベースで FPGA アクセラレータを記述することが可能となり、開発の難易度が大幅に低下する。Xilinx Vitis HLS や Quartus Prime などの高位合成（High-Level Synthesis: HLS）ツールと組み合わせることで実現される。

2.9 HLS

効率的な FPGA 利用を実現するには、一般的に VHDL や Verilog HDL を用いた RTL 設計が不可欠であるが、これはソフトウェア開発と比較して著しく高い人的・時間的リソースを要する。特に複雑なアルゴリズムの RTL 記述においては、

その煩雑さゆえに実装が困難であり、バグの発生源となる可能性も高い。さらに、アプリケーション固有のアーキテクチャ設計が最適化された場合には卓越した演算性能が達成可能である一方、そうでない場合には従来のプロセッサや ASIC に対する優位性の確保は困難となる。したがって、FPGA で高性能処理を実現するためには、アーキテクチャ設計における綿密な検討と反復的な最適化が求められ、これが開発コスト増大の主要因となっている。

HLS（**H**igh **L**evel **S**ynthesis）は、ソフトウェア開発に類似した手法で FPGA 開発を可能にする技術であり、プログラミング言語からハードウェアロジックへの自動変換を実現するものである [8]。特に C/C++言語で記述されたプログラムをハードウェアロジックへと変換する高位合成ツールは、複数のベンダーにより開発・市場投入がなされている。これまで実用レベルのツールは高額であったが、2018 年以降、主要 FPGA メーカーである AMD と Intel が、それぞれ AMD Vitis™ HLS および Intel HLS コンパイラという無償ツールの提供を開始したことにより、その導入障壁は大幅に低減された。

C/C++の全機能を活用することはできないものの、制御構文（ループ構造や条件分岐）や関数によるモジュール化といったソフトウェア設計の利点を享受することが可能である。特筆すべきは、ソフトウェア実装で頻出する多重ループ構造に対して、パイプライン化やループアンローリングといったハードウェア最適化技術を適用できる点である。さらに、目標動作周波数に基づいた実装方式の自動探索機能により、場合によっては手動でのハードウェア設計よりも効率的な回路生成が実現可能となっている。

2.10　まとめ

本章では、大規模言語モデル（LLM）の基礎知識から、FPGA（Field Programmable Gate Array）を活用したエッジ推論のための予備知識について体系的に整理した。まず、LLM の概要とその分類について説明し、クローズドソースとオープンソースという 2 つの主要なカテゴリ、およびそれぞれの特徴と代表的なモデルについて述べた。クローズドソース LLM は、OpenAI の ChatGPT や Anthropic の Claude のように、高性能だが利用に制約がある一方、オープンソース LLM は、Meta の LLaMA や Google の Gemma のように、研究や商業利用向けに公開されており、カスタマイズの自由度が高い。

次に、LLM の主要なアーキテクチャとして Transformer モデルを紹介し、その構造と特徴について詳細に説明した。Transformer は、従来の RNN や LSTM が抱えていた逐次処理の限界を克服し、Attention 機構によって並列処理を可能にすることで、長文脈の学習を実現した。Encoder-Decoder 構造を基本とし、Input Embedding、Positional Encoding、Attention、Feed Forward Network などの要素から構成される Transformer は、LLM の発展に不可欠な役割を果たしている。

さらに、オープンソースLLMであるLLaMAを例にとり、その内部構造と重みファイルの構成、およびテキスト生成の基本的な仕組みについて解説した。LLaMAは、Transformerの基本構造を踏襲しつつ、Layer Normalizationの位置変更、Swish活性化関数の導入、Rotary Positional Embeddings（RoPE）の採用などの改良を加えている。重みファイルは、Input Embedding、Decoder Layer、三角関数LUTから構成され、各層でAttentionやFFNなどの計算が行われ、最終的にSoftmax関数によって次のトークンの確率分布が得られる。

最後に、FPGAの特性とLLM推論への応用可能性、および開発効率化のためのHLS（High Level Synthesis）ツールについて概説した。FPGAは、並列処理による高い演算性能と電力効率、およびハードウェアの柔軟な再構成能力を兼ね備えており、エッジデバイスにおけるLLM推論に適している。HLSは、C/C++言語などで記述されたプログラムからハードウェア記述言語（VHDL、Verilog HDL）を自動生成する技術であり、FPGA開発の生産性を大幅に向上させることができる。

本研究では、これらの予備知識を基に、LLMのエッジ推論をFPGA上で最適化する手法を探求する。次章以降では、具体的な設計手法や評価手法について詳細に検討し、FPGA実装の実験結果を示していく。

第3章　既存研究

本章では、3.1 節で GPT-2 のマルチ FPGA 実装に関する研究、3.2 節で LLM 推論の FPGA 上の完全なマッピングフローに関する研究について述べる。3.3 節では二つの既存研究のまとめと考察を述べる。

3.1　GPT-2 のマルチ FPGA 実装に関する研究

Hong, Seongmin らは DFX という GPT-2 モデルの推論を高速かつ高効率で実行するマルチ FPGA アクセラレータ [1] を提案した。GPU がシーケンシャルな生成ステージで性能劣化する課題を解決し、データセンター向けに低遅延・高スループットのテキスト生成を実現した。DFX の最も新規性のある点は以下の三つとなる。

生成ステージの最適化 GPU は並列計算に優れているが、LLM の生成ステージにおける逐次処理は GPU のアーキテクチャと相性が悪く、ハードウェアリソースを十分に活用できないという課題がある。この課題に対し、DFX はシングルトークン処理に特化した FPGA 計算コアを開発することで、従来の GPU ベースの推論と比較して大幅な低遅延化を実現した。

マルチ FPGA によるモデル並列化 Transformer モデルは巨大なパラメータ数を有するため、単一の FPGA や GPU では処理が困難な場合がある。既存の並列化手法としては、データ並列やパイプライン並列などが知られているが、DFX はこれらに加えて複数の FPGA を利用する新たなモデル並列化手法を提案した。

具体的な実装と高い性能向上率の実証 単なる理論的な提案に留まらず、実際にマルチ FPGA システムを構築し、GPT-2 などの具体的な Transformer モデルを用いて評価を行った。DFX は商用 GPU と比較して、特に生成段階でのレイテンシを大幅に削減できることを明確に示した。

DFX は、Transformer ベースのテキスト生成におけるリアルタイム処理の課題に対し、マルチ FPGA 構成（Xilinx Alveo U280 × 4 基）による革新的なアプローチを提案し、商用 GPU（NVIDIA V100）と比較して顕著な高速化を実現した。特に生成フェーズにおいては、GPT-2 Small で約 2.5 倍、BERT では約 3.5 倍の高速化を達成し、推論処理の応答性を飛躍的に向上させた点が注目される。従来困難とされてきた大規模言語モデルの低レイテンシ推論に対する具体的かつ実効性のある解決策を提示した。

3.2 LLM 推論の FPGA 上の完全なマッピングフローに関する研究

Zeng, Shulin らは、計算効率の向上とメモリ帯域の最適化等による Transformer ベース LLM 推論の効率化を達成する革新的なソリューション「FlightLLM」を提案した。

計算効率の向上 コンフィギュラブルなスパース DSP チェーンを提案し、ブロックスパースや N:M スパースパターンをサポートしながら高効率な演算を実現した。

メモリ帯域の最適化 オンチップデコード方式を導入し、デコード段階でのメモリアクセスを抑制する。また、混合精度量子化をサポートし、メモリ帯域の使用効率を向上した。

コンパイルオーバーヘッドの削減 入力トークン長の多様性や、モデルの最適化（スパース化など）は、必要な命令量が最大 1.67TB にも達し、搭載可能なメモリ容量を超過するという課題を引き起こしていた。FlightLLM はこの課題に対し、命令の使い回しと HBM（High Bandwidth Memory）アクセス命令の統合によって対処した。この手法により、命令サイズを約 3.25GB まで大幅に削減することに成功し、コンパイルオーバーヘッドの削減を実現した。

FlightLLM は、Xilinx Alveo U280 に実装され、LLaMA2-7B や OPT-6.7B などの最新 LLM に対して、NVIDIA の V100S GPU と比較して 6.0 倍のエネルギー効率、1.8 倍のコスト効率を達成した。また、最新の Xilinx Versal VHK158 では、NVIDIA A100 より 1.2 倍のスループットを実現した。

これにより、従来の GPU や FPGA アクセラレータを超えるエネルギー効率・コスト効率を達成し、FPGA を活用した LLM 推論の新しい可能性を示した。

3.3 まとめ

本章では、LLM 推論の FPGA 実装に関する既存研究を考察した。Hong, Seongmin らによる DFX[1] は、GPT-2 の逐次的な生成ステージの最適化、マルチ FPGA を活用した新たなモデル並列化手法、HBM を活用したメモリアクセスの最適化により、GPU の課題を克服し、5.58 倍の推論速度向上、3.99 倍のエネルギー効率向上を実現した点が特徴である。Zeng, Shulin らの FlightLLM[2] は、スパース DSP チェーンによる計算効率の向上、オンチップデコード方式を採用したメモリ帯域の最適化、長さ適応型コンパイルによるオーバーヘッド削減を実現し、従来の GPU と比較して最大 6.0 倍のエネルギー効率を達成した。特に、最新の FPGA デバイス（Versal VHK158）では NVIDIA A100 GPU を超えるスループットを報告した。

これらの既存研究は、FPGA の LLM 推論タスクに対するアクセラレーションの有効性を証明し、FPGA ベースの LLM 推論が今後の AI アクセラレーションの有力な選択肢となる可能性を示した。ただし、これらの先行研究は、データセンターグレードの高性能 FPGA を対象としている。DFX や FlightLLM が採用した手法は、大規模なロジックセルと HBM を前提としているため、エッジデバイスの限られたリソース環境では直接適用することが困難である。

本研究では、これらの既存研究の知見を踏まえて、エッジコンピューティング環境特有の制約（限られたロジックセルリソース、メモリ帯域）に適応するための新たなアプローチを探究する。アクセレータモジュールの簡素化と再利用化を施して、エッジデバイス向けの FPGA デバイスでも効率的に LLM 推論を実行できるフレームワークを目指す。これにより、既存の高性能向け研究を補完し、FPGA ベースの LLM 推論の応用範囲をエッジ領域まで拡張することを目的とする。次章では、具体的なシステム設計、実装手法と実験結果について詳述する。

第4章　実装手法

本章では、システムの仕様決定、モデルおよびハードウェアプラットフォームの選定、そして実機実装に至る一連の流れをまとめる。具体的には、4.1 節でシステムの要件と仕様を定義し、4.2 節でその仕様に基づいたモデルとハードウェアプラットフォームの選定について述べる。さらに、4.3 節でシステムの構造、動作フロー、ホストアプリケーション、およびアクセラレータカーネルについて詳述する。

4.1　システム全体の要求仕様

本研究はリアルワールドに実用性のあるシステムを目指すため、想定される応用シナリオとシステムのコア性能仕様を本節に定義する。

4.1.1　システムの目的

本システムは、LLM の推論を FPGA 上で実装し、完全スタンドアローンで動作するエッジ AI デバイスを開発することを目的とする。従来のクラウドサーバー依存型 LLM サービスとは異なり、ネットワーク接続不要な推論処理を実現し、産業用途や組み込みシステムでの活用を可能にする。

4.1.2　想定用途と応用場面

低レイテンシ、インターネット接続不要かつエネルギー効率の高いシステムであるため、以下のような製品と用途例での利用を想定する。

人型ロボット・ペットロボット: ユーザーの視線、ジェスチャー、音声などのインプットを理解し、曖昧な指示も正確に汲み取る。

手持ち/ウェアラブル端末: 専用翻訳ツールとして、ユーザーの音声を認識し、リアルタイムでターゲット言語に翻訳する。

スマートコックピット・ADAS: ADAS（先進運転支援システム）機能中の会話型アシスタントとして、ドライバーや同乗者との自然な雑談、運転中の情報提供をする。音声で車の機能（ナビ、エアコン、窓の開閉など）を制御する。

4.1.3 性能仕様

前述の想定用途のように、実際の製品では用途に応じてIOポート、外部接続インターフェース、周辺ハードウェアなどが変化する。それぞれ詳細な仕様は異なるものの、共通のコア仕様として以下の三つが考えられる。すなわち、リアルタイム性、文脈保持能力、および計算資源制限下での推論能力である。

本研究では、これらの3つのコア仕様を研究対象とスコープとして設定する。各項目内容、定義、および目標値を表4.1に示す。

表 4.1: コア性能仕様

項目名	定義	目標値
応答時間	入力テキストを受けてから最初の生成テキストを出力するまでの時間	1秒以下
コンテキスト長さ	最大保持できるコンテキスト（文脈）の文字数	200文字以上
推論速度	1単語あたりの平均所要時間	500ms以下（2単語/秒）

応答時間 ユーザーが著しい遅延を感じることなく、自然な対話が可能なシステムを実現するためには、ユーザーからの入力を受けてから出力が開始されるまでの時間を一定時間内に収める必要がある。Nielsen Norman GroupのUX研究[8]によれば、以下のような結論が示されている。1秒は自然な対話を維持するための「心理的上限」であると考えられる。

0.1秒以下: 即時反応として無意識に認識される。

1秒以下: インタラクションが自然だと感じられる上限。

10秒超: ユーザーの集中が途切れる。

コンテキスト長 通常の対話文や説明文は、100〜200文字で一つの意味単位（文脈）を形成する[10]。軽量なタスクを処理するシステムにおいては、200文字程度のコンテキスト長があれば、2〜3回分の対話履歴や因果関係を保持でき、十分な文脈理解が可能であると判断される。

