FPGA accelerator for LLM

Report

Work of:

Jean-Luc Gauvrit

gauvritj37@gmail.com

Intern Student

With the supervision of:

Tae-Wan Kim

August 2025

Contents

List of Figures

Abstract

This report details the development of an FPGA-based accelerator for Large Language Models (LLMs), with a focus on a variant of the LLaMa architecture. Motivated by the need for energy-efficient inference in low-power environments, the project begins with a CPU-based implementation in C++, inspired by the llama.cpp framework and aligned with the Swan project. This software pipeline handles model loading, tokenization, forward passes, and text generation. Transitioning to hardware, High-Level Synthesis (HLS) is employed to design specialized kernels for key operations such as matrix multiplication, softmax, and rotary position encoding.

These kernels are deployed on the ZCU106 FPGA board using a heterogeneous Processing System (PS) and Programmable Logic (PL) architecture, facilitated by PetaLinux. Preliminary tests demonstrate functional inference, paving the way for optimizations including customized PetaLinux images and refined kernel synthesis. The ultimate goal is to inform the design of an Application-Specific Integrated Circuit (ASIC) for LLM acceleration in resource-constrained settings.

Résumé

Ce rapport détaille le développement d'un accélérateur basé sur FPGA pour les grands modèles de langage (LLM), en se concentrant sur une variante de l'architecture LLaMa. Motivé par le besoin d'une inférence économe en énergie dans des environnements à faible puissance, le projet commence par une implémentation basée sur CPU en C++, inspirée du framework llama.cpp et alignée sur le projet Swan. Ce pipeline logiciel gère le chargement du modèle, la tokenisation, les passes avant et la génération de texte. Passant au matériel, la synthèse de haut niveau (HLS) est utilisée pour concevoir des noyaux spécialisés pour des opérations clés telles que la multiplication matricielle, softmax et l'encodage de position rotatif.

Ces noyaux sont déployés sur la carte FPGA ZCU106 en utilisant une architecture hétérogène Système de Traitement (PS) et Logique Programmable (PL), facilitée par PetaLinux. Des tests préliminaires démontrent une inférence fonctionnelle, ouvrant la voie à des optimisations incluant des images PetaLinux personnalisées et une synthèse de noyaux affinée. L'objectif ultime est d'informer la conception d'un circuit intégré spécifique à l'application (ASIC) pour l'accélération des LLM dans des contextes à ressources limitées.

1 Introduction

1.1 Presentation of the Report

This report presents the progress of my work on designing and optimizing a hardware accelerator for Large Language Models (LLMs) using Field-Programmable Gate Arrays (FPGAs). Its objective is to document technical advancements, highlight open design questions, and ensure alignment with the research direction defined in collaboration with Professor Tae-Wan Kim. Each section outlines key investigations, experimental results, and technical challenges encountered during the week.

1.2 Presentation of the Subject

The goal of this project is to do inference with a LLM, specifically a variant of the LLaMa family, on a FPGA platform. LLaMa-Variant has recently demonstrated excellent inference performance across various benchmarks, offering competitive accuracy while being more lightweight and efficient than many other open-source LLMs of similar size. These characteristics make it a strong candidate for hardware-level optimization.

LLMs typically require considerable computational power and memory bandwidth, which limits their deployment in edge or low-power environments. While GPUs provide strong performance, their energy consumption is often prohibitive for embedded or mobile use cases.

This project explores the use of FPGAs to design a custom, energy-efficient accelerator optimized for LLM inference. The flexibility of FPGAs allows for fine-tuned architectural choices targeting both compute and memory efficiency. Our focus is on reducing power consumption while maintaining acceptable latency (e.g., 100–200 ms per token), with attention given to constraints such as voltage levels, memory interfaces, and model size.

Our final goal is to develop an Application-Specific Integrated Circuit (ASIC) that is energy-efficient and capable of accelerating LLMs in low-power environments. The development of a Proof of Concept (PoC) using FPGAs is a crucial step towards achieving this goal. The PoC will demonstrate the feasibility and potential benefits of using FPGAs for LLM acceleration, paving the way for future ASIC development.

1.3 FPGA

Field-Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits that can be programmed to implement custom digital logic. Their main advantage lies in flexibility : unlike fixed-function hardware, FPGAs allow designers to create architectures tailored to specific applications, such as neural network inference. This adaptability makes them well suited for achieving an efficient balance between performance, power consumption, and resource usage.

