FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Hardware Accelerated Edge Inference of Small Language Models on FPGA

João Pedro Reis Teixeira

Masters in Informatics and Computing Engineering

Supervisor: Nuno Miguel Cardanha Paulino, PhD

Second Supervisor: Francisco Manuel Ribeiro

October 3, 2025

Hardware Accelerated Edge Inference of Small Language Models on FPGA

João Pedro Reis Teixeira

Masters in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. Pedro Diniz
Referee: Prof. Mónica Figueiredo

October 3, 2025

Resumo

Os Large Language Models (LLMs) emergiram como ferramentas transformadoras em inúmeros domínios, no entanto, a sua implementação está predominantemente confinada a servidores na nuvem, potentes e com um consumo energético elevado. Esta dependência de infraestruturas centralizadas origina desafios significativos relacionados com a latência, a privacidade dos dados e o consumo de energia. A complexidade, a natureza sequencial e as exigências computacionais do processo de inferência colocam um desafio de como executar estes modelos de forma eficiente em dispositivos de edge com recursos limitados, para utilização em aplicações em tempo real, como sistemas autónomos e redes 5G/6G avançadas.

É necessário executar a inferência de Small Language Models (SLMs) localmente em dispositivos de edge de uma forma energeticamente eficiente e com bom desempenho. Tal eliminaria a latência, garantiria a privacidade dos dados e possibilitaria uma nova classe de aplicações de Inteligência Artificial em tempo real sem depender de uma ligação constante à nuvem, proporcionando capacidades valiosas a uma nova geração de dispositivos inteligentes.

O principal objetivo deste trabalho é estudar técnicas para acelerar a inferência de SLMs em Field Programmable Gate Arrays (FPGAs). Este trabalho testa a hipótese de que delegar as partes computacionalmente intensivas do processo de inferência para um acelerador de hardware personalizado é uma abordagem adequada, e que este codesign hardware-software pode resultar numa solução com maior eficiência energética e desempenho em comparação com a execução puramente baseada em software num processador embebido. Para este fim, é proposta uma solução, um acelerador de hardware para toda o forward pass do transformer, concebido com recurso a High-Level-Synthesis (HLS).

Para validação, é especificado um procedimento experimental, que envolve um protótipo funcional desenvolvido para a placa PYNQ-Z2, uma plataforma System-on-Chip de baixo custo. São realizados testes com o modelo TinyStories de 260k parâmetros para avaliar o desempenho da solução proposta em relação a uma base de referência apenas de software, a correr no processador ARM integrado da placa. A avaliação envolve projetos de hardware iterativos, incluindo versões de base, otimizadas para memória e quantizadas, para medir o *throughput* e identificar os estrangulamentos da implementação.

Os resultados mostram que o acelerador de hardware não conseguiu superar o desempenho da execução apenas em software no mesmo dispositivo, invalidando, assim, a hipótese proposta para esta plataforma de hardware específica. O resultado de desempenho negativo é uma descoberta valiosa, sublinhando o impacto significativo das limitações de recursos de hardware no paralelismo alcançável. No entanto, o trabalho contribui para a área ao identificar as principais armadilhas da implementação e delineia um caminho claro para investigação futura, incluindo portar o design para FPGAs mais capazes e redesenhar o *datapath* para ser totalmente baseado em números inteiros de 8 bits.

Palavras-chave: Large Language Models, Small Language Models, IA em Edge, FPGA, Aceleração em Hardware, High-Level Synthesis, Computação Energeticamente Eficiente.

Abstract

Large Language Models (LLMs) have emerged as transformative tools across numerous domains, yet their deployment is predominantly confined to powerful, energy-intensive cloud-based servers. This reliance on centralized infrastructure is the home of significant challenges related to latency, data privacy, and energy consumption. The complexity, sequential nature, and computational demands of the inference process pose a significant challenge: how to execute these models efficiently on resource-constrained edge devices for use in real-time applications like autonomous systems and advanced 5G/6G networks.

It is necessary to run Small Language Models (SLMs) inference locally on edge devices in an energy-efficient and performant way. This would eliminate latency, ensure data privacy, and enable a new class of real-time Artificial Intelligence (AI) applications without relying on constant cloud connectivity, providing valuable capabilities to a new generation of intelligent devices.

The main goal of this work is to study techniques for accelerating SLMs inference on Field Programmable Gate Arrays (FPGAs). This work tests the hypothesis that offloading the computationally intensive parts of the inference process to a custom hardware accelerator is a well-suited approach, and that this hardware-software co-design can result in a more energy-efficient and high-performance solution compared to purely software-based execution on an embedded processor. A solution, a hardware accelerator for the entire transformer forward pass, is proposed for this purpose, designed using High-Level-Synthesis (HLS).

For validation, an experimental procedure is specified, involving a functional prototype developed for the PYNQ-Z2 board, a low-cost System-on-Chip platform. Tests with the TinyStories 260k parameter model are conducted to assess the proposed solution's performance against a software-only baseline running on the board's integrated ARM processor. The evaluation involves iterative hardware designs, including baseline, memory-optimized, and quantized versions, to measure throughput and identify implementation bottlenecks.

The results show that the hardware accelerator could not outperform the software-only execution on the same device, therefore invalidating the proposed hypothesis for this specific hardware platform. The negative performance result is a valuable finding, underscoring the significant impact of hardware resource limitations on achievable parallelism. However, the work contributes to the field by identifying key implementation pitfalls and outlines a clear path for future research, including porting the design to more capable FPGAs and redesigning the datapath to be fully integer-based.

Keywords: Large Language Models, Small Language Models, Edge AI, FPGA, Hardware Acceleration, High-Level Synthesis, Energy-Efficient Computing.

Acknowledgements

Throughout my journey as a student, I was fortunate to have incredible support. The encouragement I received during challenging moments kept me going, and the opportunities to step outside my comfort zone were crucial for my growth as a person and a professional.

Firstly, my sincere thanks go to my supervisor, Nuno Miguel Cardanha Paulino, PhD, and my second supervisor, Francisco Manuel Ribeiro, for their close guidance and support throughout this work. Our weekly meetings provided essential feedback, direction, and motivation. I am especially grateful for their support during difficult moments and their consistent ability to help me find a new approach when I was stuck.

I want to offer my heartfelt thanks to my girlfriend, Joana Ferreira. Her love, patience, and encouragement were a constant source of comfort and strength. She made the difficult moments manageable, and this entire journey brighter.

To my parents and my sister, I offer my deepest gratitude. Thank you not only for your unwavering support but also for your honesty in keeping me grounded. Your sacrifices created the opportunities that led me here, and so much of who I have become is a reflection of your love and guidance.

A huge thank you goes to my friends, who provided a vital escape from the stress of this work. The moments of fun, laughter, and relaxation we shared were essential for keeping me grounded and served as a welcome reminder of life beyond this project. Thank you for your unwavering support.

This journey, and the incredible people who supported me through it, have taught me invaluable lessons. I have learned the importance of hard work and dedication, and of persevering through even the most overwhelming adversity. Above all, I've learned that strength is not found in facing challenges alone, but in the courage to ask for help and collaborate with others. It is in overcoming life's difficulties that we learn and shape ourselves as a person.

João Teixeira

"Failure is simply the opportunity to begin again,
this time more intelligently."

Henry Ford

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuits
BFP	Block Floating Point
BRAM	Block Random Access Memory
BSP	Board Support Package
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DPU	Data Processing Unit
DSP	Digital Signal Processor
FF	Flip-Flop
FFNN	Feedforward Neural Network
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GQMV	Group-wise Quantized Matrix-Vector Multiplication
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High-Level Synthesis
IoT	Internet of Things
LFSR	Linear Feedback Shift Register
LLM	Large Language Model
LUT	Look Up Table
MatMul	Matrix Multiplication
MPE	Matrix Processing Engine
MPU	Matrix Processing Unit
NLP	Natural Language Processing
OS	Operating System
PL	Programmable Logic
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
PS	Processing System
PYNQ	Python Productivity for Zynq
QKV	Query-Key-Value
RAM	Random Access Memory
RLHF	Reinforcement Learning from Human Feedback
RoPE	Rotary Positional Encoding
RNN	Recurrent Neural Network
RTL	Register-Transfer Level
SBVP	Super-Block Vector Processor
SFU	Special Function Unit
SIMD	Single Instruction, Multiple Data
SLM	Small Language Model
SoC	System-on-Chip
TPU	Tensor Processing Unit
VFU	Vector Function Unit

Chapter 1

Introduction

1.1 Context

Large Language Models (LLMs) have been widely used in recent years. Ever since ChatGPT was released in 2022, LLMs became a very popular tool for a wide public for general use, and is now being used in areas such as healthcare, education, and programming. Despite this, the energy expenditure of these kinds of technologies is extremely high. This is due to the fact that these models, especially the largest ones, run in very large systems with thousands of Graphics Processing Units (GPUs) executing tasks in parallel and require substantial memory resources to maintain very large conversation contexts [22].

These LLMs need to be executed in cloud-based servers, due to their high demand for computational power. This presents certain challenges, as cloud computing restrains the usage of LLMs in, for example, real-time applications, due to its limitations in meeting latency, bandwidth, and privacy requirements [19].

The architecture underlying LLMs is called a transformer model, which is split into two phases: prefill and decode. The prefill phase is easily parallelizable, as all the input tokens are received at once and can be processed in parallel, making its execution suitable for GPUs. In the decode phase, output tokens are generated sequentially, as each new token depends on the previously generated ones, meaning this is a sequential, memory-bound task. This sequential nature creates a bottleneck in the process and leads to underutilization of the hardware during inference, slowing down performance. This high energy and computational power demand means that LLM execution, especially for larger models, is generally limited to cloud-based servers, preventing this technology from being expanded to other devices such as smartphones or tablets, which do not have the capability to meet such high demands. 5G/6G applications also face the same issue, with the high latency of cloud computing being an obstacle to the usage of LLMs on communication nodes closer to the edge. 5G/6G networks have a very low latency and higher data rates and bandwidth requirements, which can not be met if any computing needs to be offloaded to a cloud-based service.

Technology companies have already identified this issue and have launched devices such as

NVIDIA's Jetson, Google's Edge TPU, and Apple's Neural Engine [24, 13, 16]. The problem with these solutions is that they have a low computational efficiency, poor memory bandwidth utilization, and significant compilation overheads. On top of that, they are designed with a specific task in mind. This limits their flexibility, as they do not have significant hardware reconfigurability capabilities.

1.2 Motivation

5G and 6G applications, coupled with LLMs, have the potential to revolutionize various fields, such as healthcare, autonomous driving, and smart cities [5]. By ensuring that the inference step is executed locally and not in the cloud, the problem of high latency would be solved, and this would open the door for LLMs to enter into the field of real-time applications at the edge. For example, in autonomous vehicles, LLMs, together with a 5G/6G application, can be used to process data from their sensors faster and also to help create collaborative systems between various vehicles, as it would greatly ease communication between them [32]. In smart cities, they can process data from various Internet of Things (IoT) devices faster, allowing for more efficient systems [1]. In healthcare, they can be used to improve remote care, possibly even with image and video processing to give patients a real-time diagnosis [28]. Essentially, LLMs would allow for a better analysis and interpretation of the data that is received by real-time applications, whether in the form of text, images, or video. However, despite their promise in these applications, LLMs still face limitations in these local or near-local contexts, due to the resources required to train and execute them. Researchers are well-aware of these limitations, and efforts to discover new solutions are being made, but as LLMs is a recent and rapidly expanding field, there is not yet a consensus on the most viable approach, but the need for a new one is evident.

This dissertation aims to fulfill this need by exploring a new approach to LLM execution using FPGAs, that promises to improve the effectiveness and applicability of LLMs in emerging technologies such as 5G/6G applications.

1.3 Problem

The high energy expenditure and computational power of the execution of LLMs pose a challenge when it comes to expanding their usage in fields like healthcare, robotics, and smart cities. The existing LLMs dependency on cloud computing is a challenge, because it makes them not suited for real-time applications, due to their excessive latency, which would not allow for the timely response that such applications require. It also introduces a need for permanent internet access, limiting the situations where these models can be used. The possibility of data breaches is another concern, as they can occur when data is being sent to the cloud.

A new approach that could steer away from using cloud-based servers for the inference process would mean that the usage of LLMs could be greatly expanded, eliminating previous constraints such as excessive latency, issues with scalability, and privacy concerns. This would be beneficial

for the companies that have these models deployed, opening new fields to explore, such as real-time applications, and reducing the energy costs of large server farms. It would also enhance the experience for users, removing the need for internet access and improving the security of their data.

The idea of using FPGAs to solve these problems is recent, and some studies have already addressed this approach, with promising results [33, 10] in the deployment of LLMs in FPGAs, making the transition of LLMs into edge ever more near. Despite this, these implementations still face various challenges. First of all, to execute LLMs on FPGAs, it would be needed to reduce the models' size, but this is hard to do without impacting their performance. While HLS tools capable of compiling C/C++ into FPGA hardware exist, their potential for synthesizing LLMs is yet to be fully explored.

1.4 Research Questions

Taking all that was said into account, the goal of this dissertation is to explore a solution to the dependency of LLMs on cloud-based servers using FPGAs. This can be translated into the following research questions:

Q1: How to implement LLMs in FPGAs?

Q2: What phases of the inference step are more beneficial to execute on FPGAs?

Q3: How to reduce energy consumption in the inference step using FPGAs while minimizing performance loss?

Q4: How much energy can we save when executing the inference step in FPGAs when compared to inference on other types of devices?

1.5 Hypothesis

Due to their large number of programmable logic gates, FPGAs provide hardware reconfigurable for specific tasks, generally allowing for better performance-energy trade-offs. Recent research corroborates this, showing that it is possible to solve the computation and memory overheads of LLMs using both server-grade and edge-grade FPGAs [33, 10]. This means that FPGAs are a good option to explore as an alternative to having the inference executed on GPU-based cloud servers. However, this is an underexplored field and, although the potential is high, it is still hard to implement such an alternative. These challenges include the complexity of hardware-software co-design, the need for advanced quantization techniques to manage memory and computational constraints, and the development of efficient communication strategies for multi-FPGA setups, as highlighted in recent works such as [35] and [26].

With this said, this dissertation addresses an implementation of a LLM on an FPGA based on HLS tools.

Chapter 2

Literature Review

A systematic literature review was conducted, following the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology [21]. This methodology allows for a rapid review of the existing literature by providing a set of guidelines that help in the selection of relevant studies.

2.1 Research Domain

In recent years, with Large Language Models (LLMs) gaining the spotlight when it comes to new and revolutionizing technologies, various possibilities for their future application in different fields have emerged. This has not come without challenges however, considering that the large size of most of these models entail a high energy and computational power demand.

Some solutions to tackle these problems and allow the expansion of LLMs into the world of real-time applications on edge have been explored, and one of them that shows promise is their implementation in Field Programmable Gate Array (FPGA), which is the focus of the current study, as described by the research questions in Section 1.4.

The goal of this review is to identify studies that align with these research questions and explore possible integration of an implementation of LLMs in FPGAs with 5G/6G networks and edge computing. Figure 2.1 [7] outlines the steps of applying the PRISMA methodology, which will be described in detail in the following sections.

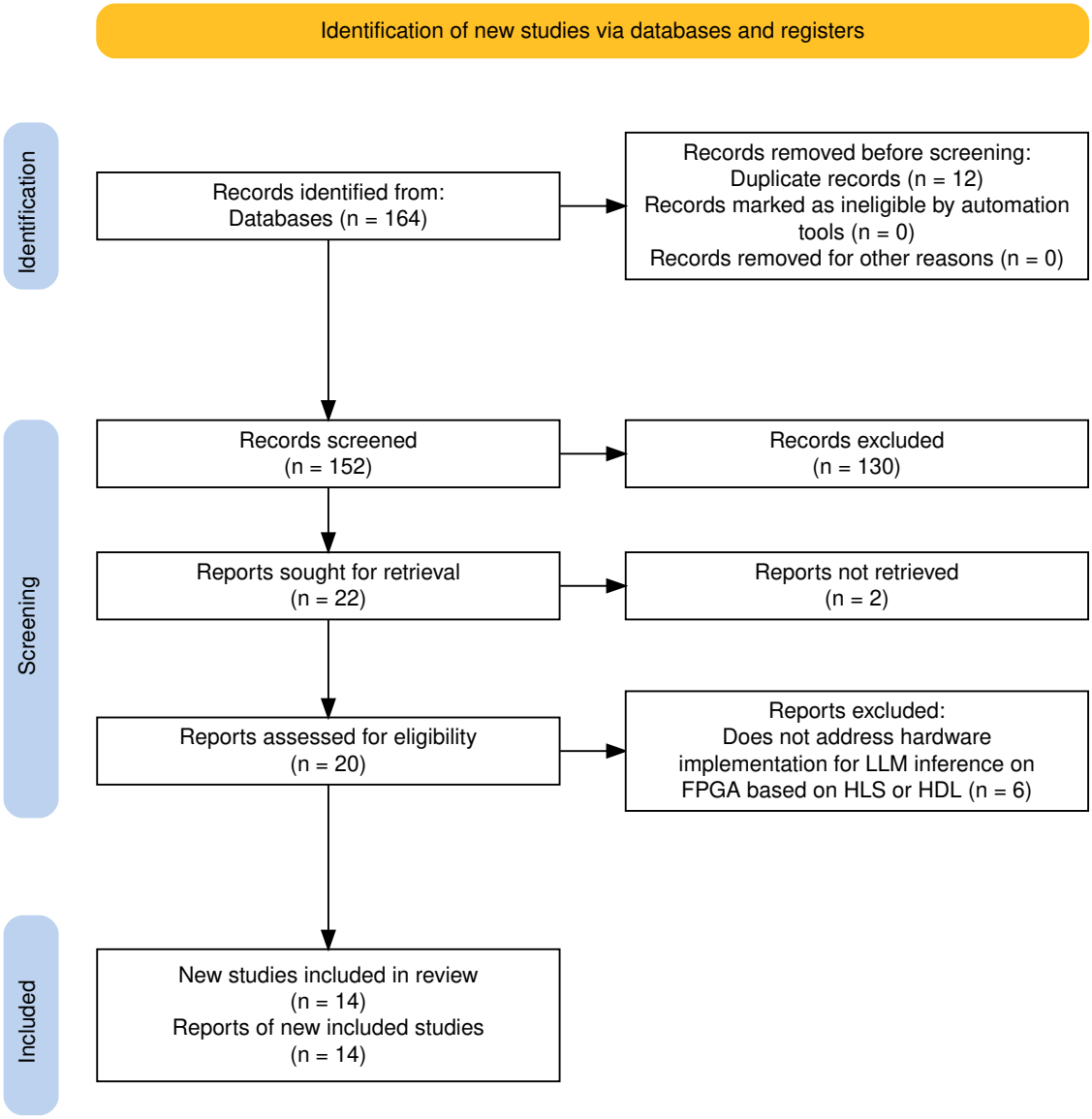


Figure 2.1: PRISMA flow diagram

2.2 Identification Phase

A selection of databases was made in order to search them for relevant studies. The platforms chosen were IEEE Xplore, SCOPUS, ScienceDirect, ACM Digital Library and arXiv. The search process involves the definition of keywords to look for. To narrow down the studies that result from the search to the most relevant ones, the keywords used have to be repeatedly refined with each querying of the databases. The firstly defined search queries are presented in Table 2.1.