推論速度 人間の自然な発話速度は、話し方のスタイルによって異なるが、一般的な日常会話やニュース読み上げは約3〜5語/秒。高齢者向け、子ども向け、入門教材などで使用される話速は約1.5〜2.5語/秒。初級者向けリスニング教材、音読指導などは約1〜1.5語/秒。このうち、約2語/秒（= 500ms/語）は「ゆっくりだが自然」と感じられる下限に相当し、リスニングや読解においても快適に情報を処理できるテンポである。

4.2 実機実装

前節において本システムの仕様および性能要件を定義した。これに基づき、本節では要求仕様に従い、トップダウン方式で本研究における提案実装手法について述べる。本節では、LLM モデルの選定と構造説明、ハードウェアプラットフォームの選定、システムアーキテクチャ、システム動作フロー、ホストアプリケーション、およびアクセラレータカーネルの構成について詳述する。

4.2.1 LLM モデルの選定

エッジデバイスにおいて LLM 推論を実行する場合、利用可能なメモリおよび演算性能に制約があるため、推論性能をある程度維持しつつ、モデルを小型化するというトレードオフを考慮する必要がある。一般的に、LLM の性能はパラメータ数に比例すると考えられている。パラメータ数が多いほど、モデルの精度や生成されるテキストの品質が向上する傾向が見られる [11]。しかしながら、エッジデバイスに用いられるハードウェアは、クラウドサーバーと比較してメモリ容量に制約があるため、大規模な LLM を選択することが困難である。ハードウェアプラットフォームのメモリ容量が明確となれば、理論的に搭載可能な LLM のパラメータ数の上限を算出することができる。必要メモリ容量と LLM のパラメータ数の概算関係は、以下の式により表される。

$$M = (Para \cdot Prec/8) \cdot 1.2$$

または

$$Para = (8 \cdot M/(1.2 \cdot Prec))$$

M： メモリサイズ（単位：GB）

$Para$： モデルパラメータ数（単位：Billion）

$Prec$： パラメータ精度ファクター（FP32 = 32、FP16 = 16、INT8 = 8）

他のオーバーヘッドや KV(key-value) キャッシュのメモリ占有のため、20 %のマージンを計算に入れている。例として、FP16 の使用を前提として、4GB メモリのハードウェアプラットフォームがサポートできる最大パラメータ数は、$8\times4/(1.2\times16) = 1.67B$ となる。

一方、LLM の小型化に関する研究も進められている。Eldan Ronen と Yuanzhi Li は、論文「Tinystories: How small can language models be and still speak coherent English?」（以下「Tinystories」という）において、一貫性のある英語文章を生成

できるLLMがどこまでコンパクトになり得るかを探求した。彼らは、適切なデータセットと訓練方法を用いれば、10M（1,000万）パラメータ程度でも「自然で一貫した英語」が生成可能であることを報告している[12]。このことから、特定タスク向けのLLMパラメータ数の下限は10M前後であると考えられる。

エッジデバイスへLLMを実装する際は、前述のパラメータ数の上限と下限を考慮し、実際の用途に合わせた適切なモデルを選定またはトレーニングする必要がある。商用アプリケーションにおいて出力品質とユーザーエクスペリエンスを向上させるためには、特定のシナリオに特化したカスタムの小型エッジLLMを訓練するのが一般的である。しかし、本研究の主目的は、エッジデバイスにおけるLLMのリアルタイム推論の実現と最適化にあるため、モデルの訓練プロセスは研究対象から外す。

論文Tinystoriesの提案をベースにして、Karpathyは教育・研究向けのプロジェクトtinyllamas[13]を立ち上げた。Tinystoriesで使われたデータセット[14]とトレーニング手法を用いた、ミニストーリーを生成するモデルtinyllamasを公開した。モデルのパラメータサイズやレイヤー数などのコンフィグレーションについては、表4.2に示す通りである。あわせて、参考として他のオープンソースLLMの仕様も併記する。

表4.2: モデルコンフィグレーション一覧

	tinyllamas	GPT-2 Small	LLaMA2-7B	LLaMA2-13B
Parameter	15M	117M	7B	13B
Dim_model	288	768	4096	5120
N_layer	6	12	32	40
Att_heads	6	12	32	40
Vocab_size	32000	50257	32000	32000
Context_length	256	1024	4096	4096

表4.2が示す通り、tinyllamasはモデル規模と入力テキスト長の両面において、本研究で設定した要件を満たしている。その軽量な構造は、エッジデバイスの計算リソース制約に適応し、高い実装可能性を有する。これらの理由から、本研究ではtinyllamasを対象モデルとして選定し、システムへの実装と評価を行う。

4.2.2 tinyllamasの構造

tinyllamasモデルはLLaMA2と同じアーキテクチャとTokenizerを持つ小型モデルである。2.5.1節と同様に、tinyllamasも類似の構造になっている。図4.1は、tinyllamasにおける重みファイルの内部構造を示す。

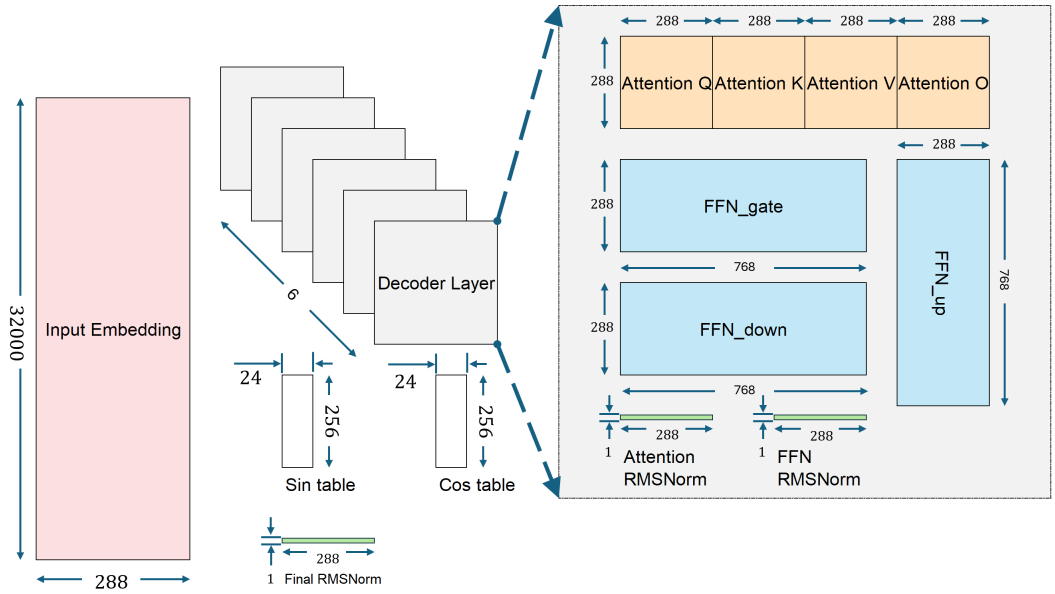

図 4.1: tinyllamas 重みファイル内部構造図

4.2.3 ハードウェアプラットフォームの選定

4.1.2〜4.1.3 節において定義した想定用途およびコア性能仕様に基づき、本研究では、データセンター向け FPGA よりも低消費電力かつ低コストでありながら、tinyllamas モデルの実装に必要な十分なリソースを有するハードウェアプラットフォームが求められる。

ハードウェアプラットフォームを選定するにあたっては、以下の主要要素を総合的に考慮する必要がある。

ハードウェアリソース: ロジックセルなどのリソースが実装要件を満たしているか、ならびにオペレーティングシステム（OS）の搭載が可能であるかを評価する必要がある。

消費電力: 低消費電力モードの有無や、バッテリー寿命がプロジェクト要件を満たすかを確認すべきである。

インターフェース設計: 各種センサーや通信モジュールの接続要件に対応可能かどうかを検証する必要がある。

開発エコシステム: 統合開発環境（IDE）ツールの完成度、公式ドキュメントの整備状況、リファレンスデザインなども重要な評価項目となる。

コスト: 想定製品の予算内に収まるかどうかも、選定における重要な意思決定要因である。

上記選定フレームワークに従い、ロジックセルが100k以上、外部メモリ1GB以上のIoT/エッジデバイス搭載向けミッドレンジSoC FPGA流通品を下記の表4.3にまとめた。

表 4.3: 候補SoC FPGA評価ボード

項目	Ultra96-V2	ZCU104	Kria KV260	Cyclone V SoC Dev Kit
SoC型番	ZU3EG	ZU7EV	ZU5EV	5CSEBA6U23I7
ロジックセル	約154K	約504K	約256K	約110K
ARM CPU	4コア	4コア	4コア	2コア
外部メモリー	LPDDR4 2GB	DDR4 2GB	DDR4 4GB	DDR3 1GB
OS搭載可否	PetaLinux対応	PetaLinux対応	Ubuntu、PetaLinux対応	Linux対応
消費電力	約5W	約10W	約7.5W	約4–6W
インターフェース	MIPI-CSI、USB 3.0、Wi-Fi	FMC、SFP+、USB 3.0	USB 3.0、DisplayPort、Ethernet	SPI、UART、USB 2.0、Ethernet
開発エコシステム	Vitis（HLS対応）、PetaLinux、リファレンスデザイン			Intel Quartus Prime（HLS対応）、Embedded Linux
参考価格	約$249	約$1,000	約$199	約$225

必要なFPGAロジックと外部メモリを十分に備え、適度な消費電力がエッジ環境に適しているため、本研究はAMD Xilinx 製 Kria™ KV260開発ボード（以下KV260とする）を選定した。KV260は、エッジAI向けのK26 SOM（System On Module）評価プラットフォームである。図4.2のように、ベースとなるキャリアカード、K26 SOM、および放熱ファンで構成される。詳しいハードウェアスペックを表4.4に示す。

図 4.2: KV260 評価ボード

表 4.4: KV260 スペック表

項目	スペック
SoC FPGA	Xilinx Zynq UltraScale+ MPSoC ZU5EV
FPGA(PL)	
ロジックセル	256,200
FF	234,240
LUT	117,120
DSP スライス	1,248
BRAM (Mb)	5.1
UltraRAM (Mb)	18
CPU(PS)	Quad-core ARM Cortex-A53
	Dual-core ARM Cortex-R5F
メモリ	DDR4 4GB
ストレージ	microSD
I/O インターフェース	GPIO、UART、1Gb Ethernet、USB3.0
消費電力	約 10W（TDP）

4.2.4　システムアーキテクチャ

従来の FPGA IC と異なり、KV260 の SoC FPGA（型番 ZU5EV）は ARM CPU をベースとするプロセッシングシステム (以下 PS とする) とプログラマブルロジック (以下 PL とする) を 1 つの IC チップに組み合わせたものである。ARM CPU と FPGA のヘテロジニアス・マルチコアの性能をいかに効果的かつ最大限に引き出すかは、システムアーキテクチャ設計における重要な課題の一つである。

AMD 公式ソフト開発ガイド [15] によると、KV260 は BareMetal、Linux、およびサードパーティの三種類のソフトウェアスタックをサポートしている。2.7 節の表 2.2 で示したように、LLM 推論演算タスク以外にも初期化や文字プリントアウトなどのタスクが存在するため、これらの各タスクをプロセッシングシステム（PS）とプログラマブルロジック（PL）に適切に切り分ける必要がある。したがって、多様なタスク管理と柔軟な開発環境を考慮すると、Linux OS の搭載が最も適切なソフトウェアスタックであると考える。

SMP（Symmetric Multiprocessing：対称型マルチプロセッシング）モードは、マルチコアプロセッサシステムに広く採用されている動作パラダイムであり、単一のオペレーティングシステムインスタンスの下で、すべてのプロセッサコアを一元的に管理することを基本理念としている。SMP モードは、システム開発の簡素化、リソース利用率の向上、およびシステムスループットの改善といった利点を提供する。統一的なスケジューリング機構、高効率な開発性、および保守性の高さから、SMP は標準的かつ優先的な選択肢とされている。本研究は AMD ファーストパーティの PetaLinux を用いて、SMP モードでモデル実装を行う。

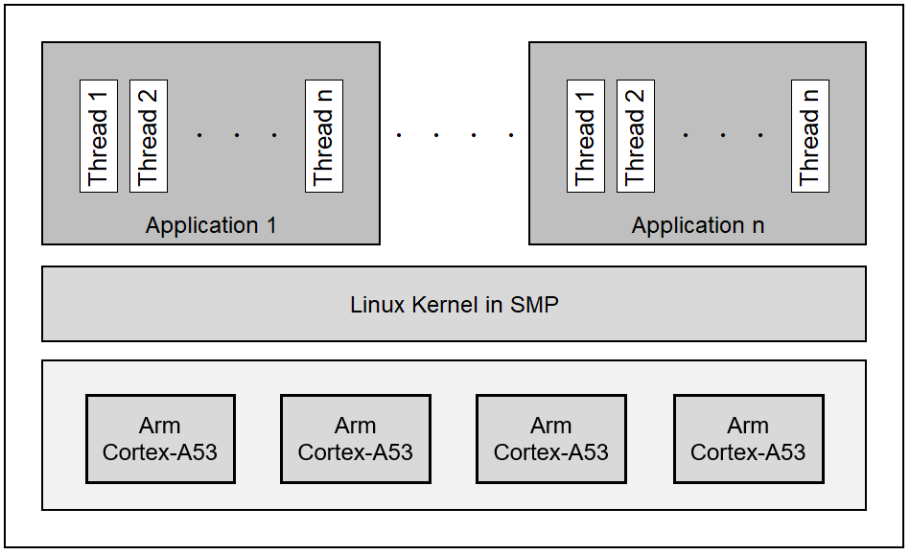

図 4.3: SMP モード Linux の例 [15]