2 LLaMa

2.1 Overview of LLaMa

LLaMa, developed by Meta AI, is a family of transformer-based language models known as Large Language Model Meta AI. These models are open-weight, offering a lightweight alternative to proprietary models like GPT-3, and have gained significant traction in the open-source AI community. As described in the original LLaMa paper [4], LLaMa employs a decoder-only transformer 1 architecture optimized for efficiency and portability.

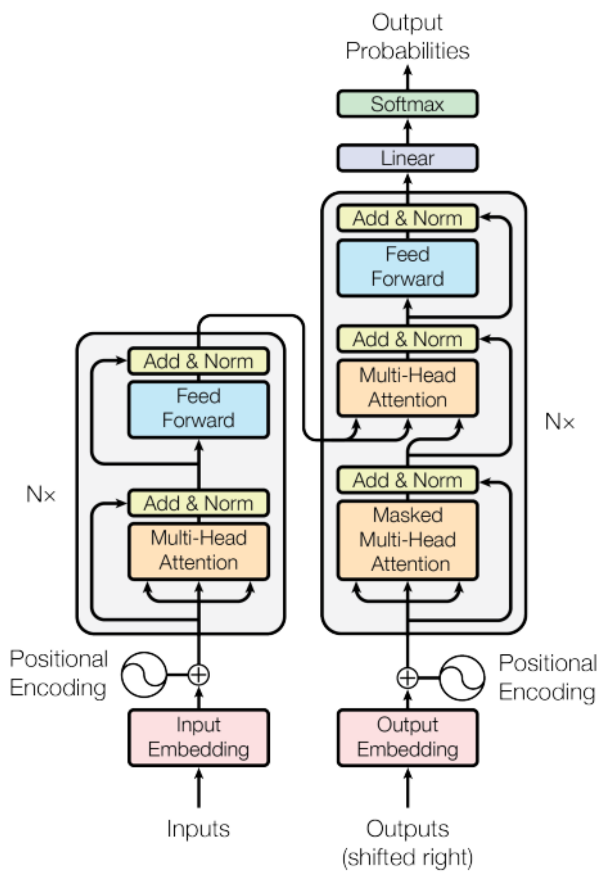

Figure 1: Transformer of LLaMA

It supports autoregressive text generation, similar to GPT-family models, but is designed for fine-tuning on modest hardware. Successors such as LLaMa 2 and LLaMa 3 have further improved model alignment, scalability, and training data quality, enhancing their applicability for various tasks.

2.2 Inference with LLaMa

The LLaMa architecture was selected for this project due to its widespread adoption as a foundation for advanced language models, including DeepSeek, Mistral, and TinyLLaMA2.