Table 2.1: Initial search queries.

ID	Query
SQ1	(llm OR "large language model") AND (fpga OR "field programmable gate array")
SQ2	(llm OR llms OR “large language model” OR “large language models”) AND multimodal AND (“edge ai”)

2.2.1 Formulation and Refinement Process

The initial formulated strings, shown in Table 2.1, were refined by executing the queries in the platform arXiv and looking through the results, identifying too broad or missing terms.

2.2.1.1 SQ1 Formulation and Refinement Process

The goal of search string SQ1 is to identify studies that correlate LLMs and FPGAs. A gap was identified, as studies that had the keywords of SQ1 in plural form would not be found in some of the databases, so the plural of these keywords were added to the search query. Then, to broaden the results, As the results were too broad, the keyword "inference" was added to the query, as the focus of this study is on the execution of the inference step of LLMs in FPGAs.

2.2.1.2 SQ2 Formulation and Refinement Process

The goal of search string SQ2 is to find studies that explore the application of LLMs in Edge AI. The initial version, in Table 2.1, presented few results in arXiv, with only 9 studies found, so, for the final version, in Table 2.2, it was added an equivalent way of referring to Edge AI, "edge computing".

2.2.1.3 Formulation and Refinement Process Result

The search queries presented in Table 2.2 are the final, refined ones that were executed in all the data sources, following the PRISMA methodology.

Table 2.2: Final search queries.

ID	Query
SQ1	(llm OR llms OR "large language model" OR "large language models") AND inference AND (fpga OR fpgas OR "field programmable gate array" OR "field programmable gate arrays")
SQ2	(llm OR llms OR “large language model” OR “large language models”) AND multimodal AND (“edge ai” OR “edge computing”)

According to each search engine's capabilities and the amount of results, some specific criteria were applied in each data source:

1. **IEEE:** The search strings were searched for in all fields
2. **Scopus:** The search strings were searched for in the title, abstract and keywords
3. **ScienceDirect:** The search strings were searched for in all fields
4. **ACM Digital Library:** The search strings were searched for in the title, abstract and keywords
5. **arXiv:** The search strings were searched for in all fields

From the execution of the search queries referred in Table 2.2 with the criteria above, a total of 164 papers were found. The results were imported into EndNote, the reference manager chosen for this review. The distribution of the number of results for each data source is presented in Table 2.3.

Table 2.3: Number of papers retrieved from each data source by each search string.

Search Engine	SQ1	SQ2	Total
IEEE	4	15	19
Scopus	5	2	7
ScienceDirect	36	65	101
ACM Digital Library	5	2	7
arXiv	10	20	30
Total	60	104	164

2.3 Screening Phase

In this phase, the inclusion and exclusion criteria are established with the goal of narrowing down the pool of results by a systematic process that allows a quicker selection of relevant studies.

2.3.1 Inclusion and Exclusion Criteria

The 164 papers found in the identification phase were subjected to the 11 phases of inclusion and exclusion criteria enumerated in Table 2.4 to find the studies most related to the topic of this work, as described in Section 2.1.

Table 2.4: Inclusion and exclusion criteria used in the selection of results

Phases	Inclusion and Exclusion criteria
1	Results from executing the search queries on the data sources
2	Not duplicated
3	Date of the publication since 2020 (inclusive)
4	Published in the English language
5	Published in Journal, Conference, pre-print
6	Full-text access
7	Published in the area of "Artificial intelligence", "FPGAs", "reconfigurable computing", "Large Language Models"
8	Must include an implementation, or simulation, of transformer models
9	Must address a hardware implementation for LLM inference on FPGA, either HLS based or HDL based
10	Should include performance metrics related to energy efficiency and/or throughput
11	Document must discuss current limitations and challenges of the research performed (if applicable)

2.3.2 Removal of Duplicates

The reference manager EndNote provides a functionality of identifying duplicates in its "Library" options. This process found 152 unique records and 12 duplicates. Out of the 12 duplicates, 6 were manually selected to be kept according to which of the references had more information or was more recent. To pick out any other possible duplicates, a manual search through the results was conducted, where studies with the same title, year and authors were selected. 9 duplicates were identified, and 3 were manually selected to be kept according to which of the references had more information or was more recent. This results in a total of 152 papers for the next phases.

2.3.3 Screened Records

The remaining 152 papers were subjected to the remaining criteria of Table 2.4. EndNote's capabilities, coupled with a manual analysis of the title and abstract of the papers, allowed to identify the studies that respect the restrictions, resulting in 22 studies.

2.4 Eligibility Phase

The eligibility phase is where it is confirmed the full-text access of the studies obtained in Section 2.3 and the remaining criteria in Table 2.4.

For this purpose, EndNote's functionality of "full-text search" was used. EndNote was able to identify 8 PDFs, 5 URLs but could not locate 9 PDFs. Searching manually for the remaining

PDFs, 2 were found to not have full-text access. All year and paper types were correct, resulting in 20 remaining papers.

2.5 Selection Phase

In this last phase, the studies obtained in Section 2.4 were analyzed by reading their abstract, introduction, conclusions and, if these were not enough to determine their relevancy, their full-text. According to the criteria defined in Table 2.4, 6 papers were excluded as they did not address a hardware implementation for LLM inference on FPGA, either HLS or HDL based.

This resulted in 14 studies that were considered relevant at the end of the PRISMA methodology [36, 35, 33, 31, 30, 26, 25, 17, 14, 12, 10, 9, 3, 2].

Chapter 3

Related Work

3.1 Background

This section will present and explain the essential concepts to optimize LLM inference in FPGAs. LLMs are at the forefront of AI research due to their versatility across domains like healthcare, autonomous systems, and education. However, their computational and energy demands limit their applicability in real-time applications. Edge AI, leveraging decentralized computation on devices like FPGAs, offers an opportunity to address these limitations. FPGAs, with their reconfigurable hardware, provide a pathway to optimize LLMs inference for edge scenarios while minimizing energy consumption and maintaining accuracy.

3.1.1 Large Language Models (LLMs)

LLMs are advanced machine learning systems designed to process and generate human-like text. They are built upon the Transformer architecture, which revolutionized Natural Language Processing (NLP) by addressing limitations of previous models such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). LLMs like GPT, BERT, and LLaMA leverage massive datasets and billions of parameters to perform tasks like translation, summarization, and code generation with accuracy.

Transformers are the backbone of LLMs, relying on the mechanism of self-attention and positional embeddings to model dependencies in sequences, regardless of their length. An example of a Transformer architecture can be seen in Figure 3.1.

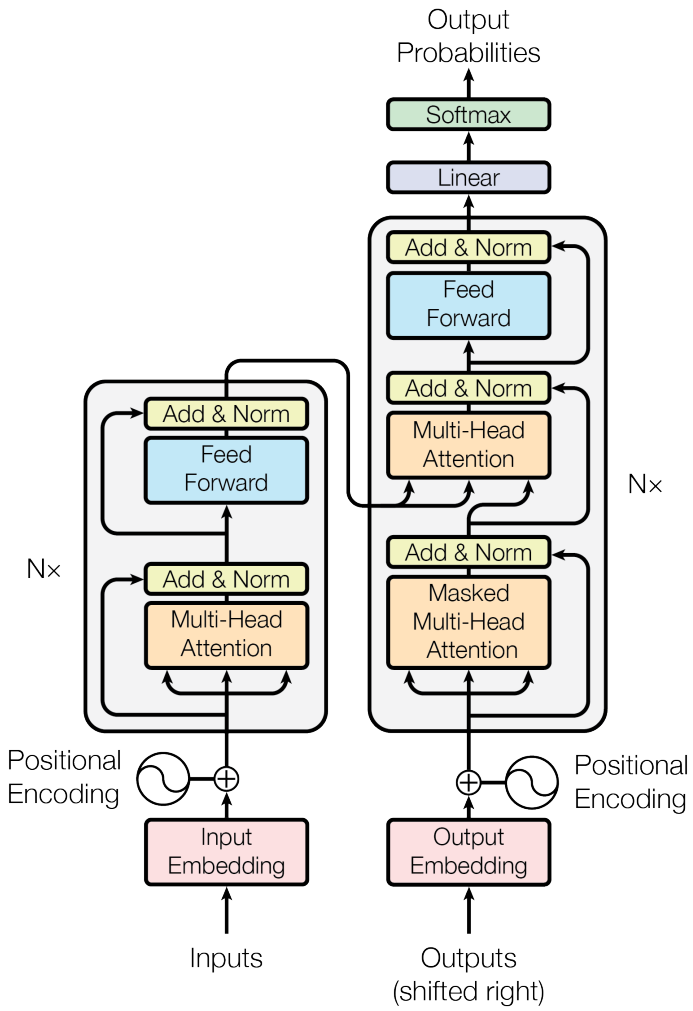


Figure 3.1: Example of a Transformer architecture [29].

The input to a Transformer is a sequence of tokens, where each token is mapped to a high-dimensional vector using an embedding matrix. Positional encoding is then added to these embeddings to retain the order of tokens in the sequence, since Transformers lack inherent sequential structures like RNNs. Transformers typically consist of an encoder and a decoder.

The encoder is a crucial component of Transformer models, designed to process input sequences and generate detailed contextual representations for each token. It operates through a series of stacked layers, with each layer comprising two primary sub-components, the multi-head self-attention mechanism and a Feedforward Neural Network (FFNN). At each layer, the multi-head self-attention mechanism allows the encoder to model relationships between all tokens in the sequence, dynamically determining which tokens are most relevant to each other. This is achieved by calculating attention scores based on queries, keys, and values, which are linear transformations of the input token embeddings. These attention scores are then used to produce weighted representations of the tokens, capturing dependencies regardless of their distance in the sequence. The FFNN, applied independently to each token, transforms these representations through non-linear activation functions, enhancing their expressiveness.

To ensure stability and efficient training, the encoder employs residual connections that add the input of each sub-layer to its output, preserving information flow across layers. Layer normalization is also applied after each sub-layer to maintain numerical stability and improve convergence. By stacking multiple encoder layers, the model refines the token representations iteratively, allowing it to capture increasingly complex patterns and relationships. The final output of the encoder is a sequence of context-rich embeddings that encapsulate information about the entire input, making it suited for tasks requiring an understanding of the full input context, such as classification or translation.

The decoder, on the other hand, is responsible for generating output sequences in tasks like text generation or machine translation. Like the encoder, it consists of multiple stacked layers but with some key differences tailored to its generative role. Each decoder layer includes a multi-head self-attention mechanism, a multi-head encoder-decoder attention mechanism, and a FFNN. The self-attention mechanism in the decoder is masked to ensure causality, meaning each token can only depend on earlier tokens in the sequence. This ensures that predictions are generated sequentially and prevents information from future tokens from leaking into the current prediction.

The encoder-decoder attention mechanism, presented in Figure 3.2 enables the decoder to incorporate information from the encoder's output. This facilitates tasks like translation, where the decoder needs to align its output with the input sequence. The multi-head self-attention mechanism is a key component of the Transformer architecture, enabling models to dynamically determine the relevance of different tokens in a sequence. It operates by projecting input token embeddings into three separate vectors: queries (Q), keys (K), and values (V). These vectors are used to compute attention scores, which measure the similarity between tokens. The attention scores are scaled by the dimensionality of the key vectors to stabilize gradients and passed through a softmax function to ensure they sum to one. These scores are then used to compute a weighted sum of the value vectors, producing a context-aware representation for each token. Using multiple attention heads allows the model to focus on diverse aspects of the sequence simultaneously, capturing relationships at different levels of granularity. The outputs from all heads are concatenated and linearly transformed to integrate the information from each head effectively.

Models like BERT, whose architecture is represented in Figure 3.3(a), utilize the encoder to perform bidirectional context understanding, where the attention mechanism considers both preceding and succeeding tokens in the sequence. This bidirectional attention allows the model to grasp the full context of a word within its sentence, making it particularly effective for tasks like question answering, sentiment analysis, and entity recognition. BERT's training objective, known as masked language modeling, involves randomly masking words in the input and predicting them based on the bidirectional context, enabling the model to capture complex dependencies within the data.

Autoregressive models, such as GPT, whose architecture is represented in Figure 3.3(b), predict the next token in a sequence based solely on previously generated tokens. This means that, at each step, the model uses its prior predictions as input, creating a left-to-right dependency, only requiring the use of a decoder. This sequential dependency makes autoregressive models suitable

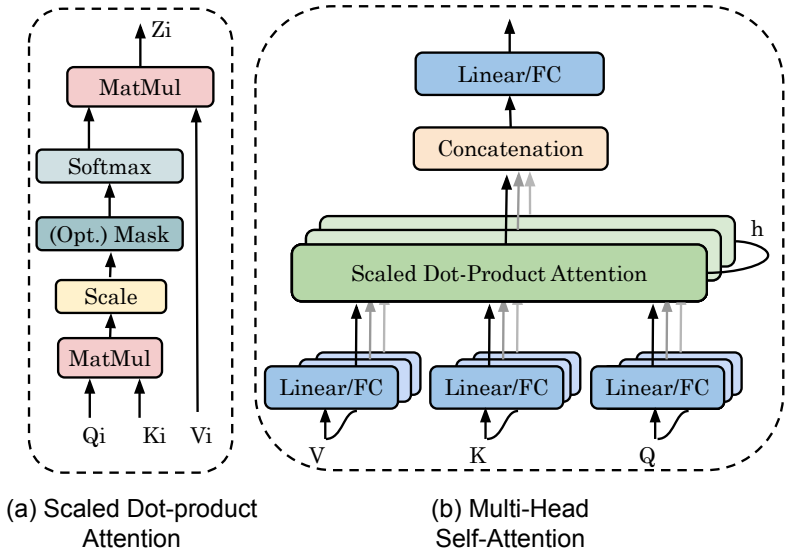


Figure 3.2: Attention mechanism in Transformers [3].

for tasks like text generation and completion but poses challenges for parallelism during inference, as tokens must be generated one at a time.

LLMs face scalability challenges due to the quadratic complexity of self-attention, which increases computational costs as the sequence length grows. This results in high demands for processing power and memory, often necessitating the use of specialized hardware like GPU or Tensor Processing Unit (TPU) clusters. Techniques such as model parallelism, where parameters are distributed across devices, and activation checkpointing, which reduces memory usage at the cost of additional computation, are used to manage these demands.

Inference bottlenecks further complicate scalability, particularly in autoregressive models like GPT. These models generate tokens sequentially, where each token depends on the previous ones, limiting parallelism and slowing down the process. This makes real-time or low-latency applications challenging without significant optimization.

3.1.2 Edge Artificial Intelligence (AI)

Edge AI refers to the deployment of AI models directly on edge devices, such as smartphones, IoT sensors, or embedded systems, rather than relying on centralized cloud computing. By processing data locally, Edge AI reduces latency, enhances privacy, and lowers energy consumption, making it ideal for real-time and distributed AI applications.