Xilinx Runtime (以下、XRT とする) は、FPGA デバイスの管理・制御を担うオープンソースのソフトウェアスタックである。XRT は、ユーザーのホストアプ

リケーションと FPGA 上のアクセラレータとの間の通信を仲介する役割を果たす。具体的には、ユーザー空間からの XRT API 呼び出しがカーネル空間の XRT ドライバに渡され、FPGA の構成、計算カーネルの実行、メモリ転送などのハードウェア処理が実行される。システムの階層構造（Xilinx 版 OpenCL では、XRT がこのランタイム環境を提供する。）を図 4.4 に示す。

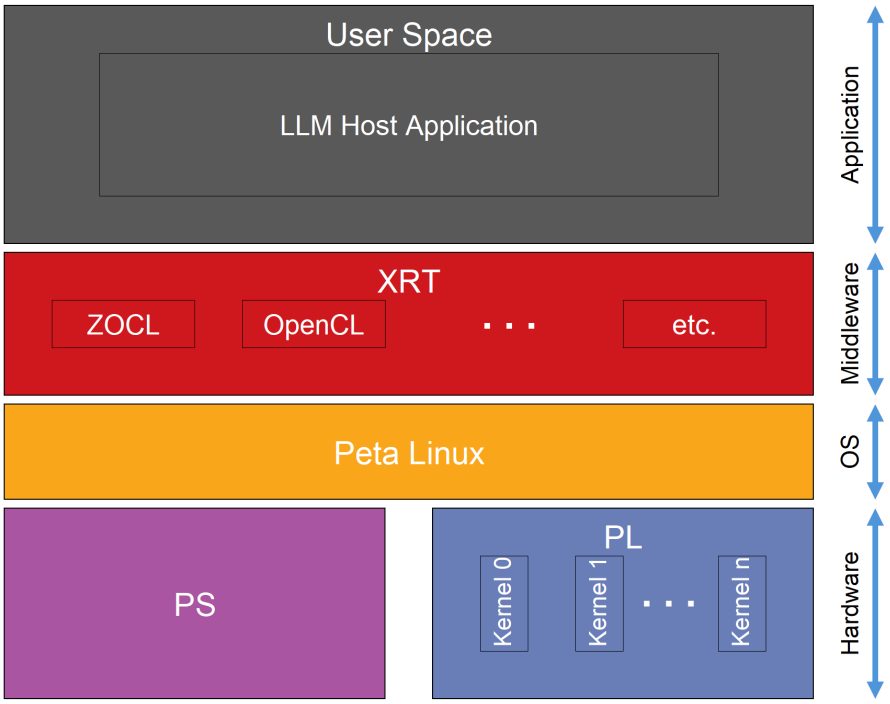

図 4.4: システム階層図

XRT は、ホスト側からの API 呼び出しをドライバ（ZOCL）に変換して FPGA に処理を指示し、完了待ち、結果取得、およびデータ転送までの一連の制御を行うことで、KV260 上でのハードウェアアクセラレーションを実現する。

2.4.2 節の LLM 動作フローによると、一つのトークンを生成するためには、重みを用いた 3 つの主要な計算処理、すなわち入力埋め込み（Input Embedding）、デコーダレイヤー（Decoder Layer）、および後処理が施される。これら全ての処理を FPGA に実装できれば理想的である。しかし、このアプローチは膨大なハードウェアリソースを必要とし、アクセラレータカーネルが tinyllamas 専用となる。その結果、モデルが更新・変更されるたびにアクセラレータカーネルを修正する必要が生じ、柔軟性が失われることになる。したがって、モデルのフル実装より、計算処理をより基礎的で、共通化・再利用可能な計算タスクへと細分化する方が望ましい。入力埋め込み、デコーダ層、および後処理の計算内容を表 4.5 にまとめた。

表 4.5: 計算タスク分解表

	モデル推論時における処理	分解後の基礎計算
入力埋め込み	入力埋め込み（Input Embedding）	-(単なる検索と読み込み)
デコーダ層	正規化（Normalization）	スカラー乗算、加算、開平算、割り算
	多頭注意（Multi-head Attention）	スカラー乗算、加算、行列乗算、Softmax 関数
	RoPE（Rotary Positional Embedding）	三角関数、乗算、加算
	残差接続（Residual Connection）	加算
	正規化（Normalization）	スカラー乗算、加算、開平算、割り算
	フィードフォワードネットワーク（FFN）	スカラー乗算、加算、開平算、Swish 活性化関数、行列乗算
	残差接続（Residual Connection）	加算
後処理	正規化（Normalization）	スカラー乗算、加算、開平算、割り算
	Logits の算出	行列乗算
	Softmax による確率分布化	Softmax 関数

上記の分解表に基づき、基礎計算のうち加算、スカラー乗算、行列乗算をそれぞれ独立したアクセラレータカーネルとして実装する。これらは、add_kernel、mul_kernel、matmul_kernel として定義する。同様に、正規化、RoPE、および Softmax 関数も機能ごとに独立したアクセラレータカーネルとする。これらは、rmsnorm_kernel、rope_kernel、softmax_kernel として定義する。イメージ図を図 4.5 に示す。