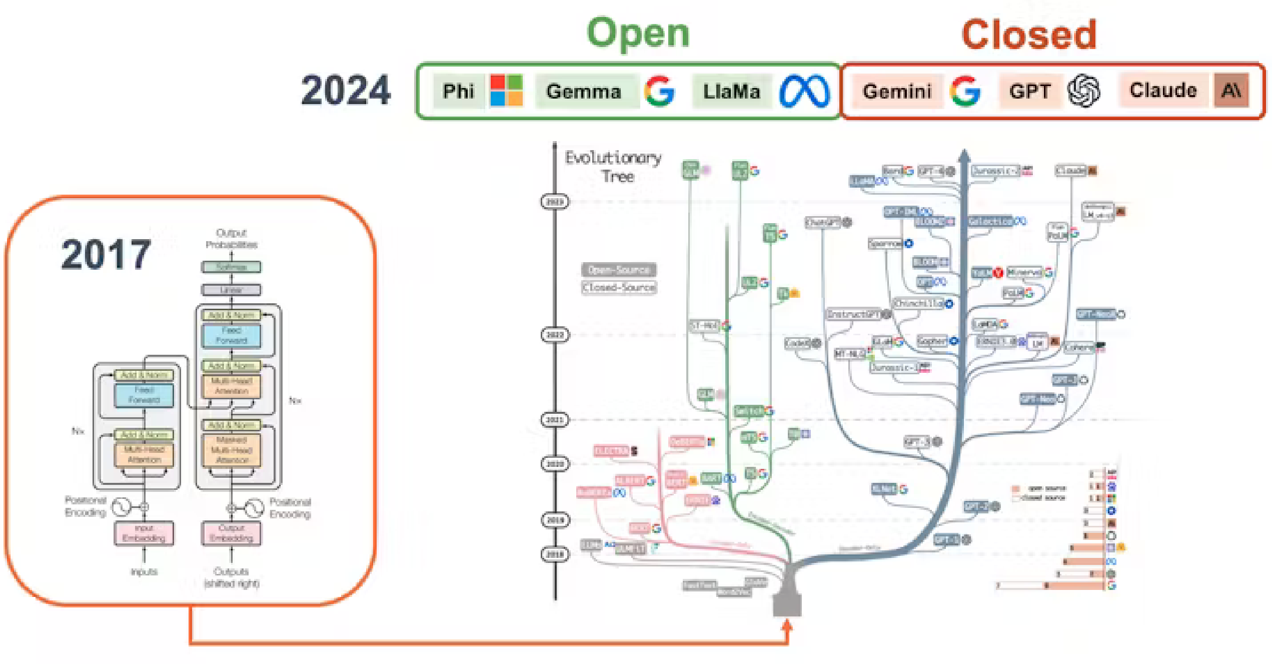

Figure 2: LLaMa variant

Developing an efficient inference kernel for LLaMa ensures compatibility with these derived models, enabling a reusable computational pipeline without requiring significant redesign. The inference process follows a standardized computational graph, encompassing embedding lookup, multi-head attention with RoPE, feed-forward layers, normalization, and token sampling. This structure facilitates hardware acceleration efforts, particularly on FPGAs, by providing a well-defined and reusable compute framework.

The standardization of LLaMa's architecture offers several advantages for hardware acceleration:

- **Reusability**: The same inference backend can support multiple LLaMa-derived models, reducing development overhead.
- **Hardware Optimization**: The clearly defined computational structure guides FPGA-based acceleration, enabling targeted optimizations for compute and memory efficiency.
- **Ecosystem Compatibility**: Support for quantized models in formats like GGUF ensures integration with existing tools such as `llama.cpp` [2].

By focusing on LLaMa for inference, this project establishes a strategic foundation for building lightweight, hardware-friendly AI pipelines, suitable for deployment in resource-constrained environments.

3 Inference on CPU

3.1 C++ Inference Implementation

Drawing inspiration from the `llama.cpp` project [2], the inference engine for this project, aligned with the Swan project [3], is implemented in modern C++. This choice ensures cross-platform portability, high computational efficiency, and compatibility with High-Level Synthesis (HLS) tools for FPGA deployment. The inference logic is designed to be flexible, allowing compilation for CPU execution or synthesis into FPGA hardware logic based on the selected backend.

The C++ inference pipeline 3 is structured as follows:

1. **Model and Tokenizer Loading**: Quantized model weights (e.g., `stories15M.bin`[1]) and the tokenizer vocabulary (`tokenizer.bin`) are loaded into memory from open repositories. These files contain the model parameters and tokenization scheme used during training.

2. **Tokenization**: The input prompt is converted into token IDs using a lightweight tokenizer implemented in `vocab.cpp/.hpp`, optimized for embedded inference.

3. **Forward Pass**: Each token is processed through a forward pass, comprising:

   - Embedding lookup to map tokens to vectors.
   - Matrix multiplications for attention and feed-forward (MLP) blocks.
   - Non-linear activation functions (e.g., ReLU, GELU).
   - Softmax normalization for output probabilities.

   These operations are executed on the CPU (`tensor.cpp`) or accelerated via FPGA modules (`tensor_fpga.cpp`).

4. **Sampling and Generation**: Output logits undergo temperature scaling and sampling to select the next token, which is appended to the sequence. This process iterates until the maximum sequence length or an end-of-sequence token is reached.

5. **Decoding**: The token sequence is converted back to human-readable text using the tokenizer vocabulary.

This modular design allows seamless substitution of operations (e.g., replacing CPU-based matrix multiplication with FPGA-optimized kernels), facilitating performance and power efficiency trade-offs for edge AI inference. The pipeline supports quantized LLMs, ensuring reasonable accuracy and responsiveness in resource-constrained environments.