Edge AI relies on specialized hardware to efficiently perform AI inference in resource-constrained environments. Devices use low-power processors, accelerators like GPUs, TPUs, and FPGAs, or purpose-built chips such as NVIDIA Jetson and Google Edge TPU. Among these, FPGAs stand out for their hardware programmability and ability to optimize performance-energy trade-offs, making them particularly useful for customized AI tasks. To meet edge devices' computational

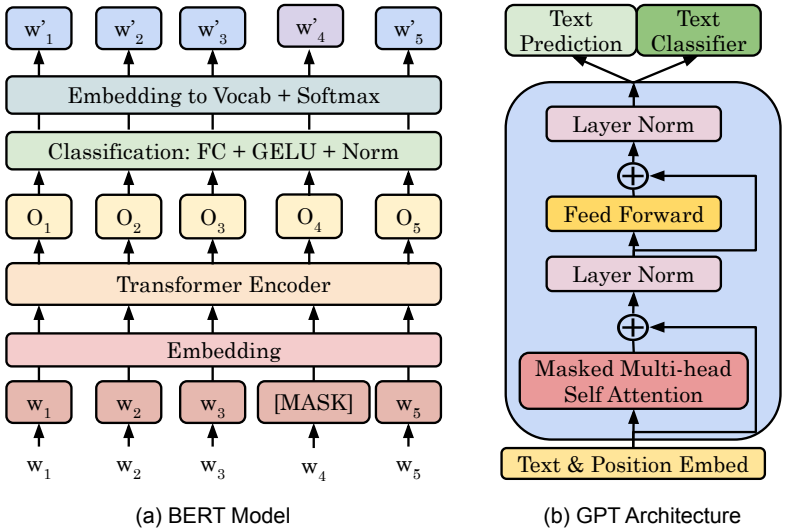


Figure 3.3: (a) BERT and (b) GPT architectures [3].

and memory limits, techniques such as quantization, pruning, and distillation are used to compress and optimize models. Lightweight architectures like MobileNet [11] or TinyML [18] are also commonly employed.

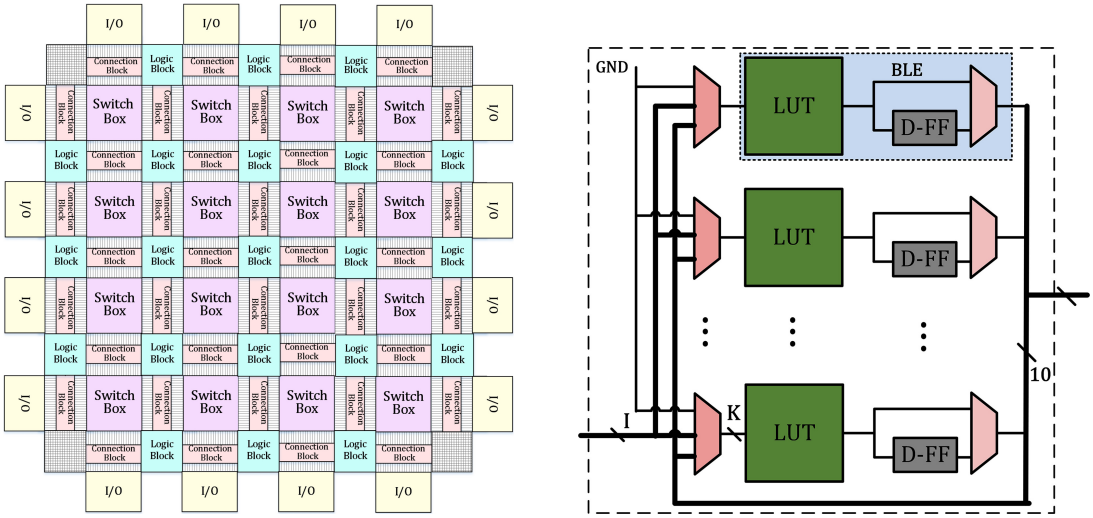
Despite its advantages, Edge AI faces challenges, such as limited resources, meeting real-time processing requirements, and the need for scalable solutions across different kinds of hardware platforms. These obstacles require efficient designs and frameworks like TensorFlow Lite or ONNX Runtime to ensure portability and compatibility.

Applications of Edge AI span various domains. In smart homes, it enables voice assistants and facial recognition systems, while in healthcare, it powers wearable devices for monitoring and diagnostics. Industrial IoT uses Edge AI for predictive maintenance and anomaly detection, and autonomous systems such as drones and robots rely on it for real-time decision-making [5, 32, 1, 28].

The benefits of Edge AI include reduced latency through local processing, enhanced data privacy by keeping information on the device, lower bandwidth costs, and improved energy efficiency. FPGAs further enhance these advantages by providing a flexible and energy-efficient platform for deploying AI models, including LLMs, in constrained environments. Techniques like quantization and pruning help overcome the computational challenges, enabling sophisticated AI capabilities at the edge.

3.1.3 Field Programmable Gate Arrays (FPGAs)

FPGAs are reconfigurable hardware devices that provide a flexible platform for implementing digital circuits. Unlike fixed-function hardware such as Application-Specific Integrated Circuitss (ASICs), FPGAs allow users to program their logic gates, enabling tailored designs for specific applications. This adaptability makes FPGAs particularly valuable for tasks requiring high performance and energy efficiency in areas such as Edge AI and LLM inference.



(a) Example of the architecture of an FPGA [23].

(b) Architecture of a Configurable Logic Block [23].

Figure 3.4: Architecture of FPGA and Configurable Logic Block

As presented in Figure 3.4, FPGAs consist of a grid of programmable logic blocks, interconnected through configurable routing channels. The architecture of the configurable logic blocks can be seen in Figure 3.4. These components allow the creation of hardware circuits, optimized for specific tasks. Additionally, FPGAs include specialized elements such as Digital Signal Processor (DSP) units specialized in arithmetic operations, block RAM for high-speed, on-chip memory storage, and high-speed I/O interfaces to enable fast communication with external devices. Unlike general-purpose processors, FPGAs achieve superior performance-energy trade-offs by eliminating the overhead of general-purpose computing, making them more suited for tasks with strict latency and power requirements.

3.1.3.1 High-Level-Synthesis (HLS)

HLS tools enable the design of FPGA-based systems using high-level programming languages like C/C++ rather than traditional Hardware Description Languages (HDLs) like Verilog or VHDL. This abstraction accelerates development, allowing software engineers to design hardware without extensive knowledge of digital circuit design. Tools such as Xilinx Vivado HLS and Intel HLS Compiler are widely used to translate high-level code into optimized FPGA configurations. HLS has become a cornerstone for implementing LLM inference on FPGAs, facilitating the integration of complex machine learning algorithms into reconfigurable hardware.

3.1.3.2 FPGAs in Edge AI

FPGAs have the potential to be used efficiently in Edge AI by addressing the computational and energy limitations of edge devices. Their ability to reconfigure hardware at the gate level enables optimized implementations for specific AI models, reducing energy consumption and improving latency. Compared to GPUs and Central Processing Units (CPUs), FPGAs present themselves as

a good alternative because of their high customizability, energy efficiency, and scalability. This makes them suitable for various edge applications, from small IoT sensors to high-performance embedded systems.

3.1.3.3 FPGAs in LLM Inference

The use of FPGAs for LLM inference is a growing area of research focused on reducing energy and latency costs. FPGAs are well-suited for accelerating computationally intensive operations in transformer models, such as matrix multiplications. Using HLS tools, researchers have developed FPGA-based accelerators that apply techniques like quantization to lower the precision of model parameters for better memory usage, pruning to remove unnecessary parameters, and pipeline parallelism to improve data flow through the model [3]. Studies show that FPGA implementations of LLMs, such as GPT and BERT, can match GPU performance while using much less energy [10]. These advancements make FPGAs a viable option for deploying LLMs on edge devices, balancing AI capabilities with resource efficiency.

3.2 State of the Art

This section will analyze and discuss the State of the Art in the acceleration of LLMs inference using FPGAs. The studies used here are the results of the Systematic Literature Review conducted in Chapter 2.

Designing Efficient LLM Accelerators for Edge Devices

In [9], the SECDA-LLM approach is presented, a novel platform for developing FPGA-based accelerators optimized for LLM inference on resource-constrained edge devices. The research addresses the challenges of high computational demands and memory overheads inherent in LLMs, making them inefficient for edge devices such as mobile phones or IoT hardware. The primary objective of the work was to streamline the development and deployment of hardware accelerators for LLM, using the SECDA methodology [8] within the *llama.cpp* inference framework [6].

The main contribution of the work is the SECDA-LLM platform, which integrates SystemC, for simulation and profiling, with HLS, for hardware generation. It connects *llama.cpp's* tensor operations with the hardware accelerator through a context handler that manages data, memory mappings, and quantization parameters. This design flow supports the prototyping, simulation, and deployment of FPGA accelerators for LLM inference.

To demonstrate the platform's capabilities, the authors present a case study where they design and evaluate a Block Floating Point (BFP) quantized MatMul accelerator, seen in Figure 3.5. This accelerator targets the MatMul operations, which represent **97%** of all computations within the target LLM, using quantized formats to optimize memory usage and computational efficiency. The design includes an instruction decoder, data mapper, scheduler, and a Super-Block Vector Processor (SBVP) to handle quantized operations efficiently. The accelerator processes quantized data with minimal accuracy loss while reducing computational overhead.

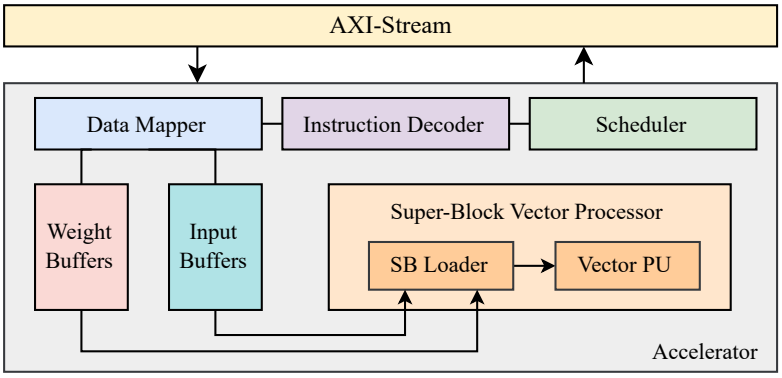


Figure 3.5: Overview of the block floating point quantized accelerator design for GGML's MatMul kernel, developed in [9].

The evaluation is conducted on the PYNQ-Z1 FPGA board, where the accelerator executes the TinyLlama model [34], a compact LLM containing 1.1 billion parameters. The results show an 11× speedup in token processing compared to a dual-core ARM NEON-based CPU, achieving a latency of 1.7 seconds per token. This performance improvement demonstrates the potential of FPGA-based accelerators to make LLM inference viable on edge devices, reducing reliance on cloud-based services and enhancing privacy and accessibility.

This work demonstrates the value of integrating hardware accelerators with software frameworks for LLM inference. SECDA-LLM addresses current limitations in FPGA design flows and provides a foundation for future advancements in Edge AI.

HLSTransform: Energy-Efficient Llama 2 Inference on FPGAs via High Level Synthesis

In [10], the development of FPGA-based accelerators for transformer models using HLS. Specifically, the work targets the llama2 model and, in general, addresses the growing need for energy-efficient inference in LLMs, as GPUs, the current dominant hardware accelerators, consume significant energy and are less suitable for edge computing applications.

The HLSTransform methodology employs HLS tools to streamline the design of FPGA accelerators, allowing hardware descriptions to be written in high-level languages like C++ rather than conventional RTL. The process integrates kernel and host code, where the kernel defines the FPGA's computational operations, and the host manages communication between the FPGA and external systems via Direct Memory Access (DMA). Key optimizations include pipelining, loop unrolling, and memory partitioning, which enhance parallelism and throughput while addressing on-chip memory constraints.

To implement Llama 2 inference, the authors used an 8-bit integer quantization technique to reduce the model's precision and memory footprint without significant accuracy loss. The quantized model is deployed on the Xilinx Virtex UltraScale+ VU9P FPGA. The experimental results show that the FPGA achieves a **12.75× reduction** in energy consumption per token, compared to a CPU, and an **8.25× reduction**, compared to a GPU. Additionally, the FPGA provides a **2.46×**

speedup in inference speed over the CPU while operating at 0.53× the inference speed of the GPU, highlighting its efficiency despite lower clock speeds and memory capacity.

The study emphasizes the advantages of using HLS for rapid prototyping and demonstrates the potential of FPGAs as viable alternatives to GPUs for energy-efficient transformer inference. The authors also open-sourced their code, to allow the expansion of the work being done on FPGA-based LLM inference accelerators.

SeedLM: Compressing LLM Weights into Seeds of Pseudo-Random Generators

In [25] a novel weight compression technique for LLMs, called SeedLM, is introduced. The method aims to address the memory and energy bottlenecks in LLM inference by compressing model weights into seeds for pseudo-random generators, significantly reducing memory access requirements during inference. This approach uses Linear Feedback Shift Registers (LFSRs) to reconstruct weights efficiently, enabling faster, energy-efficient computation on hardware-constrained platforms like FPGAs.

SeedLM operates by dividing the weight matrices of an LLM into smaller blocks and representing each block as a linear combination of pseudo-randomly generated basis vectors. Using a seed and a small set of coefficients, the technique reconstructs the weights during inference. Unlike traditional methods, SeedLM is data-free and does not require calibration data, making it generalizable across tasks. It also achieves high compression ratios with minimal accuracy loss, retaining approximately **98%** of the accuracy of the original model at 4-bit quantization.

The FPGA-based implementation of SeedLM is tested on a Xilinx Virtex7 FPGA. It includes an LFSR module for generating pseudo-random matrices and a hardware design optimized for matrix-vector multiplications. The design, presented in Figure 3.6, supports efficient memory access by compressing weights to 4 bits, allowing a fourfold increase in memory throughput compared to FP16 baselines. Experiments conducted with Llama 2 and Llama 3 models demonstrate up to a **4x speedup** in memory-bound tasks and reductions in latency and energy consumption, when compared to the FP16 reference design. For instance, the compressed Llama 3 model achieves competitive zero-shot accuracy across various benchmarks while reducing memory requirements and improving inference performance.

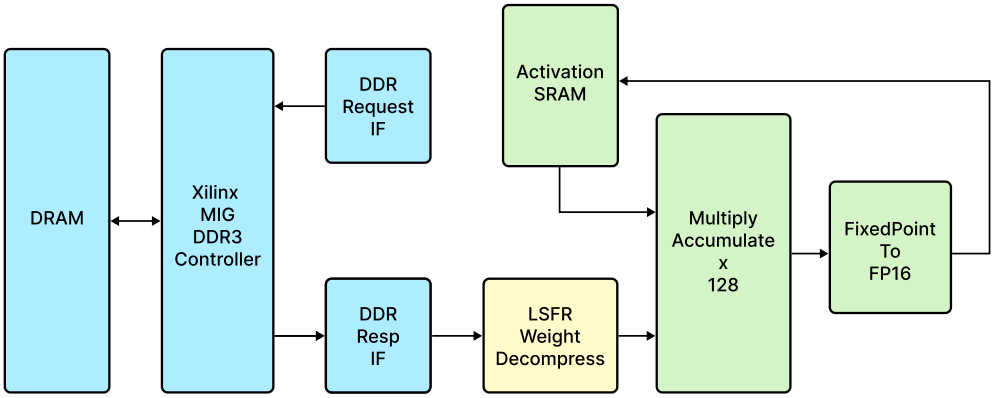


Figure 3.6: Block diagram of the RTL design, developed in [25].

This study highlights the potential of SeedLM to transform the deployment of LLMs in energy-constrained environments. By trading compute for memory efficiency, the method offers a scalable solution for real-time LLM inference on edge devices.

HotaQ: Hardware Oriented Token Adaptive Quantization for Large Language Models

In [26], HotaQ, a framework designed to optimize LLM inference through hardware-aware token adaptive quantization, is introduced. This method addresses the limitations of traditional quantization, which often leaves activations in high-precision formats, limiting acceleration potential on edge devices and FPGAs. HotaQ combines token pruning and mixed-precision quantization to balance computational efficiency with task performance.

HotaQ operates by assigning quantization precision levels to tokens based on their importance during inference. Informative tokens are processed with higher precision (8-bit), while less critical tokens use lower precision (4-bit). To manage the challenges of quantization-induced outliers, the framework employs an activation-aware token pruning mechanism, which reduces the influence of outliers and improves model performance. This strategy ensures an efficient trade-off between speedup and accuracy retention.

On FPGA platforms, HotaQ uses hardware-oriented optimizations such as memory tiling and DSP packing. The FPGA implementation includes systolic-array-based INT4 and INT8 multiplication units, enabling parallel computation and efficient use of on-chip resources. A custom tiling scheme reduces communication overhead between high-bandwidth memory and processing elements. Additionally, the framework integrates a top-k sorting module to dynamically identify and prioritize important tokens during inference, further improving performance.

HotaQ was evaluated using LLaMA, OPT, and BLOOM models across various benchmarks. Results demonstrated up to **5.2× speedup** on the Xilinx U280 FPGA for the LLaMa-65B compared to FP16 baselines. The LLaMa-7B achieved a **3.88x speedup** and a power consumption of 47W, meaning 1.22W per token calculations. The framework's ability to dynamically adjust quantization precision and prune redundant tokens shows its suitability for usage in edge and FPGA environments.

LlamaF: An Efficient Llama2 Architecture Accelerator on Embedded FPGAs

In [30], LlamaF, a specialized FPGA-based accelerator designed to improve inference performance for LLMs on embedded systems, is introduced. The study addresses the challenges of deploying models like Llama2 on constrained devices, such as limited off-chip memory bandwidth, computational intensity, and memory overhead. LlamaF aims to optimize inference through quantization and hardware acceleration, making LLMs viable for edge devices.