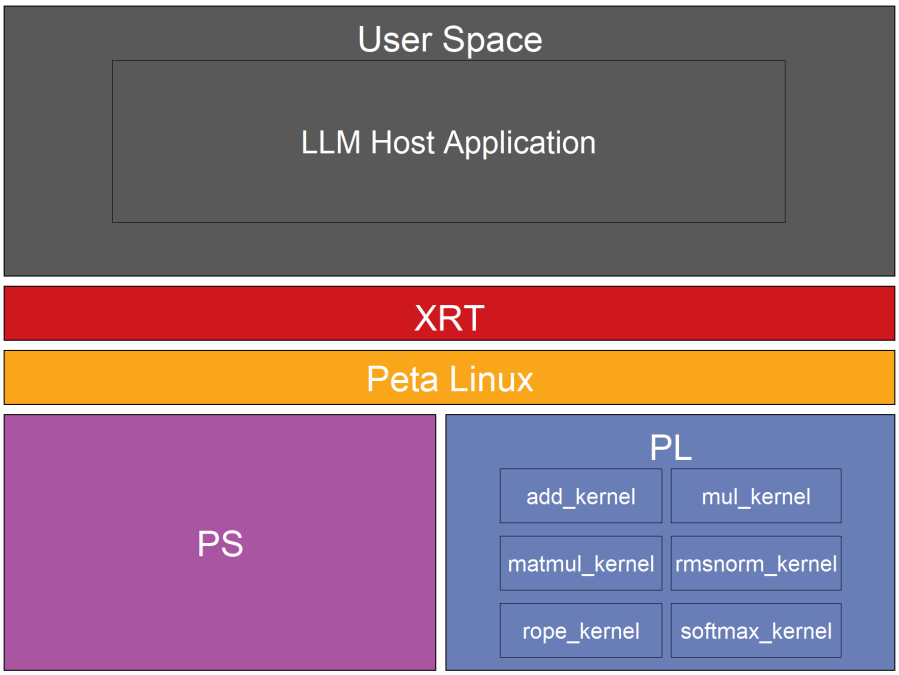

図 4.5: カーネル実装イメージ図

4.2.5 システム動作フロー

システム全体の動作の流れを図 4.6 に示す。ホストアプリケーションを実行後、次の三つのフェイズで動作する。

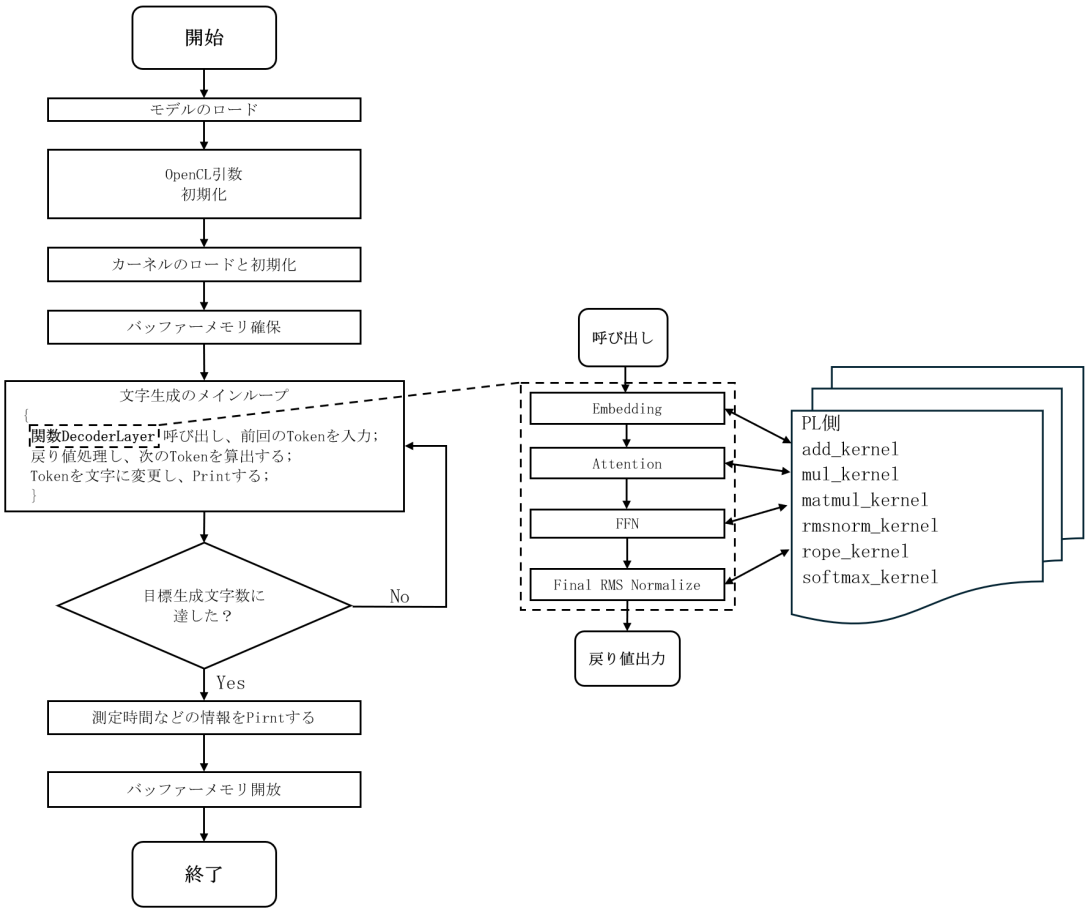

図 4.6: システムアーキテクチャ図

1. 初期化フェイズ 文字生成前の準備

モデルロード tinyllamas の重みファイルをメモリにロードする。

引数初期化 OpenCL の設定を初期化し、必要なデバイスやコンテキストの設定を行う。

カーネル初期化 モデル内で使用されるカーネルを読み込み、初期化する。

メモリ確保 必要なデータ領域（入力・中間・出力用のバッファ）を確保する。

2. 文字生成ループ 入力トークンに基づいて、モデルが次のトークンを予測・出力する。

- 関数 DecoderLayer を呼び出す。
- 前回出力を入力する。
- モデルを順伝播し、次のトークンを取得。

- トークンを文字に変換し、画面に出力（Print）。
- 生成文字列が指定長に達したかを判定する。再びループするか、あるいは終了処理へ進む。

3. 終了処理 処理時間などのログ情報を出力した後、使用したメモリ領域を解放し、プログラムを終了する。

4.2.6 ホストアプリケーション

ホストアプリケーションのソースコードを，Listing4.1 に示す．動作フローの通りに、初期化、文字生成及び終了処理の順で機能する。

Listing 4.1: ホストアプリケーションのソースコード

```
#include <Everything you need>
FUNCTION main()
  // 1.  Load model parameters and vocabrary.
  OPEN weight_fs FROM weight_path
  CALL LoadWeights(weights, tok_emb_table, weight_fs)
    // Load model weights and token embeddings from the file.
  CLOSE weight_fs
  OPEN vocab_fs FROM vocab_path
  CALL LoadVocab(vocab, vocab_fs)
    // Load vocabrary from the file.
  CLOSE vocab_fs
  ------------------------------------------------------
  // 2.  OpenCL initialization
  OPEN Xclbin_fs FROM Xclbin_path
  LoadXclbin(Xclbin, Xclbin_fs)
    // Load accelerator kernels from the file.
  CLOSE Xclbin_fs
  CALL OpenCLMacro(err, context = cl::Context(device, NULL, NULL,
      NULL, &err))
    // Creates an OpenCL context for the FPGA.
  CALL OpenCLMacro(err, q = cl::CommandQueue(context, device,
     CL_QUEUE_PROFILING_ENABLE, &err))
    // Creates a command queue for the FPGA.
  DECLARE program AS cl::Program = cl::Program(context, {device},
      Xclbin, NULL, &err)
    // Creates an OpenCL program from the binaries.
  CALL OpenCLMacro(err, add_kernel = cl::Kernel(program, "
     add_kernel", &err))
    // Creates a kernel object for element-wise addition.
  CALL OpenCLMacro(err, mul_kernel = cl::Kernel(program, "
     mul_kernel", &err))
```

```
27    // Creates a kernel object for element-wise
          multiplication.
28  CALL OpenCLMacro(err, matmul_kernel = cl::Kernel(program, "
      matmul_kernel", &err))
29    // Creates a kernel object for matrix multiplication.
30  CALL OpenCLMacro(err, rmsnorm_kernel = cl::Kernel(program, "
      rmsnorm_kernel", &err))
31    // Creates a kernel object for RMS normalization.
32  CALL OpenCLMacro(err, rope_kernel = cl::Kernel(program, "
      rope_kernel", &err))
33    // Creates a kernel object for Rotary Positional
          Embedding.
34  CALL OpenCLMacro(err, softmax_kernel = cl::Kernel(program, "
      softmax_kernel", &err))
35    // Creates a kernel object for softmax computation.
36  CALL OpenCLMacro(err, DECLARE buffer AS cl::Buffer = cl::Buffer
      (context, CL_MEM_READ_WRITE, size_in_bytes, NULL, &err))
37    // Allocates memory for buffer
38  CALL OpenCLMacro(err, ptr2buffer = q.enqueueMapBuffer(buffer,
      TRUE, CL_MAP_WRITE, 0, size_in_bytes, NULL, NULL, &err))
39    // Maps buffer from memory to a host-accessible pointer.
40  ----------------------------------------------------
41  // 3.  Decode
42  DECLARE start_clk AS CLOCK_T = CALL clock()
43    // Records the starting time for performance measurement.
44  FOR pos FROM 0 TO max_seq - 1
45    // 3-1.  Load the context input and decode the next token.
46    CALL CopyTensor1d(ctx_input, tok_emb_table[token])
47      // Copies the embedding of the current token into the
            input tensor.
48    CALL DecoderLayer(token, pos, ctx_input, ctx_k_cache,
        ctx_v_cache, ctx_final_norm, weights, q, add_kernel,
        mul_kernel, matmul_kernel, rmsnorm_kernel, rope_kernel,
        softmax_kernel, ptr2buffer, buffer)
49      // Executes the core decoding process for a single token
            at a given position with OpenCL kernels.
50    // 3-2.  Calculate the logits and softmax.
51    CALL MutmulVocab(ctx_logits, ctx_final_norm, tok_emb_table)
52      // Performs a matrix multiplication to calculate the raw
            scores (logits) for all possible next tokens.
53    SET next = Argmax(ctx_logits)
54      // Selects the token with the highest logit (greedy
            decoding).
55    SET token = next
56      // Sets the newly sampled token as the input for the
            next decoding step.
57  END FOR
58   // 4.  Print the time and speed.
```

```
59   DECLARE end_clk AS CLOCK_T = CALL clock()
60     // Records the ending time.
61   DECLARE decode_time AS DOUBLE = end_clk - start_clk
62   PRINT "Time : ", decode_time, "[s]"
63   // 5.  Flush OpenCL Buffer Memory
64   CALL OpenCLMacro(err, err = q.enqueueUnmapMemObject(buffer,
        ptr2buffer))
65   CALL OpenCLMacro(err, err = q.finish())
66   RETURN 0
67 END FUNCTION
```

文字生成のコアとなる DecoderLayer 関数の詳細な実装を、Listing4.2 に示す。表 4.5 に示すように、Transformer の Decoder 部分のアルゴリズムを基礎計算に分割し、必要に応じてアクセラレータカーネルを呼び出すことで、LLM の文字生成におけるハードウェアアクセラレーションを実現する。

Listing 4.2: DecoderLayer 関数のソースコード

```
1  #include <Everything you need>
2  FUNCTION DecoderLayer(token, pos, ctx_input, ctx_k_cache,
       ctx_v_cache, ctx_final_norm, weights, q, add_kernel,
       mul_kernel, matmul_kernel, rmsnorm_kernel, rope_kernel,
       softmax_kernel, ptr2buffer, buffer)
3    DECLARE static ctx AS Context
4    // Declare a static context object to hold intermediate
       activations.
5    DECLARE head_dim AS INTEGER = kDim / kNumLayers
6    // Calculate the dimension of each attention head.
7    DECLARE norm AS FLOAT = 1 / SQRT(head_dim)
8    // Calculate the normalization factor for scaling QK dot
       products.
9    DECLARE attn_input AS Tensor1d
10   // Declare a tensor to hold the input to the attention
       block.
11   FOR i_layer FROM 0 TO kNumLayers - 1
12   // Loop through each Transformer layer.
13     IF i_layer == 0 THEN
14       FOR idx FROM 0 TO kDim - 1
15         attn_input[idx] = ctx_input[idx]
16         // For the first layer, the attention input is the raw
             token embedding.
17       END FOR
18     ELSE
19       FOR idx FROM 0 TO kDim - 1
20         attn_input[idx] = ctx.ffn_res[i_layer - 1][idx]
21         // For subsequent layers, the attention input is the
             residual output from the previous layer's FFN.
22       END FOR
```

```
23      END IF
24      // --------- Attention ---------
25      // 1.  RMS Normalize
26      CALL RMSNormPL(ctx.attn_norm[i_layer], attn_input, w.rms_att_w
            [i_layer], q, rmsnorm_kernel, ptr2buffer, buffer)
27      // Using RMSNorm accelerator kernel.
28      // 2.  Linear transformations for Q, K, V
29      CALL MatmulPL(ctx.attn_wqx[i_layer], ctx.attn_norm[i_layer],
            w.attn_wq[i_layer], q, matmul_kernel, ptr2buffer, buffer)
30      CALL MatmulPL(ctx.attn_wkx[i_layer], ctx.attn_norm[i_layer],
            w.attn_wk[i_layer], q, matmul_kernel, ptr2buffer, buffer)
31      CALL MatmulPL(ctx.attn_wvx[i_layer], ctx.attn_norm[i_layer],
            w.attn_wv[i_layer], q, matmul_kernel, ptr2buffer, buffer)
32      // Computes the QKV by using matmul accelerator kernel.
33      // 3.  RoPE for each head
34      FOR head FROM 0 TO kNumHeads - 1
35      // Loop through each attention head.
36        CALL RoPEPL(ctx.attn_q_r[i_layer], ctx.attn_k_r[i_layer],
              ctx.attn_wqx[i_layer], ctx.attn_wkx[i_layer], w.
              cos_table[pos], w.sin_table[pos], head * head_dim,
              head_dim, q, rope_kernel, ptr2buffer, buffer)
37        // Applies RoPE to Q and k with RoPE accelerator kernel.
38      END FOR
39      // 4.  Key / Value Cache
40      CALL CopyTensor1d(ctx_k_cache[i_layer][pos], ctx.attn_k_r[
            i_layer])
41      CALL CopyTensor1d(ctx_v_cache[i_layer][pos], ctx.attn_wvx[
            i_layer])
42      // Stores the processed KV cache for future positions.
43      // 5.  Multi-Head Attention
44      FOR i_head FROM 0 TO kNumHeads - 1
45      // Loop through each attention head to compute attention.
46        DECLARE head_begin AS INTEGER = i_head * head_dim
47        // Calculate the starting index for the current head's
              slice.
48        DECLARE head_end AS INTEGER = (i_head + 1) * head_dim
49        // Calculate the ending index for the current head's
              slice.
50        // 5-1.  QK
51        CALL MutmulRanged(ctx.attn_qk[i_layer], ctx.attn_q_r[i_layer
              ], ctx_k_cache[i_layer], 0, pos, head_begin, head_end)
52        // Computes the dot product of query and cached keys for
              the current head within the attention layer.
53        // 5-2.  QK * 1/√ d
54        CALL MulPL(ctx.attn_qk[i_layer], ctx.attn_qk[i_layer], norm
              , q, mul_kernel, ptr2buffer, buffer)
55        // using Mul accelerator kernel.
```

```
56        // 5-3.  Softmax( QK/√ d )
57        CALL SoftmaxPL(ctx.attn_sm[i_layer], ctx.attn_qk[i_layer],
              pos + 1, q, softmax_kernel, ptr2buffer, buffer)
58        // Applies the softmax function by using softmax
              accelerator kernel.
59        // 5-4.  Softmax(QK/√ d) .  V
60        CALL MutmulRangedTranspose(ctx.attn_val[i_layer], ctx.
              attn_sm[i_layer], ctx_v_cache[i_layer], head_begin,
              head_end, 0, pos + 1)
61        // Computes the weighted sum of values using attention
              probabilities and cached values.
62      END FOR
63      // 6.  Output (Merge Heads)
64      CALL MatmulPL(ctx.attn_out[i_layer], ctx.attn_val[i_layer], w
            .attn_wo[i_layer], q, matmul_kernel, ptr2buffer, buffer)
65      // Linearly combines the outputs by using matmul
            accelerator kernel.
66      // 7.  Res connect
67      CALL AddPL(ctx.attn_res[i_layer], attn_input, ctx.attn_out[
            i_layer], q, add_kernel, ptr2buffer, buffer)
68      // Adds the attention output by using add accelerator
            kernel.
69      // --------- FFN ---------
70      // 1.  RMS Normalize
71      CALL RMSNormPL(ctx.ffn_norm[i_layer], ctx.attn_res[i_layer],
            w.rms_ffn_w[i_layer], q, rmsnorm_kernel, ptr2buffer,
            buffer)
72      // 2.  w1 .  x
73      CALL MatmulPL(ctx.ffn_w1x[i_layer], ctx.ffn_norm[i_layer], w.
            ffn_w1[i_layer], q, matmul_kernel, ptr2buffer, buffer)
74      // 3.  w3 .  x
75      CALL MatmulPL(ctx.ffn_w3x[i_layer], ctx.ffn_norm[i_layer], w.
            ffn_w3[i_layer], q, matmul_kernel, ptr2buffer, buffer)
76      // 4.  Swish( w1x )
77      CALL Swish(ctx.ffn_act[i_layer], ctx.ffn_w1x[i_layer])
78      // 5.  Swish(w1x) * w3x
79      CALL MulPL(ctx.ffn_dot[i_layer], ctx.ffn_act[i_layer], ctx.
            ffn_w3x[i_layer], q, mul_kernel, ptr2buffer, buffer)
80      // 6.  w2 .  Swish(w1x)*w3x
81      CALL MatmulPL(ctx.ffn_out[i_layer], ctx.ffn_dot[i_layer], w.
            ffn_w2[i_layer], q, matmul_kernel, ptr2buffer, buffer)
82      // 7.  Res connect
83      CALL AddPL(ctx.ffn_res[i_layer], ctx.attn_res[i_layer], ctx.
            ffn_out[i_layer], q, add_kernel, ptr2buffer, buffer)
84    END FOR
85    // -- Final RMS Normalize --
86    CALL RMSNormPL(ctx_final_norm, ctx.ffn_res[kNumLayers - 1], w.
          rms_final, q, rmsnorm_kernel, ptr2buffer, buffer)
```

```
87   RETURN
88 END FUNCTION
```

4.2.7 アクセラレータカーネル

4.2.4節で提案した基礎計算アクセラレータカーネル、すなわちadd_kernel、mul_kernel、matmul_kernel、rmsnorm_kernel、rope_kernel、およびsoftmax_kernelは、C++言語で記述される。これらのカーネルは、Vitisの高位合成（HLS）機能、及びVivadoの論理合成、配置配線機能を用いて、FPGAが認識可能なバイナリファイルにコンパイルされる。各カーネルのソースコードを以下に示す。

add_kernel ベクトル加算用カーネル：2つの浮動小数点ベクトルの要素ごとの加算を行う加算回路である。

Listing 4.3: add_kernelのHLSソースコード

```
1  stream_input_vector(float* input_array, hls::stream<float>&
       output_stream, int vector_length) {
2  memory_read:
3    for (int i = 0; i < vector_length; i++) {
4      output_stream << input_array[i];
5    }
6  }
7  add_vectors_streaming(hls::stream<float>& vector1_stream,hls::
       stream<float>& vector2_stream, hls::stream<float>&
       result_stream, int vector_length) {
8    for (int i = 0; i < vector_length; i++) {
9      result_stream << vector1_stream.read() + vector2_stream.read
           ();
10   }
11 }
12 write_result_to_memory(float* output_array, hls::stream<float>&
       input_stream,int vector_length) {
13 memory_write:
14   for (int i = 0; i < vector_length; i++) {
15     output_array[i] = input_stream.read();
16   }
17 }
18 extern "C" {
19  void vector_addition_kernel(float* input_vector_1, float*
        input_vector_2, float* output_vector, int vector_length) {
20 #pragma HLS INTERFACE m_axi port = input_vector_1 bundle =
       gmem0
21 #pragma HLS INTERFACE m_axi port = input_vector_2 bundle =
       gmem1
```

```
22 #pragma HLS INTERFACE m_axi port = output_vector bundle =
       gmem0
23    static hls::stream<float> vector1_stream("vector1_stream");
24    static hls::stream<float> vector2_stream("vector2_stream");
25    static hls::stream<float> addition_result_stream("
          addition_result_stream");
26 #pragma HLS dataflow
27    stream_input_vector(input_vector_1, vector1_stream,
          vector_length);
28    stream_input_vector(input_vector_2, vector2_stream,
          vector_length);
29    add_vectors_streaming(vector1_stream, vector2_stream,
          addition_result_stream, vector_length);
30    write_result_to_memory(output_vector, addition_result_stream,
          vector_length);
31  }
32 }
```

mul_kernel ベクトル乗算用カーネル：2 つの浮動小数点ベクトルの要素ごとの乗算を行い、結果を別のベクトルとして出力するストリーミングベースのベクトル乗算回路である。