[1] I use this small model, we will scale later

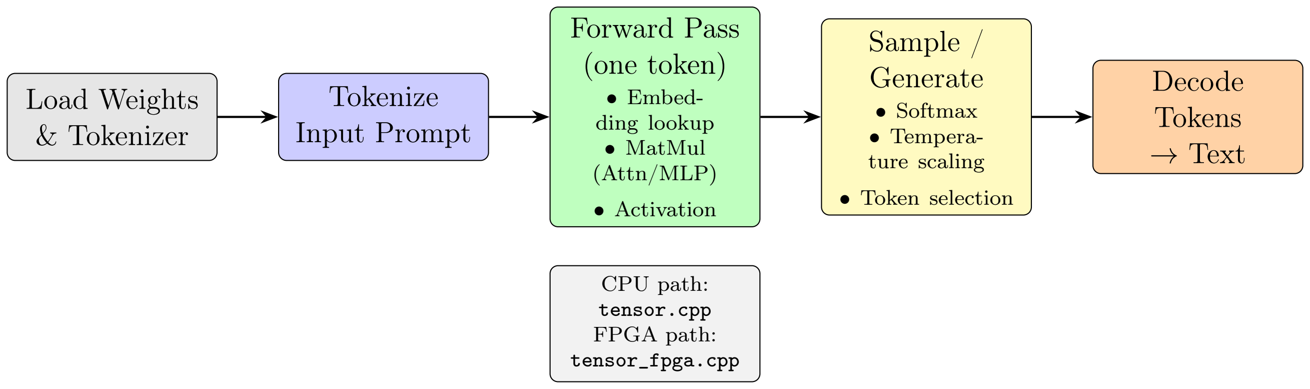

Figure 3: End-to-end inference pipeline for LLM processing, from model loading to text generation.

3.2 Main and Build System

The file `main.cpp` is the entry point of the inference framework. It coordinates the execution pipeline by:

- Parsing command-line arguments (e.g., weight path, prompt, sequence length).
- Loading model weights and the tokenizer from binary files.
- Tokenizing the input prompt and feeding it into the inference engine.
- Executing the forward pass loop to generate output tokens.
- Decoding the results back into human-readable text.

This design enables a clean separation between infrastructure code and inference logic, facilitating backend switching (CPU or FPGA).

3.3 Build, Run & Test

3.3.1 CMake-based Build System

The project uses a minimalist CMake build configuration. The root `CMakeLists.txt` 7.1 supports both CPU-only and FPGA-accelerated builds.

To build the project for CPU-only execution (no FPGA), simply run :

```
mkdir build && cd build
cmake ..
make
```

3.3.2 Command-line Inference

Once compiled, inference can be launched via terminal with a prompt and output options :

```
./inference_fpga --prompt "Once upon a time" --max_seq 64 --temp 0.7
```

Available flags include :

Flag	Description	Default
`--weight_path`	Path to `.bin` weight file	`model/stories15M.bin`
`--vocab_path`	Path to tokenizer binary	`model/tokenizer.bin`
`--prompt`, `-p`	Initial prompt (UTF-8 string)	empty (generate from BOS)
`--max_seq`	Maximum number of generated tokens (prompt is added on top)	256
`--temp`	Sampling temperature (≤ 0 means greedy decoding)	0.5
`--color`	Enable ANSI-colored token output	off
`--log`	Dump intermediate tensors to `./log/`	off
`--help`, `-h`	Show help and exit	—

Figure 4: Table of command-line options for CPU inference

This CPU fallback mode is ideal for debugging, testing, and profiling without requiring access to FPGA hardware. It also facilitates CI pipelines and quick iterations on the model logic.

3.3.3 Test

To validate the functionality and performance of the C++ inference pipeline, a test was executed on a standard development machine using the CPU-only build. The inference binary was launched with a temperature of 1 and a generation limit of 100 tokens :

```
./build/inference_fpga --temp 1 --max_seq 100
```

Upon execution, the model reports its hyperparameters, which confirm that a 15M-parameter TinyLLaMA model is being used. Key parameters include a hidden dimension of 288, 6 transformer layers, and a vocabulary size of 32,000 tokens. The system generated a coherent text sequence starting with "Once upon a time...", demonstrating proper functioning of the token generation loop.

```
Hyper Parameters
  dim       : 288
  ffn_dim   : 768
  n_layers  : 6
  n_heads   : 6
  n_kv_heads: 6
  vocab_size: 32000
  seq_len   : 256
 Once upon a time, there was a little girl named Lily...
```

The full inference took approximately 4.25 seconds, achieving an average generation speed of **23.5 tokens per second**. This confirms that the CPU-only pipeline is functional and sufficiently fast for development and debugging purposes.

```
Time : 4.25001[s]
Speed: 23.5293[tok/s]
```

4 Hardware design

4.1 Kernels

The hardware design process for the FPGA accelerator leverages a structured workflow to transition from high-level software to optimized hardware implementation. This approach utilizes a C-Testbed framework, integrating both simulation and synthesis tools to ensure compatibility and performance on the FPGA platform.

The deployed system on the `ZCU106` board leverages a heterogeneous architecture, combining:

- **Processing System (PS):** ARM Cortex-A cores executing the control flow, preprocessing, and orchestration of FPGA kernels.
- **Programmable Logic (PL):** Dedicated hardware blocks implementing computationally expensive operations for LLM inference.