A key contribution of this work is the group-wise INT8 quantization strategy (W8A8), which compresses model weights and activations from 32-bit floating point to an 8-bit representation. This reduces the memory footprint of the TinyLlama model [34] from 4.4 GB to 1.1 GB with minimal accuracy loss, as demonstrated in the WikiText-2 dataset [20]. The quantization technique divides weight matrices into groups with individually optimized ranges, improving precision while minimizing hardware complexity.

The LlamaF hardware accelerator, seen in Figure 3.7, is designed around a fully pipelined architecture optimized for matrix-vector multiplications, one of the most common operations in LLM inference. The design incorporates pre-processing, which converts and caches quantized data, a dot-product stage, that performs Single Instruction, Multiple Data (SIMD) vector multiplications, and an accumulation stage that scales and aggregates results. These stages are interconnected using a task-level scheduling approach, allowing asynchronous memory transfers to overlap with kernel execution. This reduces latency and maximizes throughput.

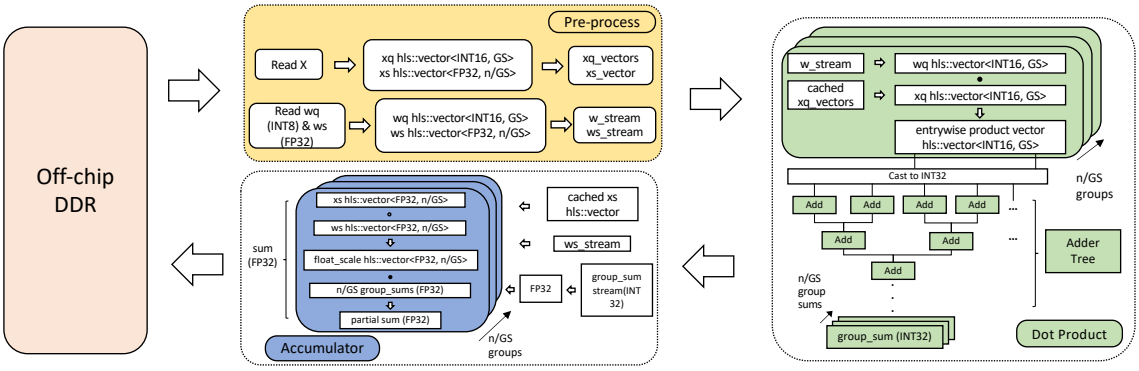


Figure 3.7: Overview of Llamaf hardware design, developed in [30].

The accelerator was implemented on the Xilinx ZCU102 platform and tested with the TinyLlama 1.1B model. The experimental results show a **14.3–15.8× speedup** in token generation and a **6.1× improvement in energy efficiency** compared to the ARM Cortex-A53 processor without FPGA acceleration. These results highlight the feasibility of using embedded FPGAs for LLM inference, achieving significant performance gains while improving power consumption.

FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs

In [33], FlightLLM, an FPGA-based accelerator designed for efficient inference of modern LLMs, is introduced. Addressing challenges like high computational demands, memory overheads, and inefficient processing of sparsified and quantized models on GPUs, the authors propose a comprehensive FPGA solution for accelerating generative models like LLaMA2-7B and OPT-6.7B.

A key feature of FlightLLM is its configurable sparse DSP chain, which supports a variety of sparsity patterns, including block-wise and N:M sparsity, while maintaining high computational efficiency. This design allows the FPGA to utilize DSP48 cores effectively for both dense and sparse matrix computations. To enhance memory bandwidth utilization, the system employs an always-on-chip decode scheme, where activations are processed directly in on-chip memory during the decode stage, significantly reducing off-chip data transfers. This approach boosts bandwidth utilization from 35.6% to 65.9%. Furthermore, the system integrates mixed-precision quantization, transforming data into a unified INT8 format to minimize LUT overhead and optimize hardware resources.

To address the issue of large compilation overheads, caused by large input lengths, FlightLLM incorporates a length-adaptive compilation method. By grouping instructions for similar token lengths and optimizing memory access patterns, the instruction size is reduced from terabytes to a manageable 3.25GB, enabling real-world deployment on FPGAs like the Xilinx Alveo U280.

The FlightLLM architecture includes a unified Matrix Processing Engine (MPE) for efficient computation, a Special Function Unit (SFU) for handling operations like softmax and normalization, and a task scheduler for workload distribution. These components are optimized for FPGA-specific features, such as High Bandwidth Memory (HBM) plus Double Data Rate (DDR) hybrid memory systems, ensuring scalability and performance.

Evaluated on the Xilinx Alveo U280 and Versal VHK158 platforms, FlightLLM achieves a **6.0× improvement in energy efficiency** and a **1.8× better cost efficiency** compared to NVIDIA V100S GPUs, and surpasses A100 GPUs in throughput under certain configurations, as can be seen in Figure 3.8. These results demonstrate the potential of FPGAs to meet the demanding requirements of LLM inference while offering significant energy and cost savings.

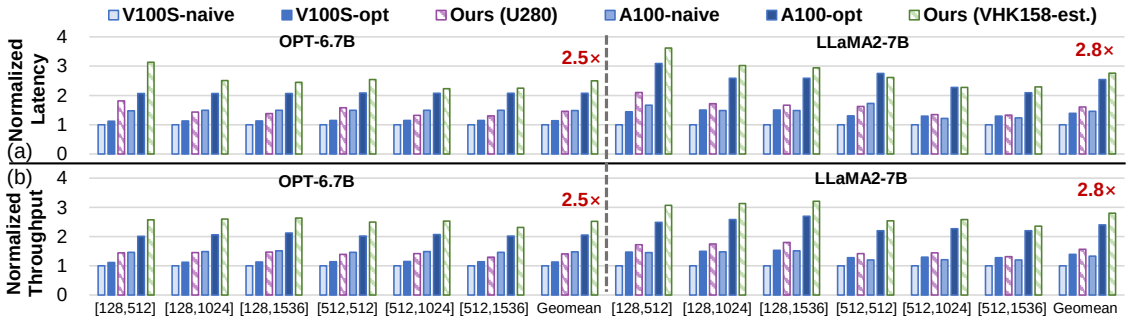


Figure 3.8: Latency and throughput of FlightLLM and V100S/A100 GPU. The horizontal axis represents [prefill size, decode size] [33].

LORA: A Latency-Oriented Recurrent Architecture for GPT Model on Multi-FPGA Platform with Communication Optimization

In [35], LORA, a low-latency acceleration platform for GPT models on multi-FPGA systems, is presented. The research focuses on addressing the challenges of high memory and computational overhead during GPT inference, particularly in the Prefill and Decode stages. LORA employs novel synchronization schemes and a recurrent hardware architecture to improve computational efficiency and reduce communication bottlenecks.

LORA's methodology involves dividing GPT model parameters across multiple FPGA boards using model parallelism. The architecture optimizes synchronization timing by overlapping computation and communication, minimizing system congestion. For example, LayerNorm and fully connected layers employ fine-grained synchronization steps, where partial sums are transmitted between boards to reduce data volume and latency. These steps enable efficient reuse of data and improve throughput.

The core of LORA's hardware design is its Matrix Processing Unit (MPU) and Vector Function Unit (VFU). The MPU utilizes tiling schemes and a broadcast array to accelerate inter-matrix and vector-matrix multiplications. Each FPGA board employs recurrent structures for low-latency processing of small batch sizes, allowing efficient execution of sequential tasks in the Decode stage. The VFU handles operations like LayerNorm, GELU, and SoftMax, using parallelism and pipelined algorithms to enhance performance.

LORA's implementation on Xilinx Alveo U280 FPGAs demonstrates significant performance improvements. The platform achieves an **11.1× speedup** over NVIDIA V100 GPUs for GPT-2 inference, with up to **4× and 2.7× improvements** in the Prefill and Decode stages, respectively, compared to existing multi-FPGA accelerators. By utilizing efficient communication strategies and leveraging FPGA hardware, LORA reduces inference latency and computational overhead, making it suitable for real-time applications in data centers.

Scalable MatMul-free Language Modeling

In [36], a novel approach to LLM inference that eliminates MatMul operations is presented. This method aims to address the significant computational and memory overheads typically associated with LLM inference. By replacing MatMul operations with ternary weight-based accumulations and lightweight operations, the proposed architecture reduces computational cost and memory usage while maintaining competitive performance at billion-parameter scales.

The core methodology involves two main innovations. First, dense layers are replaced with BitLinear modules that utilize ternary weights, restricting the values to 1, 0, and 1. This transforms MatMul into addition and subtraction operations, significantly reducing computational complexity. Second, self-attention mechanisms are replaced with a MatMul-free token mixer, based on a Linear Gated Recurrent Unit. This unit leverages element-wise operations and accumulations, eliminating traditional inter-matrix multiplications while preserving the ability to model sequential dependencies effectively.

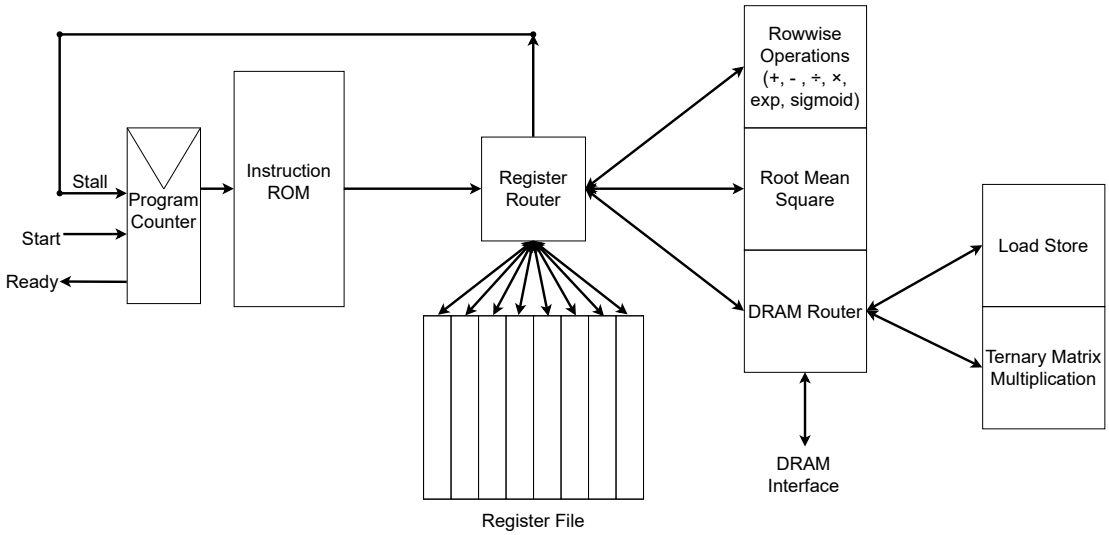


Figure 3.9: RTL implementation for running MatMul-free token generation, developed in [36].

The FPGA implementation of this MatMul-free architecture is realized using SystemVerilog on the Intel Stratix 10 FPGA. The design, presented in Figure 3.9, incorporates specialized functional units, such as a Ternary Matrix Multiplication unit, which efficiently processes ternary operations, and a Root Mean Square normalization unit. The architecture also employs a custom instruction set, written in assembly, to optimize data movement and computation. Key optimizations, including out-of-order execution and SRAM-based buffering, reduce latency and enhance parallelism.

Experimental results demonstrate the efficiency of the MatMul-free architecture. On a D5005 Stratix 10 FPGA, the implementation achieves a token processing speed of **62 tokens per second** for a 370M-parameter model, with an estimated energy consumption of **13W**. Additionally, the FPGA implementation achieves a **10× reduction in memory usage** compared to traditional Transformer models, with competitive accuracy across multiple benchmarks. The study highlights the potential for further optimizations, such as caching and pipelining, to improve performance further.

This work demonstrates that eliminating MatMul operations in LLMs is both feasible and beneficial for scalable, energy-efficient inference.

3.2.1 Summary

The reviewed works, which can be seen in Table 3.1, demonstrate substantial progress in leveraging FPGAs for LLM inference, highlighting their potential as energy-efficient and customizable alternatives to GPUs and CPUs. FPGAs are particularly well-suited for tasks involving high computational demands due to their reconfigurable nature and ability to integrate domain-specific optimizations. Various techniques, such as quantization, pruning, and pipeline parallelism, were

Table 3.1: Summary of implementations in each research article, gathered in Chapter 2

Work	**Year**	**Objective**	**Model Size**	**Language**	**Platform**	**Inference Speed (tokens/s)**
[9]	2024	FPGA-based inference for LLMs on edge devices	1.1B	C/C++ & Vivado HLS	Xilinx Z020	0.588
[10]	2024	Efficient Llama 2 inference using HLS tools	110M	C/C++ & Vivado HLS	Virtex UltraScale+ VU9P	57.11
[25]	2024	Compress LLM weights using seeds for pseudorandom number generators	70B	N/A	AMD Virtex7	N/A
[26]	2024	Token-adaptive quantization for efficient inference	7B-65B	N/A	Xilinx U280	9.5-2.4
[30]	2024	Efficient Llama2 inference on embedded FPGAs	1.1B	C/C++ & Vitis HLS	Xilinx ZCU102	1.478
[33]	2024	Complete FPGA mapping flow for LLM inference	7B	C/C++ & Vitis HLS	Xilinx Alveo U280 & Versal VHK158	92.5
[35]	2024	Low-latency GPT inference on multi-FPGA systems	345M & 774M	C/C++	Xilinx Alveo U280	N/A
[36]	2024	MatMul-free inference for lightweight LLMs	370M & 1.3B	C/C++ & SystemVerilog	Intel D5005 Stratix 10	62 & 23.8

consistently applied across the research articles to address the computational and memory challenges of LLMs.

Key achievements include improvements in energy efficiency, with reductions of up to 15× compared to traditional CPUs, and competitive throughput rates. For example, [10] showcased significant energy savings while maintaining competitive inference speed, and [33] optimized sparse operations to maximize bandwidth utilization. These advancements underscore the feasibility of deploying FPGAs for real-time applications on resource-constrained devices.

Despite these successes, several limitations persist. Many implementations struggle to balance quantization-induced accuracy loss and computational gains. Techniques like adaptive quantization, in [26], demonstrate promise but require further refinement to generalize across diverse LLM architectures. Another challenge lies in scalability, when deploying larger models, such as those exceeding 10B parameters, often results in diminished throughput due to memory bandwidth constraints and sequential processing bottlenecks.

Additionally, the complexity of hardware-software co-design remains a barrier. While HLS tools have simplified FPGA programming, achieving optimal resource utilization still demands significant manual effort and expertise.

Future work should explore improved quantization techniques, dynamic reconfiguration to adapt hardware to varying workloads, and enhanced HLS tools to further abstract hardware design complexities. The reviewed studies suggest that addressing these challenges could make FPGAs a cornerstone for deploying LLMs in edge and real-time environments.

Chapter 4

Proposed Approach

4.1 Problem Statement

The execution of LLMs is characterized by significant computational and energy demands, mostly limiting their deployment and execution to cloud-based servers [22]. While this enables the handling of extensive workloads, it creates challenges for latency-sensitive, bandwidth-intensive, and privacy-critical applications such as autonomous vehicles, healthcare, and smart cities [5, 32, 1, 28]. The inference process, particularly the decode phase, presents a bottleneck due to its sequential nature, leading to underutilization of hardware resources and increased energy consumption.

Current hardware solutions, including GPUs and dedicated accelerators like NVIDIA Jetson and Google Edge TPU, fail to address these issues fully [24, 13]. These devices have low computational efficiency, poor memory bandwidth utilization, and significant compilation overheads, restricting their use in edge devices and real-time scenarios. Furthermore, reliance on cloud-based infrastructure introduces additional challenges, such as high latency that undermines real-time responsiveness, the requirement for constant internet connectivity, and heightened risks to data privacy during cloud transmissions.

FPGAs present a promising alternative to existing solutions by offering reconfigurable hardware that allow the development of optimizations for specific tasks. Their potential lies in reducing energy consumption, improving computational efficiency, and enabling scalable deployment in latency-sensitive, real-time applications. However, several challenges remain. Executing LLMs on FPGAs requires model compression techniques to fit within FPGA constraints without a significant performance drop.

To address these issues, this dissertation is guided by the research questions presented in Section 1.4.

These questions form the basis for developing an approach that takes advantage of the flexibility of FPGAs to enhance the feasibility of deploying LLMs in edge and real-time environments. By addressing the sequential nature of the decode phase and optimizing energy-performance tradeoffs, this research seeks to overcome the limitations of current solutions, referenced in Chapter 3, ultimately expanding the applicability of LLMs to a broader range of technologies and devices.

4.2 Work Overview

The goal of this work was to produce an FPGA-based accelerator for the inference step of an LLM, using HLS.

This study considered the Xilinx ZCU102 and PYNQ-Z2 boards as possible hardware platforms. Despite the Xilinx ZCU102 offering enhanced capabilities, the PYNQ-Z2 board has an advantage in terms of ease of use and accessibility. This led to the PYNQ-Z2 being picked as the board to implement LLM inference circuits through Vitis HLS.