Listing 4.4: mul_kernel の HLS ソースコード

```
1  stream_input_vector(float* input_array, hls::stream<float>&
       output_stream, int vector_length) {
2  memory_read:
3    for (int i = 0; i < vector_length; i++) {
4      output_stream << input_array[i];
5    }
6  }
7  multiply_vectors_streaming(hls::stream<float>& vector1_stream, hls
       ::stream<float>& vector2_stream, hls::stream<float>&
       result_stream, int vector_length) {
8    for (int i = 0; i < vector_length; i++) {
9      result_stream << vector1_stream.read() * vector2_stream.read
           ();
10   }
11 }
12 write_result_to_memory(float* output_array, hls::stream<float>&
       input_stream,int vector_length) {
13 memory_write:
14   for (int i = 0; i < vector_length; i++) {
15     output_array[i] = input_stream.read();
16   }
17 }
18 extern "C" {
```

```
19 void vector_multiplication_kernel(float* input_vector_1, float*
       input_vector_2, float* output_vector, int vector_length) {
20 #pragma HLS INTERFACE m_axi port = input_vector_1 bundle =
       gmem0
21 #pragma HLS INTERFACE m_axi port = input_vector_2 bundle =
       gmem1
22 #pragma HLS INTERFACE m_axi port = output_vector bundle =
       gmem0
23   static hls::stream<float> vector1_stream("vector1_stream");
24   static hls::stream<float> vector2_stream("vector2_stream");
25   static hls::stream<float> multiplication_result_stream("
         multiplication_result_stream");
26 #pragma HLS dataflow
27   stream_input_vector(input_vector_1, vector1_stream,
         vector_length);
28   stream_input_vector(input_vector_2, vector2_stream,
         vector_length);
29   multiply_vectors_streaming(vector1_stream, vector2_stream,
         multiplication_result_stream, vector_length);
30   write_result_to_memory(output_vector,
         multiplication_result_stream, vector_length);
31 }
32 }
```

matmul_kernel 行列乗算用カーネル：列優先形式の浮動小数点行列と浮動小数点ベクトルの積（$y = A \cdot x$）を出力するストリーミングベースの行列乗算回路である。

Listing 4.5: matmul_kernel の HLS ソースコード

```
1  stream_input_vector(float* input_vector, hls::stream<float>&
       output_stream, int vector_length) {
2  memory_read_vector:
3    for (int i = 0; i < vector_length; i++) {
4      output_stream << input_vector[i];
5    }
6  }
7  stream_input_matrix(float* input_matrix, hls::stream<float>&
       output_stream, int vector_length,int column_count) {
8  memory_read_matrix:
9    for (int i = 0; i < column_count; i++) {
10     for (int j = 0; j < vector_length; j++) {
11       output_stream << input_matrix[vector_length * i + j];
12     }
13   }
14 }
15 compute_matrix_vector_multiply(hls::stream<float>& vector_stream,
       hls::stream<float>& matrix_stream, hls::stream<float>&
```

```
    result_stream, int vector_length, int column_count) {
16  float vector_buffer[MAX_DATA_SIZE];
17  float dot_product_sum = 0;
18  // Load vector into local buffer
19  for (int i = 0; i < vector_length; i++) {
20    vector_buffer[i] = vector_stream.read();
21  }
22 matrix_multiply_execute:
23  for (int i = 0; i < column_count; i++) {
24    for (int j = 0; j < vector_length; j++) {
25 #pragma HLS UNROLL
26      dot_product_sum += vector_buffer[j] * matrix_stream.read();
27    }
28    result_stream << dot_product_sum;
29    dot_product_sum = 0;
30  }
31 }
32 write_result_to_memory(float* output_vector, hls::stream<float>&
    input_stream, int result_length) {
33 memory_write:
34  for (int i = 0; i < result_length; i++) {
35    output_vector[i] = input_stream.read();
36  }
37 }
38 extern "C" {
39 void matrix_vector_multiplication_kernel(float* input_vector,
    float* input_matrix, float* output_vector, int vector_length,
    int column_count) {
40 #pragma HLS INTERFACE m_axi port = input_vector bundle = gmem0
41 #pragma HLS INTERFACE m_axi port = input_matrix bundle = gmem1
42 #pragma HLS INTERFACE m_axi port = output_vector bundle =
    gmem0
43  static hls::stream<float> vector_data_stream("vector_data_stream
      ");
44  static hls::stream<float> matrix_data_stream("matrix_data_stream
      ");
45  static hls::stream<float> multiplication_result_stream("
      multiplication_result_stream");
46 #pragma HLS dataflow
47  stream_input_vector(input_vector, vector_data_stream,
      vector_length);
48  stream_input_matrix(input_matrix, matrix_data_stream,
      vector_length, column_count);
49  compute_matrix_vector_multiply(vector_data_stream,
      matrix_data_stream, multiplication_result_stream,
      vector_length, column_count);
```

```
50   write_result_to_memory(output_vector,
        multiplication_result_stream, column_count);
51 }
52 }
```

rmsnorm_kernel RMS 正規化用カーネル： 入力行列に対して、RMS 正規化処理を実行する回路である。

Listing 4.6: rmsnorm_kernel の HLS ソースコード

```
1  stream_input_vector(float* input_array, hls::stream<float>&
       output_stream, int vector_length) {
2  memory_read:
3    for (int i = 0; i < vector_length; i++) {
4      output_stream << input_array[i];
5    }
6  }
7  apply_rms_normalization(hls::stream<float>& input_vector_stream,
       hls::stream<float>& weight_vector_stream, hls::stream<float>&
       normalized_output_stream) {
8    float input_vector_buffer[288];
9    float weight_vector_buffer[288];
10   float sum_of_squares = 0;
11 // Load input vector into local buffer
12   for (int i = 0; i < 288; i++) {
13     input_vector_buffer[i] = input_vector_stream.read();
14   }
15 // Load weight vector into local buffer
16   for (int i = 0; i < 288; i++) {
17     weight_vector_buffer[i] = weight_vector_stream.read();
18   }
19 // Calculate sum of squares for RMS computation
20   for (int i = 0; i < 288; i++) {
21     sum_of_squares += input_vector_buffer[i] * input_vector_buffer
         [i];
22   }
23 // Compute RMS normalization factor
24   constexpr float epsilon = 1e-5;
25   const float rms_normalization_factor = 1 / std::sqrt(
       sum_of_squares / 288 + epsilon);
26
27 // Apply RMS normalization with weight scaling
28   for (size_t i = 0; i < 288; i++) {
29     normalized_output_stream << input_vector_buffer[i] *
         rms_normalization_factor * weight_vector_buffer[i];
30   }
31 }
```

```
32 write_result_to_memory(float* output_array, hls::stream<float>&
       input_stream, int vector_length) {
33 memory_write:
34   for (int i = 0; i < vector_length; i++) {
35     output_array[i] = input_stream.read();
36   }
37 }
38 extern "C" {
39 void rms_normalization_kernel(float* input_vector, float*
       weight_vector, float* output_vector, int vector_length) {
40 #pragma HLS INTERFACE m_axi port = input_vector bundle = gmem0
41 #pragma HLS INTERFACE m_axi port = weight_vector bundle =
       gmem1
42 #pragma HLS INTERFACE m_axi port = output_vector bundle =
       gmem0
43   static hls::stream<float> input_vector_stream("
         input_vector_stream");
44   static hls::stream<float> weight_vector_stream("
         weight_vector_stream");
45   static hls::stream<float> normalized_output_stream("
         normalized_output_stream");
46 #pragma HLS dataflow
47   stream_input_vector(input_vector, input_vector_stream,
         vector_length);
48   stream_input_vector(weight_vector, weight_vector_stream,
         vector_length);
49   apply_rms_normalization(input_vector_stream,
         weight_vector_stream, normalized_output_stream);
50   write_result_to_memory(output_vector, normalized_output_stream,
         vector_length);
51 }
52 }
```

rope_kernel RoPE 処理用カーネル：ベクトルの偶数・奇数次元に対して RoPE 変換を適用する処理回路である。

Listing 4.7: rope_kernel の HLS ソースコード

```
1 stream_input_vector(float* input_array, hls::stream<float>&
      output_stream, int vector_length) {
2 memory_read:
3   for (int i = 0; i < vector_length; i++) {
4     output_stream << input_array[i];
5   }
6 }
7 apply_rotary_position_embedding(hls::stream<float>&
      query_input_stream, hls::stream<float>& key_input_stream, hls
      ::stream<float>& cosine_values_stream, hls::stream<float>&
```

```
    sine_values_stream, hls::stream<float>& query_output_stream,
    hls::stream<float>& key_output_stream, int head_start_index) {
  float query_buffer[288];
  float key_buffer[288];
  float cosine_buffer[24];
  float sine_buffer[24];
  float rotated_query_buffer[288];
  float rotated_key_buffer[288];
// Load input vectors into local buffers
  for (int i = 0; i < 288; i++) {
    query_buffer[i] = query_input_stream.read();
    key_buffer[i] = key_input_stream.read();
  }
// Load rotation coefficients
  for (int i = 0; i < 24; i++) {
    cosine_buffer[i] = cosine_values_stream.read();
    sine_buffer[i] = sine_values_stream.read();
  }
// Apply rotary position embedding transformation
  for (int i = 0; i < 48; ++i) {
    int even_index = head_start_index + i * 2 + 0;
    int odd_index = head_start_index + i * 2 + 1;
    float query_even = query_buffer[even_index];
    float query_odd = query_buffer[odd_index];
    float key_even = key_buffer[even_index];
    float key_odd = key_buffer[odd_index];
    float cos_val = cosine_buffer[i];
    float sin_val = sine_buffer[i];
// Apply rotation matrix:  [cos -sin; sin cos]
    rotated_query_buffer[even_index] = query_even * cos_val -
        query_odd * sin_val;
    rotated_query_buffer[odd_index] = query_even * sin_val +
        query_odd * cos_val;
    rotated_key_buffer[even_index] = key_even * cos_val - key_odd
         * sin_val;
    rotated_key_buffer[odd_index] = key_even * sin_val + key_odd
        * cos_val;
  }
// Stream out the rotated vectors
  for (int i = 0; i < 288; i++) {
    query_output_stream << rotated_query_buffer[i];
    key_output_stream << rotated_key_buffer[i];
  }
}
write_result_to_memory(float* output_array, hls::stream<float>&
    input_stream, int vector_length) {
```

```
47 memory_write:
48   for (int i = 0; i < vector_length; i++) {
49     output_array[i] = input_stream.read();
50   }
51 }
52 extern "C" {
53 void rotary_position_embedding_kernel(float* query_input, float*
       key_input, float* cosine_values, float* sine_values, float*
       query_output, float* key_output, int head_start_index) {
54 #pragma HLS INTERFACE m_axi port = query_input bundle = gmem0
       max_widen_bitwidth = 32
55 #pragma HLS INTERFACE m_axi port = key_input bundle = gmem1
       max_widen_bitwidth = 32
56 #pragma HLS INTERFACE m_axi port = cosine_values bundle =
       gmem2 max_widen_bitwidth = 32
57 #pragma HLS INTERFACE m_axi port = sine_values bundle = gmem3
       max_widen_bitwidth = 32
58 #pragma HLS INTERFACE m_axi port = query_output bundle = gmem0
       max_widen_bitwidth = 32
59 #pragma HLS INTERFACE m_axi port = key_output bundle = gmem1
       max_widen_bitwidth = 32
60   static hls::stream<float> query_input_stream("query_input_stream
       ");
61   static hls::stream<float> key_input_stream("key_input_stream");
62   static hls::stream<float> cosine_values_stream("
       cosine_values_stream");
63   static hls::stream<float> sine_values_stream("sine_values_stream
       ");
64   static hls::stream<float> query_output_stream("
       query_output_stream");
65   static hls::stream<float> key_output_stream("key_output_stream")
       ;
66 #pragma HLS dataflow
67   stream_input_vector(query_input, query_input_stream, 288);
68   stream_input_vector(key_input, key_input_stream, 288);
69   stream_input_vector(cosine_values, cosine_values_stream, 24);
70   stream_input_vector(sine_values, sine_values_stream, 24);
71   apply_rotary_position_embedding(query_input_stream,
       key_input_stream, cosine_values_stream, sine_values_stream,
       query_output_stream, key_output_stream, head_start_index);
72   write_result_to_memory(query_output, query_output_stream, 288);
73   write_result_to_memory(key_output, key_output_stream, 288);
74 }
75 }
```

softmax_kernel softmax 関数用カーネル：入力行列に対して、softmax 関数処理を施す回路である。

Listing 4.8: softmax_kernel の HLS ソースコード

```
stream_input_vector(float* input_array, hls::stream<float>&
    output_stream, int vector_length) {
memory_read:
  for (int i = 0; i < vector_length; i++) {
    output_stream << input_array[i];
  }
}
apply_softmax_activation(hls::stream<float>& input_stream, hls::
    stream<float>& output_stream, int vector_length) {
  int actual_vector_length = vector_length;
  float input_vector_buffer[MAX_DATA_SIZE];
  float softmax_output_buffer[MAX_DATA_SIZE];
// Handle special case for dynamic vector size
  if (vector_length == -1) {
    actual_vector_length = 256;
  }
// Load input vector into local buffer
  for (int i = 0; i < actual_vector_length; i++) {
    input_vector_buffer[i] = input_stream.read();
  }
// Find maximum value for numerical stability
  float maximum_value = input_vector_buffer[0];
  for (int i = 1; i < actual_vector_length; i++) {
    if (input_vector_buffer[i] > maximum_value) {
      maximum_value = input_vector_buffer[i];
    }
  }
// Compute exponentials and sum
  float exponential_sum = 0;
  for (int i = 0; i < actual_vector_length; i++) {
    softmax_output_buffer[i] = std::exp(input_vector_buffer[i] -
        maximum_value);
    exponential_sum += softmax_output_buffer[i];
  }
// Normalize by dividing each exponential by the sum
  for (int i = 0; i < actual_vector_length; i++) {
    softmax_output_buffer[i] /= exponential_sum;
  }
// Stream out the softmax probabilities
  for (int i = 0; i < actual_vector_length; i++) {
    output_stream << softmax_output_buffer[i];
  }
}
write_result_to_memory(float* output_array, hls::stream<float>&
    input_stream, int vector_length) {
memory_write:
```

```
43   for (int i = 0; i < vector_length; i++) {
44     output_array[i] = input_stream.read();
45   }
46 }
47 extern "C" {
48 void softmax_activation_kernel(float* input_vector, float*
       output_vector, int vector_length) {
49 #pragma HLS INTERFACE m_axi port = input_vector bundle = gmem0
50 #pragma HLS INTERFACE m_axi port = output_vector bundle =
       gmem0
51   static hls::stream<float> input_vector_stream("
         input_vector_stream");
52   static hls::stream<float> softmax_output_stream("
         softmax_output_stream");
53 #pragma HLS dataflow
54   stream_input_vector(input_vector, input_vector_stream,
         vector_length);
55   apply_softmax_activation(input_vector_stream,
         softmax_output_stream, vector_length);
56   write_result_to_memory(output_vector, softmax_output_stream,
         vector_length);
57 }
58 }
```