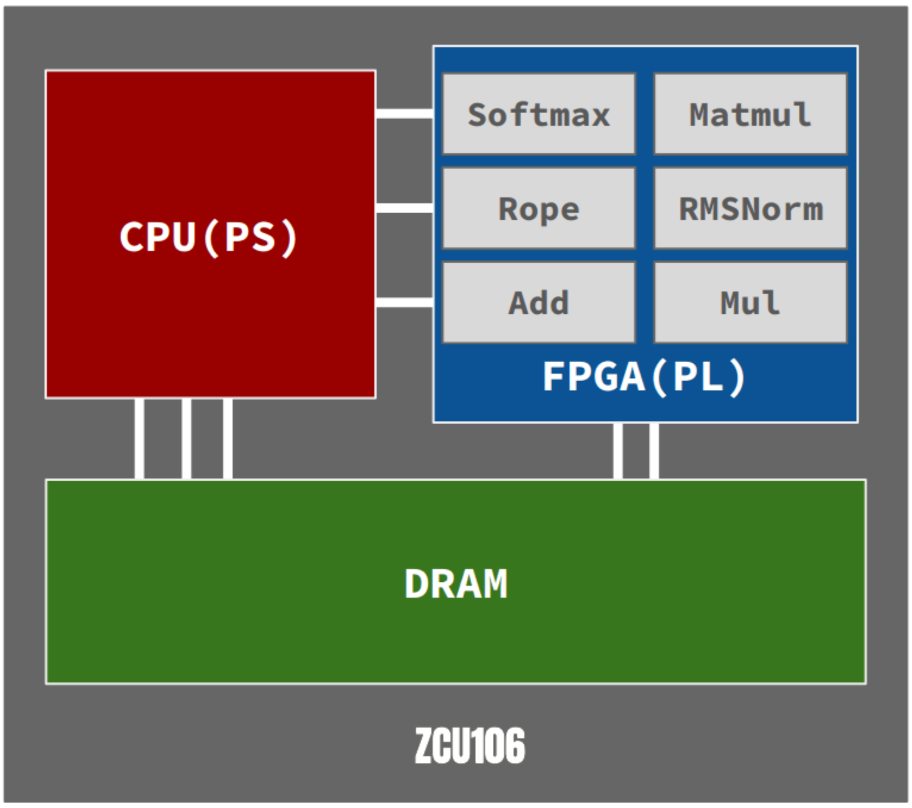

Figure 5: PS/PL design

The PL hosts the custom HLS-generated kernels:

- Softmax
- Matrix Multiplication (Matmul)
- RMS Normalization
- Rotary Position Encoding (Rope)
- Addition

- Multiplication

The PS and PL share access to the DDR4 DRAM, enabling high-bandwidth data exchange. This setup reduces latency by offloading heavy tensor operations to the PL while maintaining flexibility in the PS.

4.2 HLS Kernel Compilation and FPGA Bitstream Generation

Each kernel shown in Figure 5 was implemented as an independent C++ source file and synthesized using the Vitis HLS toolchain. The compilation process generated intermediate object files in `.xo` format, which were subsequently linked together to form a single `XCLBIN` file. This unified bitstream encapsulates all the hardware kernels required for the accelerator.