Existing LLM implementations in C/C++ were identified and analyzed for compatibility with the PYNQ-Z2 platform. The TinyLlama model [34] was chosen as the most suitable candidate to be run on the PYNQ-Z2 board due to its size. The selected implementation was adapted for synthesis using Vitis HLS.

To ensure the feasibility of deploying LLMs on FPGAs, various optimizations, some of them addressed in Chapter 3, were studied and applied, including model quantization, loop unrolling, pipelining, resource sharing, and memory partitioning. These optimizations were tested to balance energy efficiency and inference speed while trying to keep accuracy loss at a minimum.

Examples of key evaluation metrics used include throughput, token latency, and energy consumption. These metrics align with standards from related studies, identified in Chapter 3, and will ensure comparability with results from other hardware platforms.

Chapter 5

Implementation

5.1 Preparation

5.1.1 Llama2 Software Implementation

The model chosen for this work is Llama 2 [27], a family of large language models developed and released by Meta. Llama 2 is based on the standard decoder-only transformer architecture, which is highly effective for autoregressive text generation. It was trained on 2 trillion tokens of publicly available online data, a 40% increase over its predecessor, Llama 1. Key architectural improvements include a doubled context length of 4096 tokens, the use of RMSNorm for pre-normalization, the SwiGLU activation function, and rotary positional embeddings.

Llama 2 was released in several sizes, primarily models with 7 billion, 13 billion, and 70 billion parameters. This range of sizes allows it to be adapted to different computational resources. The models were also released as both pre-trained foundation models and fine-tuned chat versions, which were optimized for dialogue using Reinforcement Learning from Human Feedback (RLHF). The open availability of Llama 2, along with its lightweight nature compared to other open-source models, makes it an excellent candidate for research and custom implementations.

The foundation for this project is the *llama2.c* repository [15]. This repository offers a clean, single-file C implementation for running inference with the Llama 2 model. We chose this implementation because its simplicity makes it a strong starting point for hardware acceleration.

The *llama2.c* [15] code, specifically the *run.c* file, directly implements the transformer architecture discussed in Section 3.1.1. It avoids using large deep learning libraries like *PyTorch* or *TensorFlow*. Instead, it focuses only on the inference step. The program loads the pre-trained Llama 2 model, its configuration, and vocabulary from a file. When given a text prompt, it converts the text to tokens and begins the forward pass. This process, which includes the self-attention and feed-forward network calculations, generates a new token. The new token is then fed back into the model to produce the next one. The entire logic is contained in one C file and uses only standard C libraries, making it highly portable. The execution flow of this code can be seen in Figure 5.1.

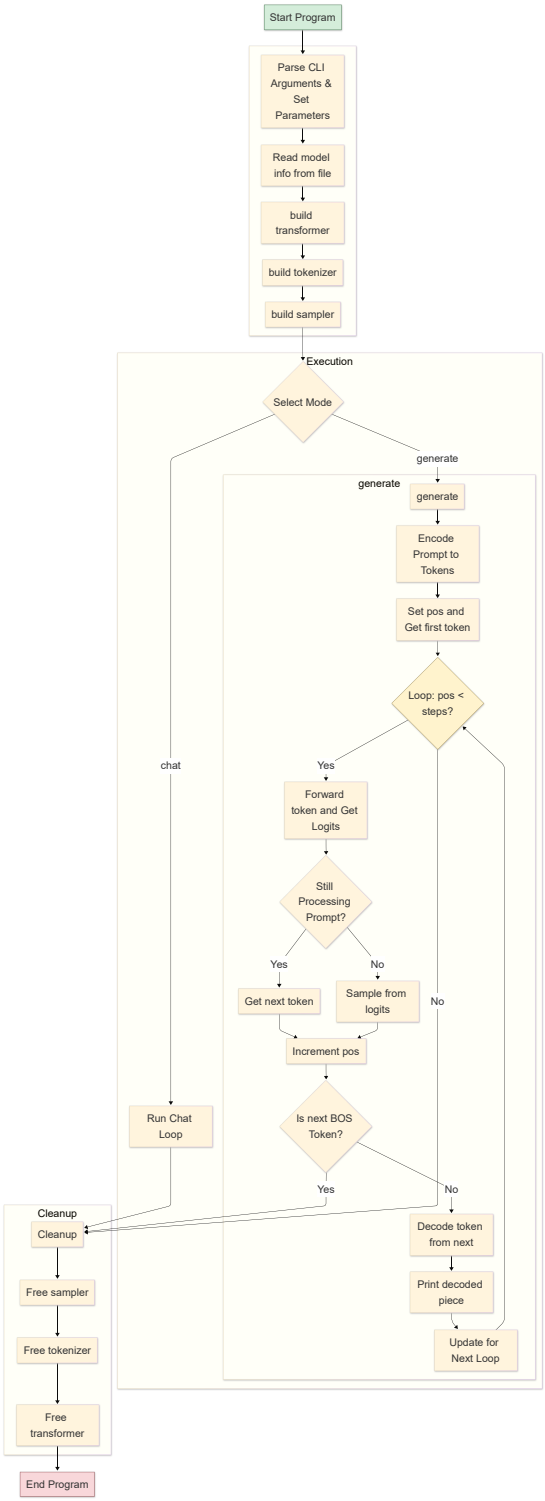


Figure 5.1: Execution flow of *run.c*.

Using a minimal implementation like *llama2.c* [15] provides several benefits for a hardware acceleration project. The clear, single-file structure makes it much easier to analyze and find computational bottlenecks. This was essential, as it made profiling the code with *gprof* much simpler, and allowed for better identification of which functions to accelerate. The simplicity of the code also helps in the hardware and software partitioning stage. It is easier to decide which parts should run on the processor and which should be offloaded to custom hardware on the FPGA. Finally, *llama2.c* has very low overhead. Unlike large frameworks, it does not consume significant resources, which is a key advantage when working with resource-limited embedded systems like the PYNQ-Z2 board. This approach gives us the control needed to effectively design and test hardware accelerators for the LLM.

5.1.2 PYNQ-Z2 Board

The PYNQ-Z2 is based on the Xilinx Zynq-7000 XC7Z020-1CLG400C System-on-Chip (SoC). This chip contains two main parts. The first is the PS, which has a 650MHz dual-core ARM Cortex-A9 processor. The Linux operating system and Python applications run on this processor. The second part is the PL, which is an FPGA fabric equivalent to an Artix-7 device.

The programmable logic provides the resources for creating custom hardware accelerators. The XC7Z020 includes 13,300 logic slices, 630 KB of block RAM for on-chip data storage, and 220 DSP slices. The DSP slices are designed for arithmetic operations, which are important for accelerating the computations in machine learning. The board also has 512MB of DDR3 RAM that is shared between the processing system and the programmable logic. This shared memory allows for data transfer between the software and hardware components.

For external connections, the PYNQ-Z2 includes a Gigabit Ethernet port, HDMI input and output, USB, and audio ports. It also provides expansion headers, such as Pmod, Arduino, and Raspberry Pi connectors. These allow the board to connect with many other peripherals and sensors. This combination of a processor, reconfigurable logic, and I/O makes the PYNQ-Z2 a functional and cost-effective platform for this project, particularly suitable for our target of edge-grade deployments where modest, inexpensive devices are preferred.

5.1.3 Python Productivity for Zynq (PYNQ)

The PYNQ framework is a set of Python libraries providing hardware abstraction, supported by a subset of AMD's FPGA SoC boards, including the PYNQ-Z2. By first deploying an Operating System (OS) image on the ARM processor side of the device, the PYNQ libraries provide an easier interaction with hardware directly from OS level, including from Python scripts or applications compiled natively, without the need for compilation of custom images and device tree blobs, and low-level drivers. An example of this abstraction can be seen in Figure 5.2.

The PYNQ framework offers a different development approach compared to traditional methods. A bare-metal system provides the best performance but is the most difficult, as the developer must write low-level drivers and manage memory manually, which is challenging for projects like

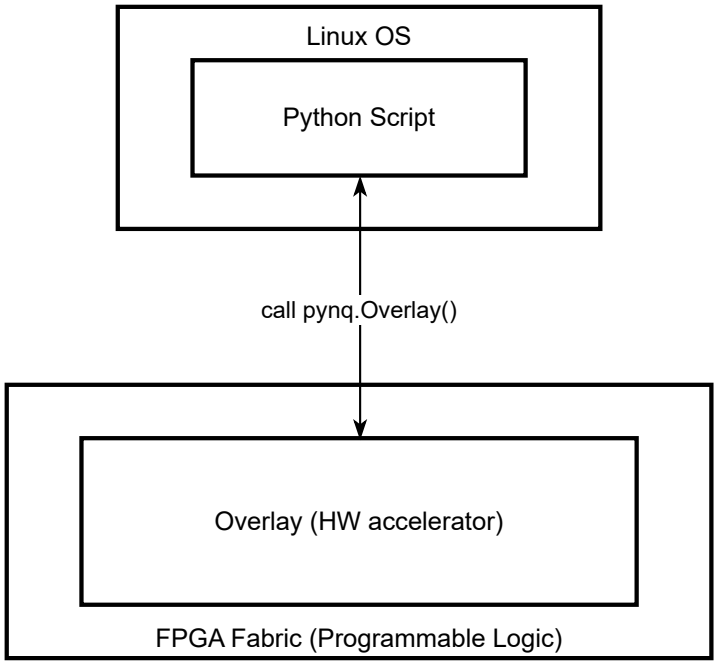


Figure 5.2: PYNQ's hardware abstraction.

LLM inference. A standard embedded Linux approach is more common but still requires writing C drivers to manage memory-mapped registers and data transfers for the FPGA.

PYNQ builds on embedded Linux by adding a Python abstraction layer. This allows developers to interact with hardware using Python objects instead of C drivers, improving productivity and ease of use. PYNQ strikes a good balance for complex projects, combining rapid Python development with the performance benefits of FPGA hardware acceleration.

5.1.4 TinyStories Models

The choice of a pre-trained model for this project depended on the hardware's limits. The TinyStories model family [4] was selected as the target for evaluation, in combination with the *llama2.c* inference engine [15], explained in Section 5.1.1. TinyStories models are small language models trained in a unique way. Instead of using web data, they learn from simple stories generated by a larger model. The stories use a vocabulary that a young child between the ages of 3 and 4 would understand. Some examples of such stories can be seen in Figure 5.3.

This training method allows even small models to learn grammar and reasoning. The models can produce coherent text while having a small number of parameters. This makes the TinyStories models a good choice for running on a device with limited resources like the PYNQ-Z2 board.

For the kind of hardware accelerator to be developed in this project, the most important hardware limit is the on-chip memory. Fast processing requires the model's weights to be stored in

Figure 5.3: Examples of the output of the TinyStories models with prompt *"Once upon a time"*.

stories260K

Once upon a time, there was a girl named Lily. She loved to play outside and show her friends. One day, Lily's mom gave her a stick for her. Lily was very excited and pointed it. Lily didn't know what to do. She said, "Mom, can you help you find a fake bird?" Her mom replied, "Sure, Lily, you can't relax the stick. It's a coffee for the girl's back." Lily thought about it and said, "Okay, Mom. I can fix it too." When they got home, Lily didn't listen to her mom. She had never seen her mom's head before. She picked up the bandag

stories15M

Once upon a time, there was a little girl named Lily. She had a teddy bear named Teddy. Teddy was her best friend and she loved him very much. One day, Lily wanted to play with her favorite doll, but she couldn't find it. She looked everywhere, but Teddy was gone. Lily started to weep and her mommy came to her. "What's wrong, sweetie?" asked mommy. "I lost Teddy," replied Lily. "Don't worry, we'll find him," said mommy. "Let's keep looking." After a while, they found Teddy hiding under the bed. Lily was so happy and gave Teddy a big hug. "I'm so glad you're back," said mommy. "Now you can always play with Teddy again." Lily smiled and hugged Teddy tightly. She didn't have any more tears for Teddy, but she was very happy that Teddy was safe.

stories42M

Once upon a time, there was a little girl named Lily. One day, she found a bottle on the ground. It was a pretty bottle with a shiny cap. She picked it up and showed it to her mom. "Look, Mommy! I found a pretty bottle!" Lily said. Her mom looked at the bottle and noticed it was empty. "That's okay, Lily. We can use it to make some juice," her mom said. Lily was so excited to make juice with her mom. She helped her mom squeeze the juice out of the bottle. They poured it into a bowl and added some water. Then, they drank it all up. Lily's dad came home and saw the bottle. "Wow, that's a nice bottle," he said. "Where did you find it?" Lily told him about finding it outside. Her dad said, "That's great, Lily. It's very useful to find things that make us happy. Let's put it in the kitchen so we can use it again later." And they did. The end.

stories110M

Once upon a time, there was a little girl named Lily. She had a fancy dress that she loved to wear. One day, Lily was playing with her toys when she heard her mom say, "Lily, it's time to clean up your toys." Lily didn't want to stop playing, but she knew she had to listen to her mom. As she picked up her toys, she found a nail on the floor. She picked it up and showed it to her mom. "Look, Mommy! I found a nail!" she said. Her mom smiled and said, "Good job, Lily. Nails are important for building things. Now, let's finish cleaning up your toys." Lily felt proud of herself for helping her mom and for finding the nail. She finished cleaning up her toys and went back to playing with her fancy dress.

this on-chip memory. The PYNQ-Z2's Zynq chip has 630 KB of Block Random Access Memory (BRAM). Storing the model in BRAM is advised to avoid the slow access times of external DDR memory. Using external memory would create a bottleneck and reduce the benefit of hardware acceleration.

To understand this memory limit, we can calculate the space needed for one of the TinyStories language models of 110 million parameters. Each parameter is a 32-bit floating-point number, which takes 4 bytes of memory. The total memory for a model of this size would be approximately 440 megabytes. This amount is much larger than the 630 kilobytes of BRAM available on the chip. Even if we reduce each parameter to 8 bits, or 1 byte, the model would still need 110 megabytes. This shows that it is not possible to store a model of this size in the on-chip BRAM. In Figure 5.4, the sizes of the TinyStories family models are presented.

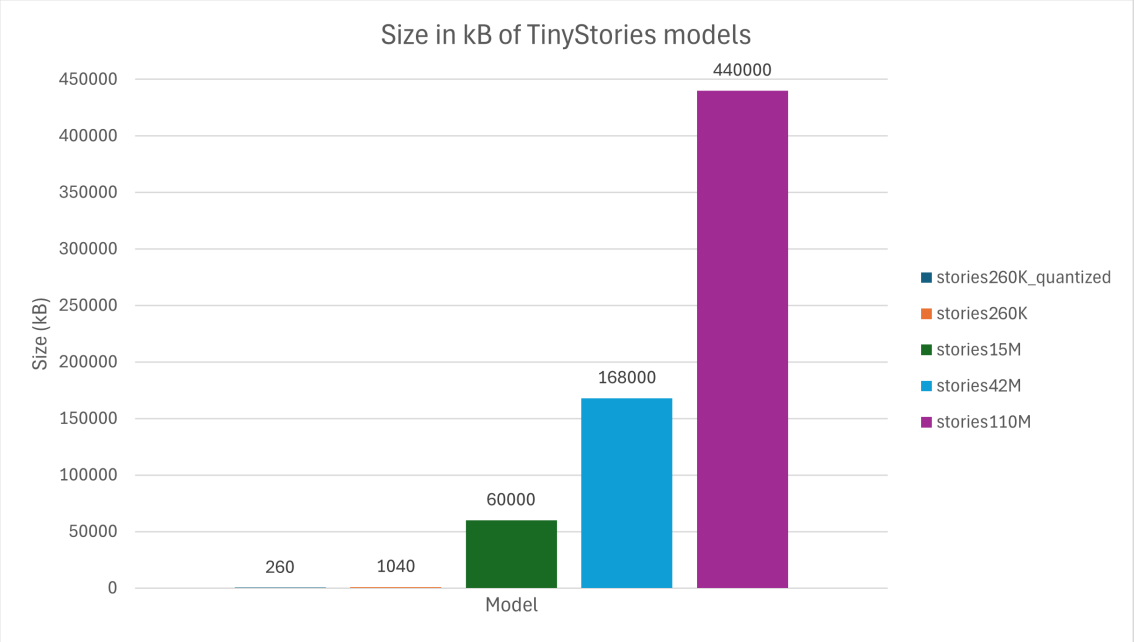


Figure 5.4: Size of TinyStories models.

This hardware limit requires the use of a much smaller model. For this project, a TinyStories model with 260,000 parameters was selected. The memory needed for this model, using 32-bit floating-point numbers, is about 1.04 megabytes. This size is still larger than the available BRAM. However, it is small enough that a significant part of it can fit into the available BRAMs. We can also apply quantization. For example, if we quantize the parameters to 8-bit integers, each parameter would take up 1 byte. The model's size would then be reduced to 260 kilobytes. This size aligns more with the available BRAM in the PYNQ-Z2. This calculation shows why a model of this small scale is the only practical option for an on-chip accelerator on the PYNQ-Z2 board.

5.2 System Architecture

The design method for this project follows a hardware and software co-design approach. The goal is to offload the computationally intensive parts of the language model's inference process to a custom hardware accelerator implemented in the FPGA's PL. This requires the *run.c* file, from the *llama2.c* repository [15], to be split between the PS and PL. The Zynq's PS is used to manage the overall application flow, handle data movement, and control the accelerator. The PL is then used for the forward pass calculations. This section details the general flow of this interaction, the specific mechanisms used for communication, and the design choices that enable the software on the PS to control the hardware on the PL.