第5章　実験評価

本章では、本研究の性能評価試験、ならびに最適化手法と最適化後の結果について述べる。まず5.1節と5.2節では、試験のセットアップ、初期実装の試験結果及び考察にについて説明する。5.3節と5.4節では、アクセレータカーネルの最適化とDecoderLayer関数に対する最適化について説明する。5.5節では、最適化後の結果と考察を示す。

5.1　性能評価試験

第4章で提案した手法をハードウェアプラットフォームに実装し、ハードウェアリソース使用量および文字生成所要時間の試験結果を収集する。さらに、各カーネルの性能を分析し、性能向上のための最適化を図る。最適化後のシステムを再度試験し、その最適化手法の妥当性と有効性を評価・考察する。

5.1.1　試験セットアップ

FPGAのハードウェアコンフィグレーション 4.2.7節のC言語ベースのソースコードを用い、Vitis IDEのHLSツールでビルドを行う。各アクセラレータカーネルが使用するハードウェアリソースを、表5.1にまとめる。6つのアクセラレータカーネルは1つのXCLBINファイルに統合される。このコンパイル済みのXCLBINファイルを、評価ボードのストレージ（MicroSDカード）内の指定された場所にコピーする。ホストアプリケーション実行後、XCLBINファイルを読み込み、PLのコンフィグレーションを行う。

表 5.1: ハードウェアリソースの使用量

Component	FF	LUT	DSP	BRAM	URAM
Interconnect	18082(7.72%)	9540(8.15%)	0	0	0
add_kernel	4663(1.99%)	4005(3.42%)	2(0.16%)	1(0.69%)	0
mul_kernel	3859(1.65%)	4499(3.84%)	3(0.24%)	1(0.69%)	0
matmul_kernel	4228(1.8%)	5146(4.39%)	12(0.96%)	2(1.39%)	0
rmsnorm_kernel	6357(2.71%)	5509(4.7%)	8(0.64%)	2(1.39%)	0
softmax_kernel	4936(2.11%)	5049(4.31%)	9(0.72%)	2(1.39%)	0
rope_kernel	9288(3.97%)	7478(6.38%)	32(2.56%)	7(4.86%)	0
Total Used	51413(21.95%)	41226(35.2%)	66(5.29%)	15(10.42%)	0

図 5.1 に FPGA における実装のレイアウトを示す。アクセラレータカーネルは色分けして表示した。

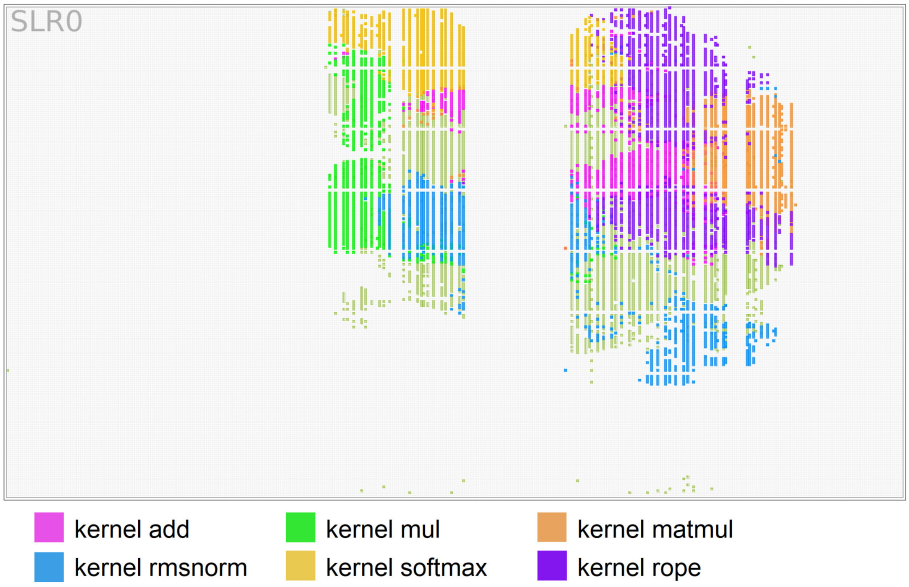

図 5.1: デバイスマップ

プログラム実行環境 本システムは KV260 に専用 OS である Petalinux を搭載し、組み込み Linux 環境に基づいてホストプログラムを実行する。AMD の KV260 専用ボードサポートパッケージ（BSP）ファイルと PetaLinux Tools を用いてシステムイメージをコンパイルし、MicroSD カードに書き込む。その後、実行に必要な XRT および OpenCL ライブラリをインストールすることで、実行環境が構築される。

評価項目 本システムの性能は、以下の項目を測定することで評価する。ホストプログラムに時間測定用のコードを追加し、32, 64, 128 及び 256 個のトークン生成

にかかるトータル時間と各アクセラレータカーネルの実行時間を計測する。トータル時間からアクセラレータカーネルの実行時間を差し引いた差分を ARM CPU の実行時間とする。

- 全体の文字生成速度
- 各アクセラレータカーネルの実行時間
- ARM CPU の実行時間

5.2 初期実装の結果と考察

前述のセットアップで初期バージョンの実装試験を行い、その結果を表 5.2 に示す。推論レイテンシは、システム仕様目標値（4.2 節参照）の約 1.5 倍であることが判明した。そのため、高速化に向けた性能最適化が必須である。

表 5.2: 初期実装の試験結果一覧表

Output Tokens	32	64	128	256
Total Time (ms)	24209.4	48453.8	97037.90	194871.00
PS Time (ms)	2460.334	4957.846	10057.925	20737.99
FPGA Time (ms)	21749.066	43495.954	86979.975	174133.01
Speed (Token/s)	1.3218	1.32085	1.31907	1.31369
Avg Latency (ms)	756.54375	757.090625	758.1085938	761.2148438

各アクセラレータカーネルの使用時間をより明確に分析するため、図 5.2 では各カーネルを個別にリストアップした。この分析から、matmul_kernel（行列乗算）が全体の時間の 80%以上を占めていることが明らかになった。したがって、まず matmul_kernel に対して最適化対策を講じる必要がある。

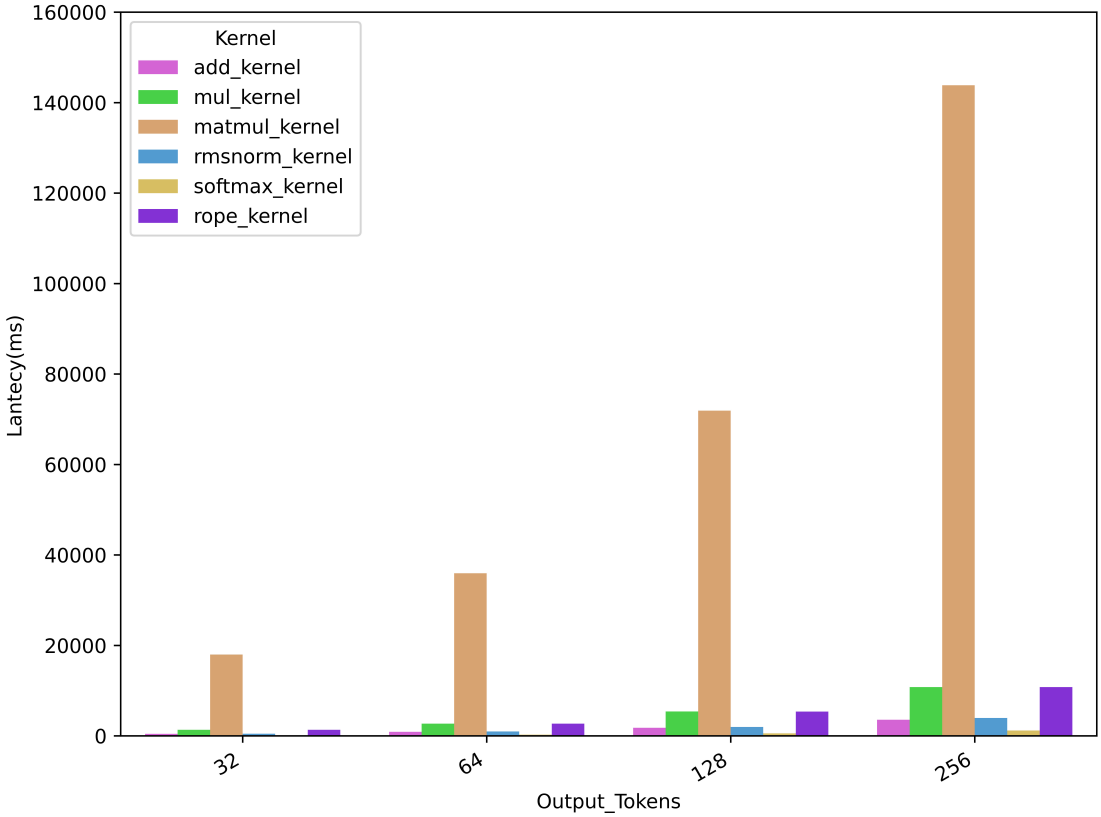

図 5.2: カーネル別の計算時間

matmul_kernel の HLS ソースコード（Listing4.5）を詳細に分析すると、このカーネルは表 5.3 に示す 4 つの関数で構成されている。

表 5.3: matmul_kernel の内部関数

関数名	タスクタイプ	備考
stream_input_vector	Data input	ベクトルを指すポインター からデータを読み込む
stream_input_matrix	Data input	マトリックスを指すポインター からデータを読み込む
compute_matrix_vector_multiply	Execution	ベクトルとマトリックス の行列乗算を計算する
write_result_to_memory	Data output	計算結果を出力する

初期実装では、表 5.3 に示す関数に対してプラグマ DATAFLOW を付与し、関数ごとの並列化を実行した。コア機能である compute_matrix_vector_multiply 関数の二重ループ（matrix_multiply_execute）の内層に対して、ループのアンローリング（UNROLL）を実施した。

matmul_kernel は汎用性を考慮し、異なるサイズのベクトルとマトリックスに対応できるように設計されている。そのため、二重ループ matrix_multiply_execute の実行回数が、Top 関数 matrix_vector_multiplication_kernel の引数 vector_length と column_count を介して外部から入力される。HLS ツールはループ回数をコンパイル時に確定できないため、ループの展開を行うことができず、UNROLL プラグマは無視される可能性がある。

5.3 アクセレータカーネルの最適化

ループ回数が外部引数として渡される前に確定できないという課題を解決するため、以下の 2 つの対策が考えられる。

1. 静的専用カーネルに変更 汎用 matmul_kernel をベクトルとマトリックスのサイズに合わせて静的なカーネルに変更し、専用カーネルを追加する。

2. パイプライン化 matmul_kernel 内部関数を全てパイプライン化することで、レイテンシ削減を図る。

5.3.1 静的専用カーネル

DecoderLayer 関数における行列乗算は、ベクトルとマトリックスのサイズによって 3 種類存在する。二重ループ matrix_multiply_execute のループ展開を目的として、以下の 3 種類の専用カーネルを用意する。

matmul_kernel_288·288： QKVO 値計算用

matmul_kernel_288·768： FFN_gate と FFN_up 重みとの乗算用

matmul_kernel_768·288： FFN_down 重みとの乗算用

表 5.1 から、FPGA リソースの約 65%割が未使用であることがわかるため、このアプローチは試す価値がある。初期実装のソースコードに対して、下記 Listing5.1（matmul_kernel_288·768 を例とする）のように、ソースコードの該当箇所をベクトルとマトリックスのサイズに修正して、再度 HLS を行う。