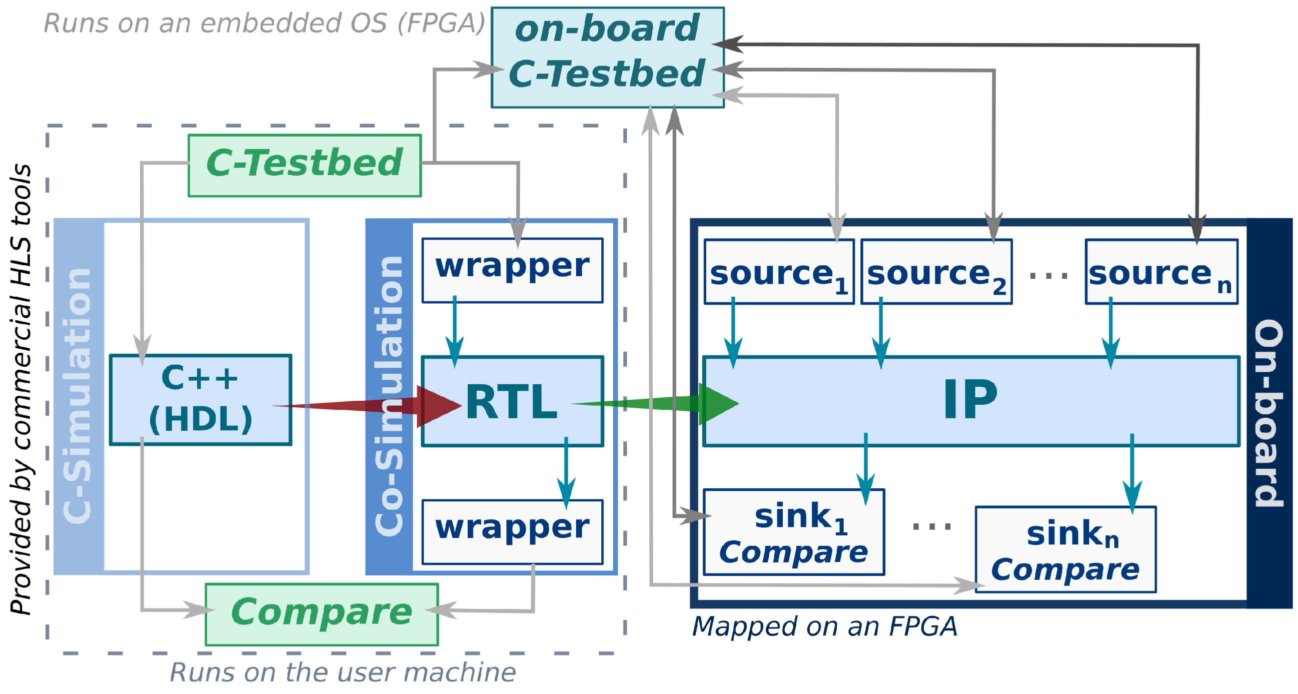

Figure 6: FPGA-Based Solution for On-Board Verification of Hardware Modules Using HLS [1]

The resulting IP (Intellectual Property) block, mapped onto the FPGA, incorporates multiple source files (source$_1$, source$_2$, ..., source$_6$) that define the accelerator's core logic. These source files are processed alongside sink components (sink$_1$, sink$_2$, ..., sink$_6$) and a compare module to verify the output against expected results. The on-board C-Testbed, running on an embedded operating system (OS)[2] within the FPGA, facilitates real-time execution and data exchange with the IP block. This setup allows for dynamic testing and optimization, ensuring the accelerator meets the project's performance and power efficiency goals for LLM inference.

A visual representation of this workflow would illustrate the flow from C-Testbed simulation on the user machine to IP deployment on the FPGA, highlighting the interaction between C++ simulation, RTL synthesis, and on-board execution. [Insert diagram here showing the C-Testbed, co-simulation wrappers, RTL, IP block, source and sink components, and compare module, with arrows indicating data flow between the user machine and FPGA.

Once generated, the bitstream was extracted from the XCLBIN and programmed onto the ZCU106 FPGA, using either Vivado or the `xsct` command-line utility, making the hardware design ready for deployment and execution of LLM inference workloads.

[2]Petalinux

4.3 Board Programming and Deployment

4.3.1 PetaLinux Setup

To simplify the integration and execution of inference workloads on the ZCU106 FPGA board, I utilized the **PetaLinux** development tools provided by AMD Xilinx. PetaLinux enables the creation of a fully customized embedded Linux distribution tailored for the Zynq UltraScale+ MPSoC architecture, offering a seamless environment for deploying and running software alongside FPGA hardware acceleration.