In this project, there are two different versions of this interaction implemented, whose flow can be seen in Figure 5.5. Both of these versions are based on the original *run.c* inference engine from the *llama2.c* repository [15].

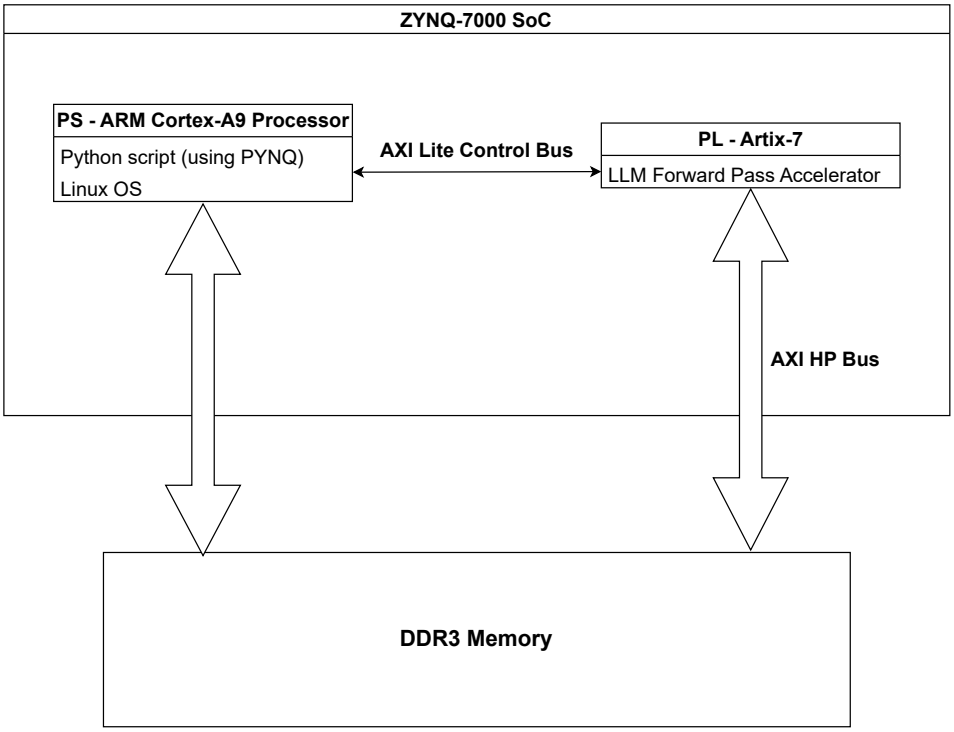


Figure 5.5: PS and PL interaction.

In the first variant implemented, described in Section 5.2.1, the PYNQ framework provides the high-level software infrastructure to manage the software-hardware interaction. The general flow of the application is orchestrated by a Python script running on the Linux operating system within the PS. This script acts as the controller for the entire inference process. It is responsible for initializing the system, programming the PL, managing all data transfers between the PS and PL, and issuing commands to the hardware accelerator. In the second variant implemented, explained in Section 5.2.2, the PYNQ framework is still responsible for initializing the system, programming

the PL, but the remaining tasks of managing the data transfers between the PS and PL and issuing commands to the hardware accelerator are done by a C program.

The need to use Python emerged because no Board Support Package (BSP) was found for the PYNQ-Z2 board, which would allow the creation of an OS with Petalinux with a device tree that reserved the required address space for the hardware. An attempt was made, but some key configurations, such as the IP and COM port, weren't available, and the board was not accessible due to that. This meant that the only way to reserve the address space for the hardware was using the *pynq.Overlay* class.

The first and most essential action performed by the control script is the configuration of the PL. At power-on, the FPGA fabric is not configured with any hardware overlay. Before any hardware acceleration can occur, the custom-designed circuit, known as the accelerator, must be loaded onto it. This is accomplished by programming the PL with a bitstream file, or overlay. The Python script must load this overlay to make the custom hardware logic available for use. This step is a prerequisite for any subsequent operations involving the accelerator. Without loading the overlay, the hardware does not exist from the software's perspective, and any attempt to communicate with it would fail.

As explained, we leverage the PYNQ libraries to abstract FPGA programming and handle communication with hardware. The *pynq.Overlay* class is instantiated with the path to the bitstream file. This single command initiates a series of low-level operations that configure the FPGA fabric. PYNQ also parses an accompanying hardware handoff file, which contains metadata about the hardware design, including the IP blocks it contains, their memory-mapped addresses, and their interrupt connections. By parsing this file, the PYNQ library automatically creates Python objects that correspond to the IP blocks in the overlay. For instance, after loading, the accelerator becomes accessible in the Python script as an attribute of the overlay object. This dynamic mapping of hardware into the Python programming environment is a central feature of the PYNQ framework, as it removes the need for the developer to manually manage memory addresses and register maps.

Once the hardware is configured and accessible, the next critical task is to manage the flow of data between the software and the accelerator. The model's weights, the input prompt tokens, and the output logits must be moved between the PS and the PL. The PYNQ library provides specific tools for this purpose because standard memory allocation in Python is not suitable for hardware communication. Hardware accelerators require access to memory buffers that are physically contiguous. The Linux kernel's virtual memory system does not guarantee that a standard block of memory will occupy a continuous range of physical addresses, which furthermore must correspond to the addresses where the hardware blocks were allocated on the peripheral bus.

To address this, the PYNQ library includes the *pynq.allocate* function. This function requests a block of memory from the operating system that is guaranteed to be physically contiguous. It returns a buffer object that can be used like a standard Python array but also has a *physical_address* attribute. This physical address is essential, as it is the address that the hardware accelerator in the PL will use to access the data. The Python script is responsible for allocating these buffers for all

data that needs to be shared with the hardware, including the model parameters and the input and output tensors for the inference computation.

5.2.1 Python-only Controller

For prototyping purposes, the whole *run.c* file, except the forward pass, was converted to Python. This is because in Python, with PYNQ, there is access to functions like *ip.write(offset, value)* and *ip.read(offset)* that make development more straightforward by allowing a more direct and easy communication with the hardware accelerator.

With the overlay loaded and the memory buffers prepared, the Python script can begin the main inference loop. For each token to be generated, the script first prepares the input data in the allocated buffers. It then initiates the hardware accelerator's operation. This is done by communicating with the accelerator's control registers, which are exposed to the Python script through the IP object created by PYNQ. The *ip.write(offset, value)* method is used to write data to specific register addresses within the accelerator. The script uses this function to pass control signals, such as a *START* signal, to the hardware. It also uses this method to provide the accelerator with the physical addresses of the input and output buffers that it needs to use for the current computation.

Because the PS and the PL operate in parallel, the Python script must have a way to know when the accelerator has finished its task. After starting the accelerator, the software cannot simply proceed, as it needs the result of the hardware's computation. A synchronization mechanism is therefore required. In this project, a simple and effective polling method is used. The hardware accelerator is designed with a status register. When the accelerator completes its computation, it writes a specific value, or a *DONE* flag, to this register. The Python script, after starting the accelerator, enters a loop where it continuously reads the value of this status register using the *ip.read(offset)* method. The loop continues until the script reads the *DONE* flag. This polling mechanism ensures that the software waits for the hardware to finish before it attempts to use the results.

Once the *DONE* flag is detected, the Python script knows that the output data is ready in the corresponding output buffer. It can then access this buffer to retrieve the results of the computation. The script processes this result by selecting the next token based on the output logits, and then prepares the inputs for the next iteration of the inference loop. This cycle of preparing data, starting the accelerator, waiting for completion, and retrieving results continues until the full output sequence has been generated. This entire flow, from loading the hardware to managing the token-by-token generation, is controlled and orchestrated from a single, high-level Python script.

The full execution flow of this Python-only variant can be seen in Figure 5.6.

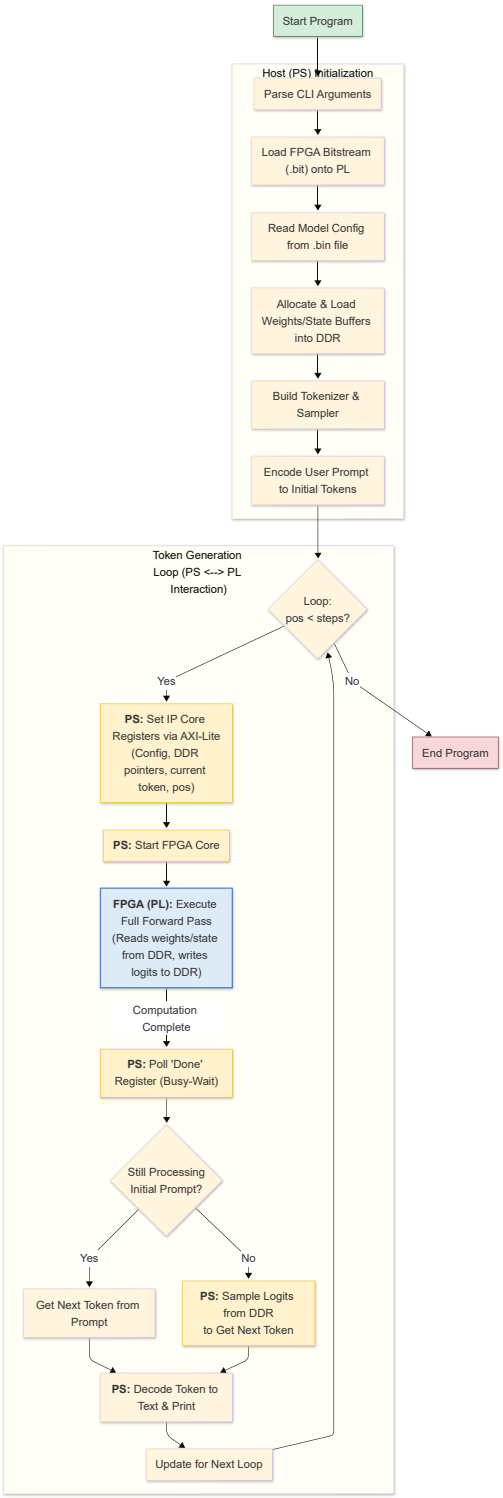


Figure 5.6: Execution flow of Python-only variant.

5.2.2 Hybrid Controller

While controlling the entire process from a single Python script is effective for prototyping, a more optimized approach was to go back to the original *run.c*, from the *llama2.c* repository [15], and add to it the communication with the hardware accelerator, all the while keeping the setup of the overlay in Python. The high-level abstractions of Python, while excellent for rapid development, can introduce noticeable overhead in performance-critical loops that require frequent and repetitive interaction with the hardware. This means that the software's responsibilities are divided into two parts. A Python script for the initial setup and a C program for the main execution and control loop. This design leverages the strengths of each language, using Python for its powerful libraries and ease of system configuration, and using C for its low-level control and higher execution speed.

The Python script handles the task of memory management. It begins by reading the header of the model's checkpoint file to determine its configuration, such as the model's dimension, number of layers, and other architectural details. The Python script then allocates the necessary memory buffers in the external DDR memory using the *pynq.allocate* function. This function is critical because it reserves buffers that are physically contiguous in memory. The Python script allocates separate buffers for the model's weights, the transformer's run state, and other data structures. After allocation, the script loads the model weights from the checkpoint file directly into the newly allocated buffer. This information is then written to a text file, which is read by the C program.

Once all memory is allocated and initialized, the script creates the communication channel for the C program. It retrieves the unique physical memory address of each buffer and the base physical address of the accelerator's memory-mapped control registers. This address information is the key that will allow the C program to find and interact with the hardware and data buffers. The script then writes these addresses, one per line, into a text file. This file serves as a handoff mechanism between the Python and C environments. With the hardware programmed, memory prepared, and the address file written, the Python script's job is complete. It uses Python's *subprocess* module to launch the pre-compiled C program, passing along any command-line arguments intended for it, such as the desired temperature for sampling or the number of tokens to generate.

The C program then takes over complete control of the main inference loop. As a compiled binary, it runs directly on the processor without the overhead of a Python interpreter, which is crucial for the performance of the repetitive loop required for token-by-token generation. The first action in the C program's main function is to read the physical addresses from the addr.txt file. It opens this file, reads each line, and uses converts the text representation of the addresses into unsigned integer variables.

The most critical step in the C program is gaining direct access to the hardware's physical memory. The program, like any user-space application on Linux, operates in a virtual memory space and cannot use physical addresses directly. To overcome this, the program uses memory mapping. This is a standard Linux feature that allows a process to map a device's physical memory addresses into its own virtual address space. The C program achieves this by first opening the */dev/mem* device file in read-write mode. This file represents the entire physical memory of the

system. The program then makes a call to the *mmap* system function. This function takes the physical base address of the hardware accelerator, which was read from the text file created by the Python script, and maps a portion of this physical address space to a virtual address within the C program's memory. The *mmap* function returns a pointer. This pointer, which is a virtual address, now points to the beginning of the accelerator's memory-mapped registers. From this point forward, the C program can directly read from and write to the hardware's registers by dereferencing this pointer with different offsets.

With this direct access established, the C program can control the hardware with high efficiency. It uses pointer arithmetic to access specific control registers at the offsets defined in the header files. These control registers are defined by Vitis HLS during synthesis. For example, to start the accelerator, it writes the value 1, representing a *START* command, to the control register. To inform the accelerator where the input data is and where the output results should be stored, it writes the physical addresses of the relevant buffers to the accelerator's data pointer registers.

After initiating the accelerator, the C program must wait for it to complete its computation. This synchronization is achieved by polling a status register. The hardware is designed to set a *DONE* flag, corresponding to bit 1 of the control register, to high when it has finished. The C program enters a *while* loop, repeatedly reading the value of the control register and using a bitwise *AND* operation to check if the *DONE* bit is set. The loop continues to poll the register until this bit becomes high. Once the bit is set, the loop terminates, and the C program knows that the output data is valid and ready in the output buffer. It then proceeds with the rest of the inference logic, such as sampling the next token from the output logits, and prepares for the next iteration of the loop. This hybrid architecture successfully combines the high-level convenience of Python for system setup with the low-level performance and control of C for the critical execution loop.

The full execution flow of this hybrid variant can be seen in Figure 5.7.

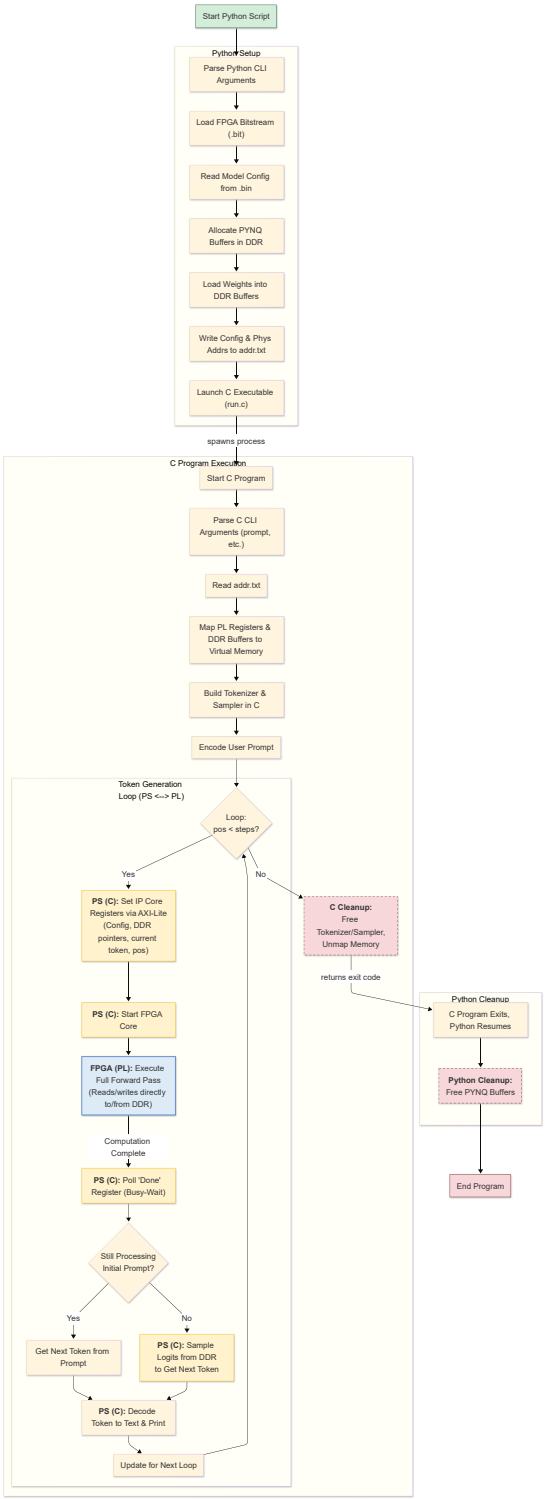


Figure 5.7: Execution flow of hybrid variant.

5.2.3 Software-Hardware Partitioning

The partitioning of software and hardware tasks is central to designing FPGA-based accelerators. The aim is to combine the flexibility of a general-purpose processor with the parallel processing ability of hardware logic. In this project, this method is applied to Llama 2 inference on the PYNQ-Z2 platform. The most computationally intensive kernels of the inference algorithm are offloaded to the FPGA's PL, while the ARM processor in the PS handles the application flow, data transfers, and control logic.