Listing 5.1: matmul_kernel_288·768 の HLS ソースコード

```
#define vector_length 288
#define column_count 768
stream_input_vector(float* input_vector, hls::stream<float>&
    output_stream, int
    vector_length) {
memory_read_vector:
  for (int i = 0; i < vector_length; i++) {
    output_stream << input_vector[i];
  }
}
stream_input_matrix(float* input_matrix, hls::stream<float>&
output_stream, int vector_length,int column_count) {
memory_read_matrix:
  for (int i = 0; i < column_count; i++) {
    for (int j = 0; j < vector_length; j++) {
      output_stream << input_matrix[vector_length * i + j];
    }
  }
}
compute_matrix_vector_multiply(hls::stream<float>& vector_stream,
    hls::stream<float>& matrix_stream, hls::stream<float>&
    result_stream, int vector_length, int
    column_count) {
  float vector_buffer[MAX_DATA_SIZE];
  float dot_product_sum = 0;
Load vector into local buffer
  for (int i = 0; i < vector_length; i++) {
    vector_buffer[i] = vector_stream.read();
  }
matrix_multiply_execute:
  for (int i = 0; i < column_count; i++) {
    for (int j = 0; j < vector_length; j++) {
#pragma HLS UNROLL
      dot_product_sum += vector_buffer[j] * matrix_stream.read();
    }
```

```
31      result_stream << dot_product_sum;
32      dot_product_sum = 0;
33    }
34  }
35  write_result_to_memory(float* output_vector, hls::stream<float>&
        input_stream, int
        result_length){
36  memory_write:
37    for (int i = 0; i < column_count; i++) {
38      output_vector[i] = input_stream.read();
39    }
40  }
41  extern "C" {
42  void matrix_vector_multiplication_kernel(float* input_vector,
        float* input_matrix, float* output_vector, int vector_length,
        int column_count) {
43  #pragma HLS INTERFACE m_axi port = input_vector bundle = gmem0
44  #pragma HLS INTERFACE m_axi port = input_matrix bundle = gmem1
45  #pragma HLS INTERFACE m_axi port = output_vector bundle =
        gmem0
46    static hls::stream<float> vector_data_stream("vector_data_stream
        ");
47    static hls::stream<float> matrix_data_stream("matrix_data_stream
        ");
48    static hls::stream<float> multiplication_result_stream("
        multiplication_result_stream");
49  #pragma HLS dataflow
50    stream_input_vector(input_vector, vector_data_stream,
        vector_length);
51    stream_input_matrix(input_matrix, matrix_data_stream,
        vector_length, column_count);
52    compute_matrix_vector_multiply(vector_data_stream,
        matrix_data_stream, multiplication_result_stream
53    , vector_length, column_count);
54    write_result_to_memory(output_vector,
        multiplication_result_stream,
        column_count);
55  }
56  }
```

matmul_kernel_288·288、matmul_kernel_288·768 及び matmul_kernel_768·288 のリソース使用量を表 5.4 に示す。静的カーネルに対して UNROLL を適用すると、ポインタが指すデータが一度オンチップメモリ（BRAM）に読み込まれて保存されるため、BRAM リソースが大幅に消費されることが判明した。3 つの静的専用カーネルの合計 BRAM 使用量が FPGA の容量を超過するため、この静的専用カーネルのアプローチは破綻した。

表 5.4: 静的専用カーネルのハードウェアリソース使用量

	FF	LUT	DSP	BRAM	URAM
matmul_kernel_288·288	18834	14546	5	118	0
matmul_kernel_288·768	18836	14550	5	118	0
matmul_kernel_768·288	8516	6812	5	118	0
Total	46186	35908	15	354	0
Available	234240	117120	1248	288	64
Utilization (%)	19.7	30.7	1.2	122.9	0

5.3.2 パイプライン化

前述 4 つのカーネル内部関数（表 5.3）は、データの読み込み、データの処理、データの出力という 3 つのパターンに分類できる。それぞれの内部関数中のループに対して PIPELINE プラグマを追記し、内部関数ごとにパイプライン化を行った。修正後のソースコードを Listing5.2 に示す。

Listing 5.2: パイプライン化カーネルの HLS ソースコード

```
1  stream_input_vector(float* input_vector, hls::stream<float>&
       output_stream, int vector_length) {
2  memory_read_vector:
3    for (int i = 0; i < vector_length; i++) {
4    #pragma HLS PIPELINE II=1
5      output_stream << input_vector[i];
6    }
7  }
8  stream_input_matrix(float* input_matrix, hls::stream<float>&
       output_stream, int vector_length,int column_count) {
9  memory_read_matrix:
10   for (int i = 0; i < column_count; i++) {
11     for (int j = 0; j < vector_length; j++) {
12     #pragma HLS PIPELINE II=1
13       output_stream << input_matrix[vector_length * i + j];
14     }
15   }
16 }
17 compute_matrix_vector_multiply(hls::stream<float>& vector_stream,
       hls::stream<float>& matrix_stream, hls::stream<float>&
       result_stream, int vector_length, int column_count) {
18   float vector_buffer[MAX_DATA_SIZE];
19 #pragma HLS ARRAY_PARTITION variable=vector_buffer cyclic
       factor=32
20   float dot_product_sum = 0;
21   // Load vector into local buffer
22   for (int i = 0; i < vector_length; i++) {
```

```
23    #pragma HLS PIPELINE II=1
24      vector_buffer[i] = vector_stream.read();
25    }
26  matrix_multiply_execute:
27    for (int i = 0; i < column_count; i++) {
28      for (int j = 0; j < vector_length; j++) {
29      #pragma HLS PIPELINE II=1
30        dot_product_sum += vector_buffer[j] * matrix_stream.read();
31      }
32      result_stream << dot_product_sum;
33      dot_product_sum = 0;
34    }
35  }
36  write_result_to_memory(float* output_vector, hls::stream<float>&
        input_stream, int result_length) {
37  memory_write:
38    for (int i = 0; i < result_length; i++) {
39    #pragma HLS PIPELINE II=1
40      output_vector[i] = input_stream.read();
41    }
42  }
43  extern "C" {
44  void matrix_vector_multiplication_kernel(float* input_vector,
        float* input_matrix, float* output_vector, int vector_length,
        int column_count) {
45  #pragma HLS INTERFACE m_axi port = input_vector bundle = gmem0
46  #pragma HLS INTERFACE m_axi port = input_matrix bundle = gmem1
47  #pragma HLS INTERFACE m_axi port = output_vector bundle =
        gmem0
48    static hls::stream<float> vector_data_stream("vector_data_stream
        ");
49    static hls::stream<float> matrix_data_stream("matrix_data_stream
        ");
50    static hls::stream<float> multiplication_result_stream("
        multiplication_result_stream");
51  #pragma HLS dataflow
52    stream_input_vector(input_vector, vector_data_stream,
        vector_length);
53    stream_input_matrix(input_matrix, matrix_data_stream,
        vector_length, column_count);
54    compute_matrix_vector_multiply(vector_data_stream,
        matrix_data_stream, multiplication_result_stream,
        vector_length, column_count);
55    write_result_to_memory(output_vector,
        multiplication_result_stream, column_count);
56  }
```

```
57 }
```

表 5.5: パイプライン化カーネルのハードウェアリソース使用量

	FF	LUT	DSP	BRAM	URAM
matmul_kernel_pipeline	4340	6676	12	40	0
Available	234240	117120	1248	288	64
Utilization (%)	1.9	5.7	1.0	13.9	0

パイプライン化した matmul_kernel_pipeline のリソース使用量を表 5.5 に示す。

5.4 DecoderLayer 関数の最適化

Transformer の Decoder における計算は、多くが逐次的であり、現在の入力データは以前の計算結果に依存する。ただし、Attention 層における QKV 値の計算では、正規化された入力データと QKV それぞれの重みとの行列乗算を行う。QKV の 3 つの計算は相互に独立しているため、QKV の並列計算を試み、レイテンシ削減を目指す。

初期実装の DecoderLayer 関数（Listing4.2）における QKV 値は、MatmulPL 関数で順次計算される。MatmulPL は OpenCL のタスクを生成し、アクセラレータカーネル matmul_kernel を利用して計算を実行する。計算完了後、バッファ内の結果を回収し、その戻り値を返す。

MatmulPL 関数を並列計算バージョンに書き換え、QKV それぞれのアクセラレータカーネルとして matmul_kernel_Q、matmul_kernel_K、および matmul_kernel_V を追加するというアプローチが考えられる。QKV 値以外の行列乗算は matmul_kernel_Q を使う。修正ソースコードを Listing5.3 に示す。

Listing 5.3: DecoderLayer 関数の修正ソースコード

```
1 #include <Everything you need>
2 FUNCTION DecoderLayer(token, pos, ctx_input, ctx_k_cache,
      ctx_v_cache, ctx_final_norm, weights, q, add_kernel,
      mul_kernel, matmul_kernel_Q, matmul_kernel_K,
      matmul_kernel_V, rmsnorm_kernel, rope_kernel, softmax_kernel,
      ptr2buffer, buffer, ptr2buffer_Q, buffer_Q, ptr2buffer_K,
      buffer_K, ptr2buffer_V, buffer_V)
3           ⋮
4           ⋮
5
6     // 2.  Linear transformations for Q, K, V
7     CALL MatmulPL(ctx.attn_wqx[i_layer],
         ctx.attn_norm[i_layer], w.attn_wq[i_layer], q,
         matmul_kernel, ptr2buffer, buffer)
```

```
8       CALL MatmulPL(ctx.attn_wkx[i_layer],
            ctx.attn_norm[i_layer], w.attn_wk[i_layer], q,
            matmul_kernel, ptr2buffer, buffer)
9       CALL MatmulPL(ctx.attn_wvx[i_layer],
            ctx.attn_norm[i_layer], w.attn_wv[i_layer], q,
            matmul_kernel, ptr2buffer, buffer)
10      CALL MatmulPL_QKV(ctx.attn_wqx[i_layer], ctx.attn_norm[
            i_layer], w.attn_wq[i_layer], q, matmul_kernel_Q,
            matmul_kernel_K, matmul_kernel_V, ptr2buffer_Q,
            buffer_Q, ptr2buffer_K, buffer_K, ptr2buffer_V,
            buffer_V)
11
12              ⋮
13              ⋮
14    RETURN
15  END FUNCTION
```

5.5 最適化後の結果

5.4.2 節のパイプライン化提案も盛り込み、初期実装の matmul_kernel を matmul_kernel_Q, matmul_kernel_K 及び matmul_kernel_V に変更する。ホストアプリケーションを Listing 5.3 のように修正し、再度ビルドする。最適化後のアクセラレータカーネルのリソース使用量とデバイスマップを、表 5.6 と図 5.3 に示す。

表 5.6: 最適化後のハードウェアリソースの使用量

Component	FF	LUT	DSP	BRAM	URAM
Interconnect	21200(9.05%)	13539(11.56%)	0(0%)	58(40.28%)	0
add_kernel	4663(1.99%)	4005(3.42%)	2(0.16%)	1(0.69%)	0
mul_kernel	4499(1.92%)	3859(3.29%)	3(0.24%)	1(0.69%)	0
matmul_kernel_Q	6183(2.64%)	5833(4.98%)	10(0.8%)	1(0.69%)	0
matmul_kernel_K	6183(2.64%)	5833(4.98%)	10(0.8%)	1(0.69%)	0
matmul_kernel_V	6183(2.64%)	5833(4.98%)	10(0.8%)	1(0.69%)	0
rmsnorm_kernel	6357(2.71%)	5509(4.7%)	8(0.64%)	2(1.39%)	0
softmax_kernel	4936(2.11%)	5049(4.31%)	9(0.72%)	2(1.39%)	0
rope_kernel	9288(3.97%)	7478(6.38%)	32(2.56%)	7(4.86%)	0
Total Used	69492(29.67%)	56938(48.62%)	84(6.73%)	74(51.39%)	0

matmul_kernel_K と matmul_kernel_V の 2 つのカーネルが新規追加されるため、リソース使用量が増加し、デバイスの総リソースのおおよそ半分程度を占めるようになる。

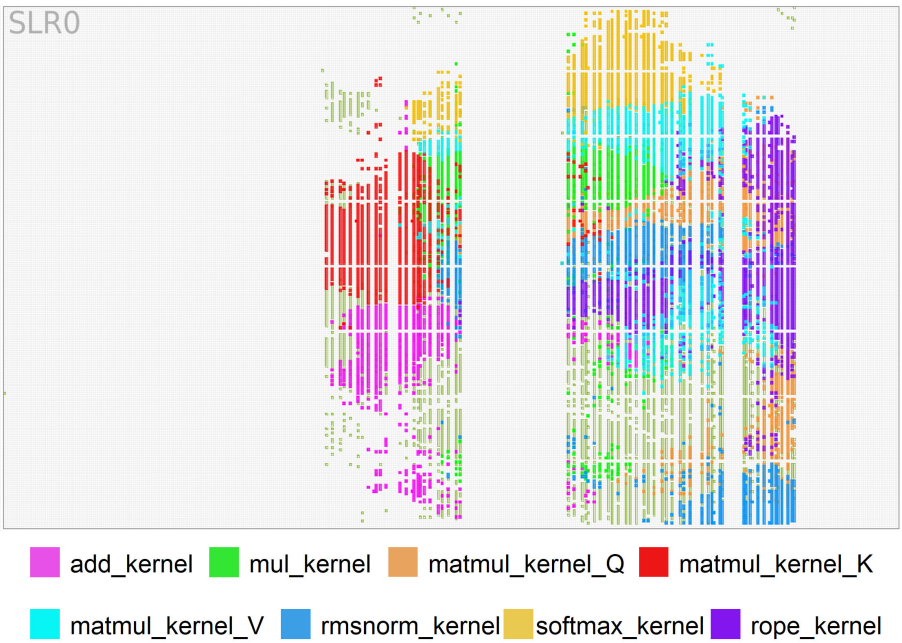

図 5.3: デバイスマップ

最適化後のホストアプリケーションとアクセラレータカーネルを KV260 に上書きし、評価項目を再度実施した。その結果を表 5.7 に示す。

表 5.7: 最適化後の試験結果一覧表

Output Tokens	32	64	128	256
Total Time (ms)	13718.1	27477	55182.50	110873.00
PS Time (ms)	1965.838	3971.059	8080.307	16792.74
FPGA Time (ms)	11752.262	23505.941	47102.193	94080.26
Speed (Token/s)	2.33268	2.32922	2.31958	2.30895
Avg Latency (ms)	428.690625	429.328125	431.1132813	433.0976563

最適化対策を実行したことで、Token 生成速度が 76.5%上昇し、平均遅延が 43.3%削減された。これにより、当初のコア性能仕様で要求されている推論速度を満たす結果となった。