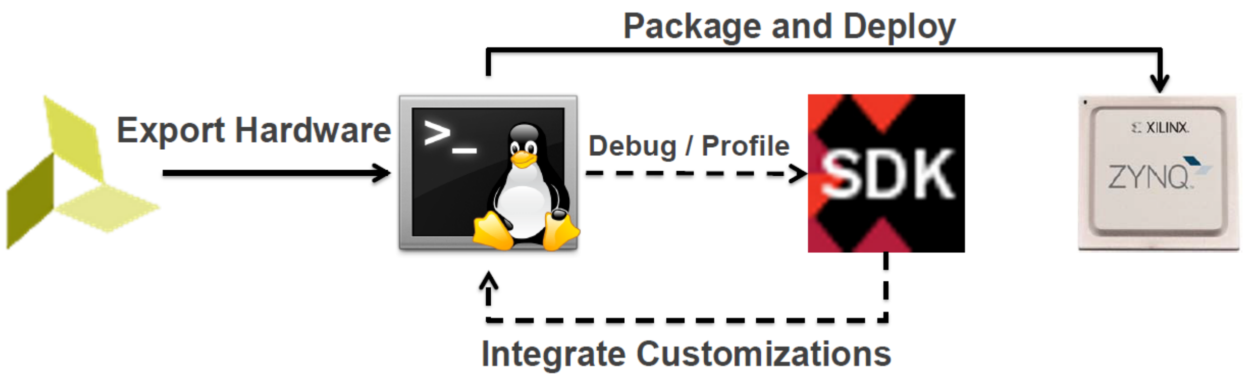

Figure 7: PetaLinux workflow for deployment and debug on Zynq MPSoC platforms

4.3.2 Deploy

To deploy the system on the ZCU106 board, the generated `BOOT.BIN`, Linux kernel image (`Image`), and device tree blob (`system.dtb`) were copied onto a bootable microSD card. The board was then configured to boot from the SD card by adjusting its hardware DIP switch settings. After powering on, the system successfully initialized into the embedded Linux environment, with UART communication providing real-time console output for monitoring the boot process.

The inference host application previously cross-compiled with the ARM toolchain was then executed within this Linux environment. This application leverages the XRT runtime to communicate with the FPGA and load the `.xclbin` bitstream containing the hardware-accelerated kernels.

This configuration enabled end-to-end inference directly on the FPGA. Performance evaluation was carried out by measuring runtime metrics and token generation rates across various prompts. The complete workflow from boot to inference demonstrated a smooth integration of hardware and software, validating the effectiveness of the PetaLinux-based deployment approach.

5 Docker-Based Development and Deployment Environment

To enhance the development, testing, and deployment of the FPGA-accelerated inference system, a containerized environment was established using Docker and Docker Compose. This setup ensures a consistent, isolated, and efficient workflow, covering all stages from source code preparation to deployment on the FPGA board.

The architecture is structured around multiple specialized containers, each dedicated to a specific function:

- **User Interface (UI) Container:** A lightweight Flask-based web application hosted locally, enabling developers to manage the build and deployment process via a browser. It supports tasks such as source code preparation, model selection, and build initiation without relying on command-line interfaces.
- **Build-CPU Container:** Equipped for native compilation of the inference software for CPU execution. It supports building both the reference implementation and custom user-provided code and models.
- **Build-FPGA Container:** Configured with the ARM cross-compilation toolchain and Xilinx Vitis environment, this container compiles binaries for the ZCU106 Processing System (PS) and manages FPGA kernel synthesis and linking.
- **Shared Source Volume:** A persistent `source/` directory shared across containers to store C++ source files and models, facilitating seamless data exchange between the UI, build, and deployment processes.