The baseline *llama2.c* implementation was profiled using the GNU Profiler, *gprof*, during an inference run. The profiler measures the execution time of each function and highlights the main computational bottlenecks. The results, shown in Table 5.1, identify the functions that dominate runtime.

Table 5.1: GProf Profiling Results for Llama 2 Inference.

Function Name	**% Time**	**Cumulative % Time**
`matmul`	68.00%	68.00%
`multihead_attention`	24.00%	92.00%
`softmax`	8.00%	100.00%
`rmsnorm`	0.00%	100.00%
`residual_connection`	0.00%	100.00%

The profiling shows that most of the runtime comes from a small number of functions used in the transformer forward pass. The *matmul* function, which performs matrix multiplications, accounts for 68% of the time. The *multihead_attention* function takes 24%, and the *softmax* function 8%. Together, these functions consume almost all the computation required to produce a token. Other operations, such as *rmsnorm* and *residual_connection*, run frequently but are too fast to register in the profiler as significant. These results were the basis for exploring partitioning options.

One possible option is to implement the entire inference loop in hardware, including the forward pass, sampling logic, vocabulary lookup, and state management. On the PYNQ-Z2, this is not feasible. The control flow in sampling is sequential and data-dependent, making it better suited for the PS. In addition, the logic and memory required would exceed the 13,300 logic slices and 630 KB of BRAM on the Zynq-7000 SoC.

A second option is to accelerate only the *matmul* function. Since profiling shows it is the largest bottleneck, this could be effective. However, matrix multiplications in the transformer are not uniform. They are used for Query-Key-Value (QKV) projections, attention output, and the feed-forward network, with different input and output sizes. A hardware accelerator for this would need to be either a set of optimized, highly specific, separate kernels specialized for different dimensions, which is very resource-heavy and adds complexity, or one general matmul, that would reduce complexity but not be as optimized. In both of these implementations, data movement also becomes an issue, where the PS would need to repeatedly transfer weight and activation matrices to the PL and fetch results, adding high communication overhead that could reduce the benefits.

The strategy chosen in this project is to offload the entire *forward* function to a single hardware accelerator. This captures all major bottlenecks in one block. The PS only calls the accelerator once per token, which reduces communication overhead compared to a *matmul*-only design. The hardware block can pipeline operations and use on-chip BRAM for intermediate values, avoiding slow transfers to external DDR memory. This approach balances performance and design complexity, producing a system that accelerates the critical inference stage efficiently.

5.2.4 Hardware Optimizations

The acceleration of a transformer inference kernel on an FPGA requires a systematic adaptation process. A C implementation developed for sequential CPU execution must be restructured to expose parallelism and to reduce memory bottlenecks, which are the primary barriers to efficient hardware execution. In the baseline design, the kernel depends heavily on off-chip DDR memory. The top-level forward function receives all weight matrices and state vectors as pointers to external memory, which forces every read and write operation to traverse the DDR interface. These transactions incur high and variable latency and consume limited memory bandwidth. Since they appear within the innermost loops of the matrix multiplication operations and the attention mechanisms, external memory becomes the principal factor constraining throughput. Even when burst transactions are used, the repeated fine-grained DDR accesses that occur during inner-loop execution prevent efficient pipeline utilization.

The optimized design addresses this bottleneck by relocating the majority of model data into BRAMs. State vectors, intermediate activations, and parameter matrices are declared as static arrays inside the top-level HLS function, which causes the synthesis tool to allocate these structures in BRAM. These include the hidden state vector updated across layers, the projection matrices used to compute queries, keys, and values for the attention mechanism, the per-layer normalization weights, the feedforward network matrices, and the self-attention caches. Storing these structures in BRAM ensures that the vast majority of data accesses occur at low latency, in contrast to the costly transactions required by DDR. This change allows the matrix multiplication units, normalization routines, and other computational blocks to operate at higher throughput, since pipeline stalls due to memory wait states are largely eliminated.

To further exploit locality, the design introduces an explicit preloading strategy. At the beginning of each transformer layer, the weight matrices required by that layer are transferred in bulk from DDR into their BRAM-resident counterparts. In the attention sub-block, the normalization weights and the projection matrices for queries, keys, values, and outputs are staged in local memory. In the feedforward sub-block, the normalization weights and the three linear transformation matrices are preloaded before computation begins. This creates a two-phase execution pattern. First, a preload phase performs coarse-grained burst transfers from DDR to BRAM. Second, the compute phase executes all subsequent operations using only BRAM-resident data. The effect of this restructuring is that the cost of DDR latency is incurred only once per parameter block, while the compute-intensive loops of the transformer operate exclusively from on-chip memory.

By amortizing the DDR access cost over entire layers, the design avoids memory bottlenecks in the inner loops and achieves higher arithmetic unit utilization.

The self-attention key and value caches are optimized using the same principle. In the baseline design, attention score calculations repeatedly fetch cache entries from DDR during every timestep. In the optimized version, the segment of the key and value caches that corresponds to the current sequence length is copied into BRAM at the start of each layer's execution. When processing the current token, the new key and value vectors are written both into the BRAM-resident caches and back to DDR, ensuring global consistency. The subsequent attention operations-dot products between query vectors and cached keys, followed by weighted aggregations of the cached values—are then performed entirely on the BRAM copies. This eliminates repeated fine-grained DDR transactions within the innermost loops of the attention mechanism, where the computation is most sensitive to latency.

The calculation of Rotary Positional Encoding (RoPE) was also substantially optimized. In the baseline implementation, positional encodings are computed by repeatedly invoking floating-point power, sine, and cosine functions. In the optimized design, these values are precomputed once and stored in a BRAM-based lookup table. During the first invocation of the kernel, the helper function generates all required sine and cosine values for the supported sequence length and head dimensions and writes them to on-chip memory. Subsequent executions apply positional encodings by retrieving values directly from the lookup table, replacing expensive arithmetic operations with constant-time memory accesses. This reduces resource utilization and avoids the inclusion of complex trigonometric operators in the datapath.

Performance is further improved through the use of synthesis directives that expose parallelism. Loop pipelining is applied to computation-intensive routines such as matrix multiplications and dot-product accumulations. Pipelining allows a new iteration of a loop to begin before the previous one has finished, possibly reducing the initiation interval of the loop to nearly one cycle and significantly increasing throughput. Loop unrolling is applied in cases where the number of iterations is small and fixed, such as the operations across head dimensions in the attention mechanism. Unrolling duplicates loop bodies in hardware, enabling multiple operations to be executed in parallel. This strategy increases parallelism at the cost of additional resource consumption, a trade-off that was carefully balanced against the available logic and DSP blocks of the PYNQ-Z2 board. Together, pipelining and unrolling enable the hardware to exploit both fine-grained and coarse-grained parallelism present in the transformer architecture.

Through these combined optimizations, the kernel is restructured from a sequential, memory-bound software implementation into a parallel and memory-efficient hardware design. The reliance on DDR is reduced to coarse-grained parameter transfers and cache synchronization, while nearly all computation-intensive operations operate exclusively on BRAM-resident data. Preloading improves data locality, cache management eliminates redundant memory accesses, precomputed positional encodings reduce arithmetic overhead, and synthesis directives maximize concurrency in the datapath. These transformations collectively enable efficient inference execution on FPGA hardware.

In Table 5.2, a summary of all the optimizations done and their purpose is presented.

Table 5.2: Summary of Key Hardware Optimizations.

Optimization	Rationale	Expected Impact
Data Relocation (DDR to BRAM)	Reduce memory access latency by moving frequently accessed data closer to the processing elements.	Higher throughput, lower power consumption, and more predictable performance.
RoPE Precomputation	Replace expensive, real-time arithmetic (sine, cosine) with a simple, low-latency memory lookup.	Reduced DSP and logic resource utilization, and decreased latency per layer.
Loop Pipelining & Unrolling	Exploit both fine-grained and coarse-grained parallelism inherent in the transformer architecture.	Significantly increased data processing throughput and better utilization of hardware resources.

5.2.4.1 Quantization

While the previous optimizations improved performance, the model still used 32-bit floating-point numbers. Floating-point arithmetic is costly on FPGAs, requiring more logic resources and power. The second optimization phase introduced quantization to address this issue.

Quantization is the process of converting a high-precision number, like a 32-bit float, into a lower-precision representation, such as an 8-bit integer. This technique is beneficial for hardware acceleration. An 8-bit integer uses four times less memory than a 32-bit float, which allows more data to fit into the limited on-chip BRAM and reduces data transfer volumes from DDR. Additionally, integer multipliers are smaller, faster, and more power-efficient than their floating-point counterparts. This allows for more parallel processing units on the FPGA, leading to higher throughput.

The first step was to use the *export.py* file to quantize the original 32-bit floating-point TinyStories 260K parameter model to an 8-bit integer model. This model was then used on the adapted 8-bit integer software and hardware programs.

A group-wise, on-the-fly quantization scheme was implemented. The code defines a new type, *q_t*, as an 8-bit integer, using *ap_int<8>*. A 32-bit integer type, *acc_t*, is used for accumulation during matrix multiplication to prevent overflow. A function handles the conversion of activations. Instead of using a single scaling factor for a whole vector, this function divides the vector into smaller groups. For each group, it finds the maximum absolute value and calculates a per-group scaling factor. This factor maps the floating-point range of that group to the 8-bit integer range.

The standard *matmul* function was replaced with *matmul_quantized*. This new function performs the core computation using integer arithmetic. It multiplies the 8-bit quantized activation with the 8-bit quantized weight and accumulates the product in a 32-bit integer. After the accumulation is finished, it dequantizes the result back to a floating-point number by multiplying the

integer sum with the scaling factors from both the activation and the weight groups. This step returns the output to its correct floating-point value.

This method results in a mixed-precision system. The main state of the transformer remains in floating-point to maintain accuracy during additions and other operations. However, the matrix multiplications, which are the most computationally demanding parts, are performed using efficient 8-bit integer logic. This on-the-fly quantization of activations before each *matmul* call provides a good balance between hardware efficiency and model accuracy.

Chapter 6

Results

This chapter presents the results of the implementation and evaluation of the hardware-accelerated inference for the TinyStories 260K parameters model [4] on the PYNQ-Z2 FPGA platform. The evaluation is structured to answer the research questions posed in Chapter 1 by analyzing functional correctness, resource utilization, and execution performance across the different implementations mentioned in Chapter 5.

First, the experimental setup is detailed, outlining the hardware platforms, software configurations, and performance metrics used. Next, the functional validation of the hardware accelerator is presented. Following this, the performance of the software-only baselines on both an x64 platform and the PYNQ-Z2's ARM processor is established. Subsequently, the synthesis results and execution performance of the FPGA-based hardware accelerator are detailed for both 32-bit floating-point and 8-bit integer quantized versions. Finally, a comparative analysis is conducted to quantify the speedup and efficiency gains achieved by the hardware acceleration approach.

6.1 Experimental Setup

To ensure a systematic and reproducible evaluation, a well-defined experimental setup was established. This section describes the hardware platforms, the language model, and the metrics used for evaluation.

The experiments were conducted on two distinct hardware platforms. A standard desktop computer was used as the x64 baseline platform to establish an upper-bound performance reference, featuring an Intel Core i5-10300H @ 2.50GHz CPU, 16 GB DDR4 of Random Access Memory (RAM). The edge platform was the PYNQ-Z2 board, which was used for both the software baseline on its ARM core and the hardware-accelerated implementation. This board contains a Xilinx Zynq-7000 XC7Z020-1CLG400C SoC, which includes a dual-core ARM Cortex-A9 processor running at 650MHz and an Artix-7 FPGA fabric, along with 512 MB of shared DDR3 RAM.

The language model used for all experiments was the TinyStories 260K parameter model [4]. Two versions of this model were tested: a 32-bit floating-point version and an 8-bit integer

quantized version. To ensure a fair comparison, a consistent set of prompts was used across all tests. The primary prompt for performance measurement was *"Once upon a time"*, and for each run, 256 tokens were generated.

The evaluation was based on a set of key metrics. Functional validation was performed by comparing the output of the hardware accelerator against the software implementation on a token-by-token basis. FPGA resource utilization was measured by the amount of Look Up Tables (LUTs), Flip-Flops (FFs), Block Random Access Memory (BRAM), and Digital Signal Processors (DSPs) consumed by the hardware design, as reported by Vitis HLS synthesis. Performance was measured in terms of execution time, defined as the average latency to generate a single token (s/token), and throughput, measured as the number of tokens generated per second (tokens/s).

6.2 Functional Validation

Before evaluating performance, it was essential to verify that the hardware accelerators produced correct results. The outputs from the hardware implementations, for both 32-bit floating-point and 8-bit integer quantized versions, were compared against their respective software baseline counterparts running on the ARM processor and the standard desktop computer.

On the Python-only controller version, with the 32-bit floating-point model and the baseline hardware accelerator, the generated token sequence is identical to the software baseline, as can be seen in Figure 6.1, confirming the logical correctness of both the baseline hardware accelerator and the controller. In the optimized hardware accelerator, the generated token sequence was incoherent, as all the logits coming out of the hardware were being read in software as 0.020. In the 8-bit integer quantized version, a successful run could not be executed.

Figure 6.1: Python-only controller generated token sequence example.

stories260K

Once upon a time, there was a little girl named Lily. She loved to chose cookies and jump on the shovel. One day, she met a big fox named Whiskers. While they were twins, Lily said, "Wow, your show is so much fun!" Lily felt scared of the fox and said, "Don't worry, I'll ask What I want to mers?" Whiskers said, "Why is that forest. I need to check bottles and enjoy things." And they went to their house, and they all played together every day.Once upon a time, there was a big enormous box with a feather. The boy loved to play in the park

In the hybrid version, the generated token sequence from the 32-bit floating-point model with the baseline hardware accelerator was similar to the sequence generated by the 32-bit floating-point software baseline, as can be seen in Figure 6.2, confirming the logical correctness of both the baseline hardware accelerator and the hybrid controller. In the optimized hardware accelerator, the generated token sequence was incoherent, as all the logits coming out of the hardware were

being read in software as 0.020. In the 8-bit integer quantized version, a successful run was executed, but the same problem as the optimized version was present here.

Figure 6.2: Hybrid controller generated token sequence example.

stories260K

Once upon a time, there was a little girl named Lily. She loved to play with her friends, the boy. One day, Lily found a small box that he was fixing it. She was happy because she didn't know how to feel good. But then, something unexpected happened. The boy started to smile. Lily's friend Johnny saw the big tree and said, "I can help you, Johnny! We can'thave it." But Lily didn't like the boy's snack. Johnie took Lily to the tree and picked it up in a new place. When they opened the tree, the tree broke into broken tre

6.3 Baseline Performance

As mentioned in Section 6.1, performance baselines were established by running the *run.c* inference code, from the *llama2.c* repository [15], in software on a standard desktop computer and on the PYNQ-Z2's ARM processor. In Table 6.1 and Table 6.2, the latency and throughput of these runs are presented.

Table 6.1: Performance of Software-only Inference on Desktop Computer.

Model	Latency (ms/token)	Throughput (tokens/s)
`260K FP32`	0.56	1795.77
`260K INT8`	0.31	3187.50

Table 6.2: Performance of Software-only Inference on PYNQ-Z2's ARM processor.

Model	Latency (ms/token)	Throughput (tokens/s)
`260K FP32`	6.68	149.74
`260K INT8`	5.28	189.31

As evidenced by the measurements in Table 6.1 and Table 6.2, quantization improved the throughput and latency of the inference. This increase was more pronounced in the desktop computer, with an increase of approximately 44% in throughput, than in the PYNQ-Z2, where the increase in throughput was only approximately 21%.

6.4 Hardware Accelerators

This section details the synthesis results and the final execution performance of the hardware accelerators executed on the PYNQ-Z2's PL.

6.4.1 Synthesis and Resource Utilization

In Table 6.3, the FPGA resources required for the different forward pass accelerators after synthesis with Vitis HLS are shown. The percentages are relative to the total resources available on the PYNQ-Z2.

Table 6.3: Hardware Accelerators resource usage on PYNQ-Z2.

Accelerator	BRAM %	DSP %	FF %	LUT %
`Baseline`	1	32	23	60
`Optimized`	77	62	32	53
`Quantized`	23	62	41	97

The synthesis results reveal a clear narrative of design trade-offs. The shift from the Baseline to the Optimized version shows an increase in BRAM utilization from 1% to 81%. This is a direct and intended consequence of the primary optimization strategy of moving model parameters and intermediate activations from external DDR into on-chip BRAM to eliminate memory bottlenecks. This change is accompanied by a doubling of DSP usage and significant increases in FF and LUT resources, which reflects the implementation of a more complex and parallel datapath needed to process data on-chip efficiently.

The transition from the Optimized to the Quantized version demonstrates the benefits of lower-precision arithmetic. BRAM usage is reduced from 81% to 23%, a nearly four-fold decrease that directly corresponds to the smaller data size of 8-bit integers compared to 32-bit floats. DSP usage increases to 62% because the quantized design applies more parallelization and arithmetic optimization. The LUT utilization for the Quantized version reaches 97%, indicating that the design consumes nearly all available logic resources. This increase is attributed to the additional logic required for the on-the-fly quantization and dequantization steps, as well as the control logic for managing the 8-bit datapath.