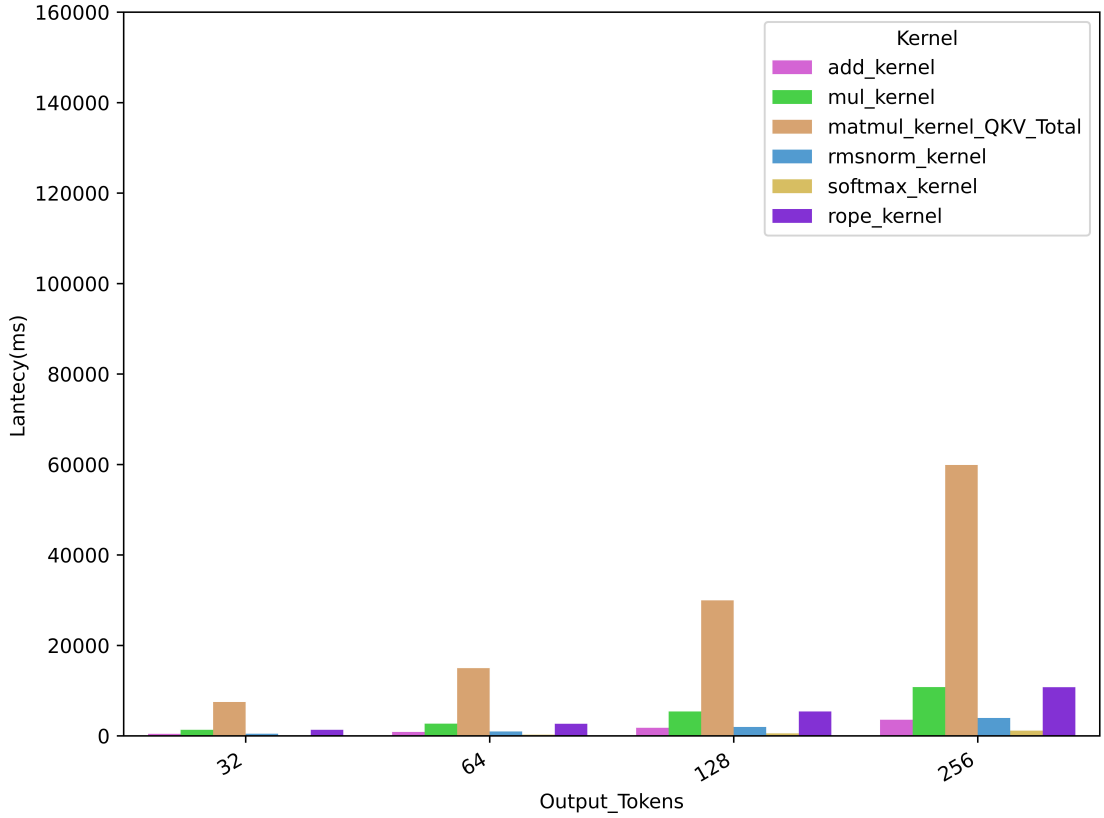

図 5.4: 最適化後のカーネル別の計算時間

図 5.4 に最適化後のカーネル別の計算時間を示す。初期実装の matmul_kernel と比較して、行列乗算の時間が 58.4%削減された。このことから、matmul_kernel のパイプライン化と DecoderLayer 関数における QKV 計算の並列化が有効であったことが示された。

5.6 まとめ

本章では、提案手法の性能評価と最適化結果について詳細に述べた。まず、初期実装では行列乗算がシステム全体の処理時間の大部分を占めており、これが性能ボトルネックであることが明らかとなった。これに対し、matmul_kernel の最適化として 2 つの手法を検討した。

1 つ目の静的専用カーネル化アプローチは、ループ展開による高速化を意図していたが、オンチップメモリ（BRAM）の使用量が FPGA の制約を超過し、実用的でないことが判明した。

2 つ目のパイプライン化アプローチでは、各関数の内部ループに対して #pragma HLS PIPELINE を適用することで、データ転送と演算処理のスループットを向上さ

せた。さらに、DecoderLayer 関数における QKV 計算を並列化し、matmul_kernel を Q/K/V に分割することで並列実行を可能にし、アクセラレータ全体の効率を高めた。

最適化後の試験結果では、トークン生成速度が最大 76.5%向上し、平均レイテンシは 43.3%削減された。これにより、当初設定したシステム性能目標を達成し、組込み FPGA 上での LLM 推論処理の高速化が有効であることが実証された。

第6章　おわりに

6.1　本研究の成果と結論

本研究では、エッジデバイス上で大規模言語モデル（LLM）を効率的に推論可能とするため、FPGA 上への LLM の実装および最適化手法について総合的に検討・評価を行った。主要な成果は以下の通りである。

モデルの選定と実装： パラメータ数 15M の軽量モデル（tinyllamas）を選定し、その構造を詳細に解析した上で、FPGA 実装に適した重み構造と演算フローを明確化した。

FPGA SoC プラットフォームへの適応： Zynq UltraScale+ MPSoC を対象とし、ARM CPU による制御と FPGA ロジックによる演算の役割分担を明確にし、OpenCL および HLS を活用した効率的な開発フローを確立した。

アクセラレータ設計と最適化： 加算、乗算、行列乗算、正規化処理、Softmax 処理及び RoPE 処理をアクセラレータとして分離し、逐次処理と並列化のバランスを考慮したカーネルを構築した。

性能評価： 実機評価において、設定したコア要件（応答時間 1 秒以下、コンテキスト長 200 文字以上、推論速度 2 語/秒）を満たす実装例を達成し、エッジ LLM として実用的なレベルでの動作を確認した。

以上の成果により、FPGA を活用した軽量 LLM 推論がエッジ AI 応用における有効な選択肢となることを実証した。特に、従来クラウド依存であった生成 AI 技術をスタンドアロンかつ低遅延で実装する可能性を示した。

6.2　今後の課題

FPGA による LLM 推論の実現性を確認できた一方で、いくつかの技術的・実用的な課題も明らかとなった。モデルとアクセラレータカーネルの現状は FP32 を基準としているが、さらなる省メモリ化および演算効率の向上を図るには、INT8 や混合精度（mixed precision）推論の導入が有効である。これにより、メモリ帯域の削減と消費電力の抑制が期待できる。モデル演算を ARM CPU と 6 つのアクセラ

レータカーネルに分散させている。デコーダ層を一つのアクセラレータカーネルに統合し、内部サブ関数に対して高粒度な並列化やパイプライン最適化を施すことで、オンチップメモリと外部 DDR4 メモリ間の I/O によるオーバーヘッドを削減し、さらにレイテンシを改善したい。

また、本研究は実装と最適化に注力するため、モデルトレーニングのプロセスを省略し、tinyllamas のような特定の小型モデルに限定した。エッジデバイス向けに特定のタスクへ絞り込み、関連性の高い生テキストデータを用いて専用モデルをトレーニングすれば、より高いユーザー体験をもたらすシステムが実現できるだろう。

これらの課題を踏まえ、今後は「より高性能かつ汎用的な LLM の FPGA 実装」に向けて研究を発展させていく予定である。FPGA の特性を最大限に活かしながら、社会実装可能な生成 AI ソリューションの実現を目指す。

付 録A　Petalinux環境の構築方法

PetaLinux は、SoC FPGA を開発するためのエンベデッド Linux ソフトウェア開発キット（SDK）である。本付録では、KV260 における PetaLinux 環境の構築方法について説明する。

A.1　事前準備

本付録は、以下の事前準備を実施した上で、PetaLinux 環境の構築方法を展開する。

- 作業用 PC の OS は Ubuntu 22.04.5 LTS であること。
- PetaLinux をコンパイルするためのソフトウェア「PetaLinux Tools」が事前にインストール済みであること。
- KV260 の BSP ファイルが事前にダウンロード済みであること。

A.2　ビルド

SD カード用のイメージファイルをビルドする手順は以下の通りである。

- Ubuntu の Terminal で KV260 の BSP ファイル（xilinx-kv260-starterkit-v2023.2-10140544.bsp）を用いて新しいプロジェクトを作成する。
- 新規作成したディレクトリへ移動する。
- Petalinux Tool を用いて Petalinux カーネルをビルドする。
- ビルド成果物をイメージファイルとしてパッケージ化する。

上記手順のコマンドを ListingA.1 に示す。

Listing A.1: ビルドのコマンド

```
petalinux-create -t project -s ../xilinx-kv260-starterkit-v2023.2-10140544.bsp
cd xilinx-kv260-starterkit-2023.2
petalinux-build
petalinux-package --boot --u-boot --force
petalinux-package --wic --images-dir images/linux/ --bootfiles "ramdisk.cpio.gz.u-boot,boot.scr,Image,system.dtb,system-zynqmp-sck-kv-g-revB.dtb"
```

A.3 SD カードに書き込む

前述の手順でビルドした WIC イメージファイルを KV260 評価ボードの SD カードに書き込む必要がある。balenaEtcher のようなツールを用いて、WIC イメージファイルを SD カードへ書き込む。

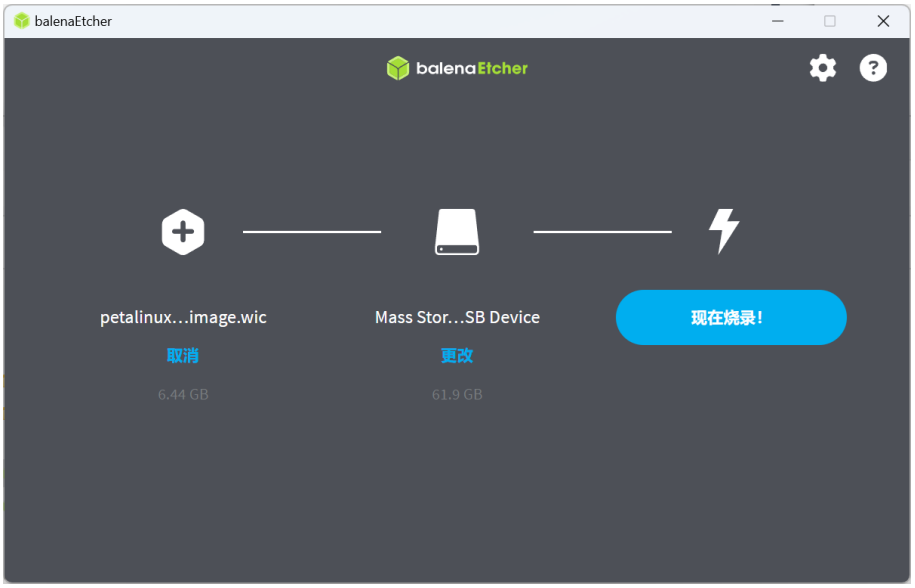

図 A.1: 書き込みのイメージ図

A.4 必要なライブラリのインストール

Petalinux イメージを書き込んだ SD カードを KV260 に挿入し、電源を投入する。Petalinux にログインし、ホストアプリケーション実行に必要なライブラリ（ZOCL と XRT）を手動でインストールする。ListingA.2 に示すコマンドを実行することで、PetaLinux 環境の構築が完了する。

Listing A.2: ライブラリのインストールコマンド

```
sudo dnf install zocl-202210.2.13.479-r0.0.xilinx_k26_kv
sudo dnf install xrt-202210.2.13.479-r0.0.cortexa72_cortexa53
```

謝辞

本研究を遂行するにあたり、多大なるご指導とご助言を賜りました指導教員・田中清史教授に、心より御礼申し上げます。研究計画の立案からレポートの完成に至るまで、技術的・学術的な面で丁寧かつ的確なご指導をいただき、本研究を無事にまとめることができました。

また、井口寧教授の並列処理の授業および、Natthawut Kertkeidkachorn 先生による副テーマのご指導を通じて、貴重な知見を得ることができました。これらは本研究において大いに活用させていただきました。

日々の研究活動を支えてくださった研究室の皆様にも、深く感謝申し上げます。活発な議論と温かな励ましは、研究を前向きに進めるうえで大きな支えとなりました。

さらに、本研究において参考とさせていただいたオープンソースのプロジェクトやハードウェア設計事例を通じて、貴重な情報と知見をご提供くださった関係企業や開発コミュニティの皆様にも、この場を借りて厚く御礼申し上げます。

最後に、日頃より温かく見守り、支えてくれた家族に心より感謝いたします。本研究は、多くの方々のご支援とご協力のもとに成り立っており、ここに深く感謝の意を表します。

参考文献

[1] Hong, Seongmin, et al. "Dfx: A low-latency multi-fpga appliance for accelerating transformer-based text generation." 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2022.

[2] Zeng, Shulin, et al. "Flightllm: Efficient large language model inference with a complete mapping flow on fpgas." Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2024.

[3] TURING INC., "A Lightweight Language Model Execution Environment Using FPGA", https://github.com/turingmotors/swan

[4] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[5] IBM, "LLM（大規模言語モデル）とは", https://www.ibm.com/jp-ja/think/topics/large-language-models

[6] Crockett, Louise H., et al. The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC. Strathclyde Academic Media, 2014.

[7] The Khronos Group Inc,"OpenCL Overview", https://www.khronos.org/opencl/

[8] AMD Xilinx, "Vitis High-Level Synthesis User Guide (UG1399)", https://docs.amd.com/r/en-US/ug1399-vitis-hls/

[9] Jakob Nielsen, "Response Times: The 3 Important Limits", https://www.nngroup.com/articles/response-times-3-important-limits/

[10] 竹内和広, et al. 『日本語話し言葉コーパス』の談話境界情報について. https://clrd.ninjal.ac.jp/csj/manu-f/discourse.pdf, 2006

[11] Hugging Face HF team,"Open LLM Leaderboard",https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard#/

[12] Eldan, Ronen, and Yuanzhi Li. "Tinystories: How small can language models be and still speak coherent english?." arXiv preprint arXiv:2305.07759 (2023).

[13] Andrej Karpathy, "tiny llamas", https://huggingface.co/karpathy/tinyllamas

[14] roneneldan,"TinyStories Datasets", https://huggingface.co/datasets/roneneldan/TinyStories

[15] AMD Xilinx, " Zynq UltraScale+ MPSoC Software Developer Guide ", https://docs.amd.com/r/2024.2-English/ug1137-zynq-ultrascale-mpsoc-swdev