6.4.2 Performance Measurement

In Table 6.4, the results of the performance measurement for the Python-only controller are presented. As can be seen, the optimized version brought a **2.4x speedup** when compared to the baseline version, showing the power of the optimizations implemented.

Table 6.4: Python-only controller Performance.

Accelerator	Latency (ms/token)	Throughput (tokens/s)
`Baseline`	265.97	3.76
`Optimized`	110.50	9.05

In Table 6.5, the results of the performance measurement for the hybrid controller are presented. As can be seen, the optimized version brought a **3x speedup** when compared to the baseline version, again showing the power of the optimizations implemented.

Table 6.5: Hybrid controller Performance.

Accelerator	Latency (ms/token)	Throughput (tokens/s)
`Baseline`	263.15	3.80
`Optimized`	85.98	11.63
`Quantized`	303.95	3.29

6.5 Result Analysis and Comparison

The experimental results, shown in Table 6.6, provide a clear picture of the trade-offs and challenges involved in accelerating LLM inference on a resource-constrained edge device like the PYNQ-Z2. This section analyzes the performance of the various software and hardware implementations, contextualizes and compares the results, and discusses the likely causes for the observed outcomes.

Table 6.6: Latency and throughput comparison across implementations.

Platform	Controller	Accelerator	Precision	Latency (ms/-token)	Throughput (tokens/s)
Desktop CPU	Software-only	N/A	FP32	0.56	1795.77
Desktop CPU	Software-only	N/A	INT8	0.31	3187.50
PYNQ ARM CPU	Software-only	N/A	FP32	6.68	149.74
PYNQ ARM CPU	Software-only	N/A	INT8	5.28	189.31
PYNQ-Z2 FPGA	Python-only	Baseline	FP32	265.97	3.76
PYNQ-Z2 FPGA	Python-only	Optimized	FP32	110.50	9.05
PYNQ-Z2 FPGA	Python-only	Quantized	INT8	N/A	N/A
PYNQ-Z2 FPGA	Hybrid	Baseline	FP32	263.15	3.80
PYNQ-Z2 FPGA	Hybrid	Optimized	FP32	85.98	11.63
PYNQ-Z2 FPGA	Hybrid	Quantized	INT8	303.95	3.29

The hybrid controller with the optimized hardware accelerator reached a throughput of **11.63 tokens/s**. In comparison, the baseline 32-bit floating-point software on the ARM processor achieved **149.74 tokens/s**, more than an order of magnitude higher. The ARM-only execution remains faster because the PYNQ-Z2 has limited resources, with **13,300 logic slices**, **630 KB of BRAM**, and **220 DSP slices**. These constraints reduce the degree of parallelism available in hardware. Optimizations such as pipelining and memory partitioning improved performance by about **3**$\times$ relative to the baseline hardware design, but they could not compensate for the fabric size and the PS to PL communication overhead.

The optimized hardware accelerator did not produce valid output, unlike the baseline version. The optimized design increased BRAM utilization from 1% to **77%** by preloading parameters and intermediate values. After this change, all logits collapsed to a constant value of 0.020. The most likely causes theorized were:

- **Data Coherency:** Errors in DDR to BRAM transfers.
- **Addressing Logic:** Incorrect memory indexing in BRAM arrays.
- **Timing Closure:** Violations introduced by higher resource usage.

The timing closure issue was excluded, as that was only possible if the worst negative slack in Vivado was negative. In the case of our optimized hardware accelerator, it was positive.

The attempt to introduce 8-bit integer quantization to the hardware accelerator proved to be counterproductive. The chosen implementation performed on-the-fly quantization of activations and dequantization of results for every matrix multiplication, keeping the main transformer state in 32-bit floating-point format. While this mixed-precision approach can balance accuracy and performance, its implementation in hardware was extremely costly. The logic required to perform these constant conversions between floating-point and integer formats consumed a lot of resources, as evidenced by the 97% LUT utilization for the quantized accelerator. This left few to no logic resources available for the actual performance optimizations, such as deeper pipelines or wider parallel compute units. The hardware was dedicating most of its fabric to data type conversion rather than efficient computation. This led to the quantized accelerator being slower than even the baseline accelerator. This contrasts sharply with the software-only results, where the 8-bit quantized model ran approximately 21% faster than its floating-point counterpart on the ARM processor. This shows that quantization is a valid optimization strategy, but its hardware implementation must be designed to avoid the kind of conversion overhead that this version of the accelerator has.

Chapter 7

Conclusion

This dissertation investigated the acceleration of SLM inference on resource-constrained edge devices using the hardware capabilities of FPGAs. The central problem was the high computational and energy requirements of language models, which limit their execution to cloud servers. This constraint prevents their use in real-time, latency-sensitive edge applications. The primary hypothesis was that offloading the main computational kernels of the LLM inference stage to an FPGAs-based hardware accelerator would yield performance and energy efficiency improvements compared to software-only execution on an embedded processor.

7.1 Summary of Work

The project used a hardware-software co-design methodology, starting with a C implementation of the Llama 2 architecture [15]. The PYNQ-Z2 board was selected as the target implementation platform. The TinyStories model [4] with 260,000 parameters was selected to fit within the board's on-chip memory limitations.

Software profiling identified the transformer's forward pass, which includes matrix multiplication, attention, and normalization, as the main computational bottleneck. The core of this work was therefore the design of an HLS-based hardware accelerator for this function.

Two system architectures were developed to manage the interaction between the ARM PS and the FPGA's PL. A Python-only controller was used for prototyping, and a hybrid controller combined a Python setup script with a C program for the control loop.

The hardware accelerator was implemented in three versions. A baseline 32-bit floating-point version was a direct HLS port of the C code. An optimized FP32 version moved data from external DDR to on-chip BRAMs to reduce memory bottlenecks. An 8-bit integer quantized version was designed to reduce the memory footprint and use integer arithmetic.

7.2 Discussion of Findings & Research Questions

The experimental results provided insights into hardware acceleration on edge devices and addressed the initial research questions, presented in Section 1.4.

Q1: How to implement LLMs in FPGAs? This work demonstrated an HLS-based implementation pathway. The process involves profiling a software model to identify bottlenecks, partitioning the application, and designing a hardware accelerator for the identified kernels. The hybrid controller architecture was an effective model, using Python for system initialization and C for low-level hardware interaction.

Q2: What phases of the inference step are more beneficial to execute on FPGAs? The profiling results showed that the forward pass is the primary phase to accelerate. Offloading the entire function as a single hardware kernel was an effective strategy that minimized communication overhead between the PS and PL, which was a significant performance factor.

Q3: How to reduce energy consumption in the inference step using FPGAs while minimizing performance loss? The primary method investigated to improve efficiency was quantization. However, the mixed-precision implementation strategy proved to be inefficient for hardware. The logic required for on-the-fly data type conversions between floating-point and integer representations consumed 97% of the available LUTs, leaving few resources for computational acceleration. This overhead resulted in the quantized accelerator being the slowest hardware version, indicating that an integer-only datapath is required for an effective low-power implementation. While direct power measurements were not conducted, the high resource utilization and poor performance of the quantized version suggest that this approach did not reduce energy consumption and, in fact, incurred a significant performance loss.

Q4: How much energy can we save when executing the inference step in FPGAs when compared to inference on other types of devices?

A quantitative answer regarding energy savings was not possible, as power measurement was outside the scope of the experimental setup. Instead, a performance comparison was made, and the results contradicted the initial hypothesis. The highest-performing hardware implementation achieved a throughput of 11.63 tokens/s, which was significantly slower than the software-only baseline on the PYNQ-Z2's ARM core that reached 149.74 tokens/s. This outcome is a direct consequence of the transformer's autoregressive and sequential nature, a core challenge mentioned at the outset of this work. LLM inference generates one token at a time, where each new token depends on the previous one. This creates a fundamental sequential bottleneck. While the operations within a single token's forward pass can be parallelized, the overall process remains serial. The ARM Cortex-A9 processor is highly optimized for fast sequential execution. To outperform it, an FPGA must leverage massive parallelism to make the computation of a single token drastically faster. However, the hardware constraints of the PYNQ-Z2, with its limited logic, DSP, and BRAM resources, restricted the degree of achievable parallelism. Ultimately, the small-scale parallelism implemented on the FPGA was insufficient to overcome the high sequential execution speed of the ARM processor. The hardware accelerator could not make each step in the sequential

chain fast enough to beat the software baseline.

Several findings emerged from this outcome. Hardware constraints were the primary factor. The PYNQ-Z2's limited logic, DSP, and BRAM resources restricted the achievable hardware parallelism. The ARM Cortex-A9's sequential execution speed surpassed the small-scale parallelism implemented on the FPGA. The cost of the quantization implementation was also a factor. The mixed-precision strategy, while functionally correct, was inefficient. The logic for data type conversions consumed 97% of available LUTs, leaving few resources for acceleration. This resulted in the quantized accelerator being the slowest hardware version, which indicates that an integer-only datapath is required for an effective implementation. Finally, the project highlighted the debugging complexity of HLS. The optimized 32-bit floating-point hardware failed functional validation, likely due to data coherency or addressing issues, which represents a challenge in FPGA development.

7.3 Limitations and Future Work

This dissertation, while not achieving a performance speedup, successfully identified the critical challenges and trade-offs of this domain. The limitations of this work directly inform paths for future research.

The foremost limitation is the sequential bottleneck inherent to autoregressive models. Any accelerator's success depends on its ability to compute each token's forward pass so rapidly that it overcomes the serial nature of the overall task. The hardware platform was a primary constraint that compounded this issue. The PYNQ-Z2's modest resources prevented the implementation of a sufficiently parallel datapath, which was necessary to overcome the ARM processor's efficient sequential performance. Furthermore, the mixed-precision quantization strategy was unsuitable for hardware. An offline, full-model quantization strategy with an integer-only datapath is required for performance gains.

These limitations suggest several directions for future work. The immediate step is to resolve the functional error in the optimized hardware accelerator that prevented it from generating coherent output. Following that, a critical next step is to port the design to a larger FPGA, such as a Xilinx Alveo or Versal board. A more capable platform would enable the deeper pipelines and greater parallelism necessary to properly test the hypothesis. Future work should also redesign the accelerator for a pure 8-bit integer pipeline to eliminate the conversion logic overhead that hindered the quantized version. Finally, a formal power consumption analysis should be conducted to quantify the tokens-per-watt metric, as the FPGA may still offer higher energy efficiency even if performance is lower.

In conclusion, this research is a case study of language model inference implementation on edge FPGAs. It demonstrates that achieving performance gains over embedded CPUs with FPGAs is a challenge where hardware resources, system architecture, and implementation strategy are determining factors. Despite the negative performance results, valuable findings were still identified, contributing to the field by outlining a path for future research.

References

[1] Muhammad Adil, Houbing Song, Muhammad Khurram Khan, Ahmed Farouk, and Zhanpeng Jin. 5g/6g-enabled metaverse technologies: Taxonomy, applications, and open security challenges with future research directions. *Journal of Network and Computer Applications*, 223:103828, 2024.

[2] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. Understanding the potential of fpga-based spatial acceleration for large language model inference. *ACM Trans. Reconfigurable Technol. Syst.*, April 2024. Just Accepted.

[3] Krishna Teja Chitty-Venkata, Sparsh Mittal, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. A survey of techniques for optimizing transformer inference. *Journal of Systems Architecture*, 144:102990, 2023.

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

[5] Othmane Friha, Mohamed Amine Ferrag, Burak Kantarci, Burak Cakmak, Arda Ozgun, and Nassira Ghoualmi-Zine. Llm-based edge intelligence: A comprehensive survey on architectures, applications, security and trustworthiness. *IEEE Open Journal of the Communications Society*, 5:5799–5856, 2024.

[6] Georgi Gerganov. llama.cpp: Llm inference in c/c++, 2023.

[7] Neal R. Haddaway, Matthew J. Page, Chris C. Pritchard, and Luke A. McGuinness. Prisma2020: An r package and shiny app for producing prisma 2020-compliant flow diagrams, with interactivity for optimised digital transparency and open synthesis. *Campbell Systematic Reviews*, 18(2):e1230, 2022.

[8] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli. Secda: Efficient hardware/software co-design of fpga-based dnn accelerators for edge inference. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 33–43, 2021.

[9] Jude Haris, Rappy Saha, Wenhao Hu, and José Cano. Designing efficient llm accelerators for edge devices. *arXiv preprint arXiv:2408.00462*, 2024.

[10] Andy He, Darren Key, Mason Bulling, Andrew Chang, Skyler Shapiro, and Everett Lee. Hlstransform: Energy-efficient llama 2 inference on fpgas via high level synthesis. *arXiv preprint arXiv:2405.00738*, 2024.

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[12] Milan Jain, Nicolas Bohm Agostini, Sayan Ghosh, and Antonino Tumeo. Analyzing inference workloads for spatiotemporal modeling. *Future Generation Computer Systems*, 163:107513, 2025.

[13] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Beom Jin Kang, Hae In Lee, Seok Kyu Yoon, Young Chan Kim, Sang Beom Jeong, Seong Jun O, and Hyun Kim. A survey of fpga and asic designs for transformer inference acceleration and optimization. *Journal of Systems Architecture*, 155:103247, 2024.

[15] Andrej Karpathy. llama2.c, 2023.

[16] David Kasperek, Pawel Antonowicz, Marek Baranowski, Marta Sokolowska, and Michal Podpora. Comparison of the usability of apple m2 and m1 processors for various machine learning tasks. *Sensors*, 23(12), 2023.

[17] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*, 2024.

[18] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, and Song Han. Tiny machine learning: Progress and futures [feature]. *IEEE Circuits and Systems Magazine*, 23(3):8–34, 2023.

[19] Zheng Lin, Guanqiao Qu, Qiyuan Chen, Xianhao Chen, Zhe Chen, and Kaibin Huang. Pushing large language models to the 6g edge: Vision, challenges, and opportunities. *arXiv preprint arXiv:2309.16739*, 2023.

[20] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[21] Matthew J Page, Joanne E Mckenzie, Patrick M Bossuyt, Isabelle Boutron, Tammy C Hoffmann, Cynthia D Mulrow, Larissa Shamseer, Jennifer M Tetzlaff, Elie A Akl, Sue E Brennan, and et al. The prisma 2020 statement: an updated guideline for reporting systematic reviews. *BMJ*, page n71, 2021.

[22] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2023.

[23] Zeinab Seifoori, Zahra Ebrahimi, Behnam Khaleghi, and Hossein Asadi. Chapter seven - introduction to emerging sram-based fpga architectures in dark silicon era. In Ali R. Hurson and Hamid Sarbazi-Azad, editors, *Dark Silicon and Future On-chip Systems*, volume 110 of *Advances in Computers*, pages 259–294. Elsevier, 2018.

[24] Liam Seymour, Basar Kutukcu, and Sabur Baidya. Large language models on small resource-constrained systems: Performance characterization, analysis and trade-offs, 2024.

[25] Rasoul Shafipour, David Harrison, Maxwell Horton, Jeffrey Marker, Houman Bedayat, Sachin Mehta, Mohammad Rastegari, Mahyar Najibi, and Saman Naderiparizi. Seedlm: Compressing llm weights into seeds of pseudo-random generators. *arXiv preprint arXiv:2410.10714*, 2024.

[26] Xuan Shen, Zhaoyang Han, Lei Lu, Zhenglun Kong, Peiyan Dong, Zhengang Li, Yanyue Xie, Chao Wu, Miriam Leeser, Pu Zhao, Xue Lin, and Yanzhi Wang. Hotaq: Hardware oriented token adaptive quantization for large language models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2024.

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[28] Shamsher Ullah, Jianqiang Li, Jie Chen, Ikram Ali, Salabat Khan, Abdul Ahad, Farhan Ullah, and Victor C. M. Leung. A survey on emerging trends and applications of 5g and 6g to healthcare environments. *ACM Comput. Surv.*, 57(4), December 2024.

[29] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[30] Han Xu, Yutong Li, and Shihao Ji. Llamaf: An efficient llama2 architecture accelerator on embedded fpgas. *arXiv preprint arXiv:2409.11424*, 2024.

[31] Han Xu, Xingyuan Wang, and Shihao Ji. Towards energy-efficient llama2 architecture on embedded fpgas. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 5570–5571, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Bo Yang, Xuelin Cao, Kai Xiong, Chau Yuen, Yong Liang Guan, Supeng Leng, Lijun Qian, and Zhu Han. Edge intelligence for autonomous driving in 6g wireless system: Design challenges and solutions. *IEEE Wireless Communications*, 28(2):40–47, 2021.

[33] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. Flightllm: Efficient large language model inference with a complete mapping flow on fpgas. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, page 223–234, New York, NY, USA, 2024. Association for Computing Machinery.

[34] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model, 2024.

[35] ZhenDong Zheng, Qianyu Cheng, Teng Wang, Lei Gong, Xianglan Chen, Cheng Tang, Chao Wang, and Xuehai Zhou. Lora: A latency-oriented recurrent architecture for gpt model on multi-fpga platform with communication optimization. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 332–338, 2024.

[36] Rui-Jie Zhu, Yu Zhang, Ethan Sifferman, Tyler Sheaves, Yiqiao Wang, Dustin Richmond, Peng Zhou, and Jason K Eshraghian. Scalable matmul-free language modeling. *arXiv preprint arXiv:2406.02528*, 2024.