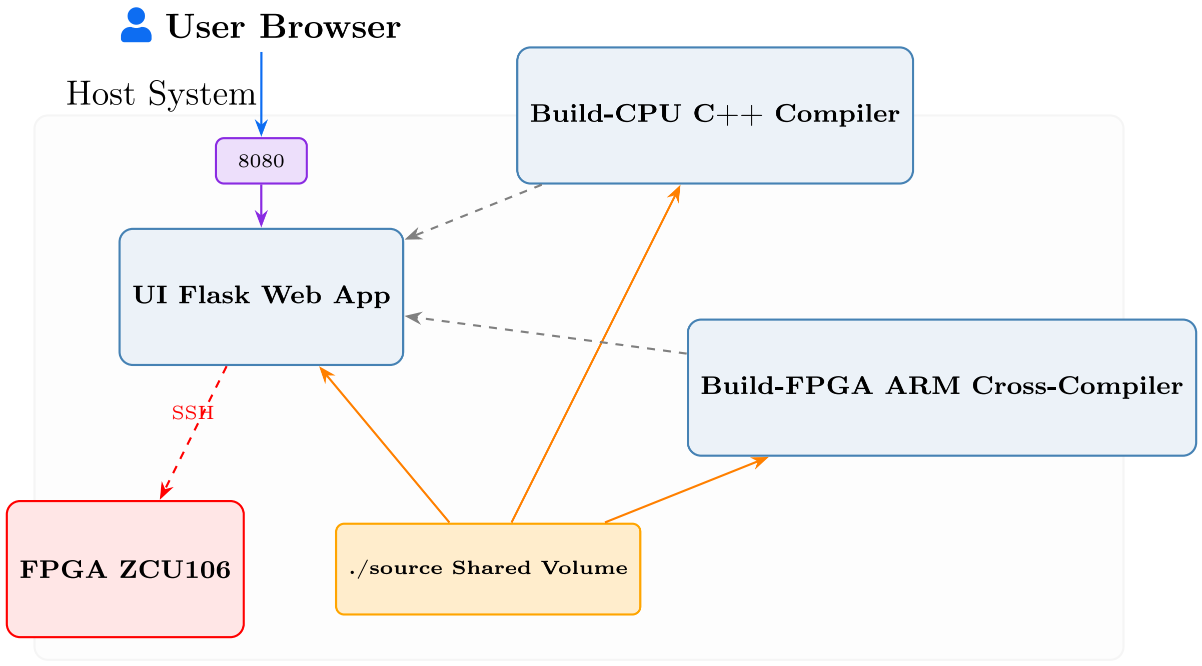

Figure 8: Docker-based development and deployment architecture for FPGA-accelerated inference.

To optimize the development workflow, `docker compose watch` is utilized to monitor the source directory and automatically propagate changes to the relevant containers, enabling near real-time updates without manual rebuilds. This significantly accelerates the development cycle.

For deployment, the containerized environment simplifies the process of transferring compiled binaries and FPGA bitstreams to the ZCU106 board, either via network protocols (SCP/SSH) or by generating an SD card image within a container. This approach ensures consistency across different development machines, mitigating common issues with FPGA toolchain configuration.

In summary, this Docker-based toolchain reduces the complexity of FPGA development by providing a reproducible and collaborative environment. It allows developers to focus on optimizing algorithms and kernel designs, streamlining the path from development to deployment.

6 Conclusion

In this report, we have presented the development of an FPGA-based accelerator for Large Language Models (LLMs), focusing on the LLaMa architecture. Starting from a CPU-based inference implementation inspired by `llama.cpp` and aligned with the `Swan` project, I transitioned to a hardware design leveraging High-Level Synthesis (HLS) on the `ZCU106` board. Key components include the integration of specialized kernels for operations such as matrix multiplication, softmax, and rotary position encoding, deployed through a heterogeneous PS/PL architecture. The end-to-end pipeline was validated through CPU testing and FPGA deployment using PetaLinux, demonstrating functional inference with promising performance metrics.

Despite these advancements, there remain opportunities for further optimization of the inference process in two primary ways. First, the current deployment relies on the standard PetaLinux image provided for the ZCU106 board, which is not tailored to our specific accelerator needs. To enhance efficiency, a customized image should be recreated using Vivado, allowing for better resource allocation, reduced overhead, and improved integration with the hardware kernels.

Second, the kernel generation process can be refined by gaining full control over the synthesis workflow. Presently, kernels are created directly from C++ sources using the `v++` compiler within the Vitis toolchain, which limits fine-grained optimizations. By mastering the entire kernel generation pipeline—including manual adjustments to RTL descriptions, advanced HLS directives, and iterative co-simulation—we can achieve higher performance, lower latency, and better power efficiency.

These improvements will pave the way toward our ultimate goal of developing an energy-efficient ASIC for LLM acceleration in low-power environments. Future work will focus on scaling to larger models, quantitative benchmarking, and exploring quantization techniques to further reduce resource demands.

7 Annex

7.1 CMakeLists.txt to compile for CPU

```
cmake_minimum_required(VERSION 3.3)

SET (CMAKE_CXX_FLAGS "-Wall -Wextra -std=c++2a -mcmodel=large")
add_definitions(-DUSE_CPU_ONLY)

file(GLOB_RECURSE SOURCES
    src/*.cpp
    src/*.hpp
)

message("# SOURCES: ${SOURCES}")

include_directories(src)

project(inference-CPU CXX)
add_executable(inference-CPU ${SOURCES})
```

7.2 CMake with the aarch64 compiler

```
# toolchain-aarch64.cmake
set(CMAKE_SYSTEM_NAME        Linux)
set(CMAKE_SYSTEM_PROCESSOR   aarch64)

set(CMAKE_C_COMPILER   /tools/Xilinx/2025.1/Vitis/gnu/aarch64/lin/
   aarch64-linux/bin/aarch64-linux-gnu-gcc)
set(CMAKE_CXX_COMPILER /tools/Xilinx/2025.1/Vitis/gnu/aarch64/lin/
   aarch64-linux/bin/aarch64-linux-gnu-g++)

set(CMAKE_C_FLAGS   "-O3 -fPIE")
set(CMAKE_CXX_FLAGS "-O3 -fPIE -std=c++17")
```

7.3 FPGA with GPU design

7.3.1 Third party with a GPU

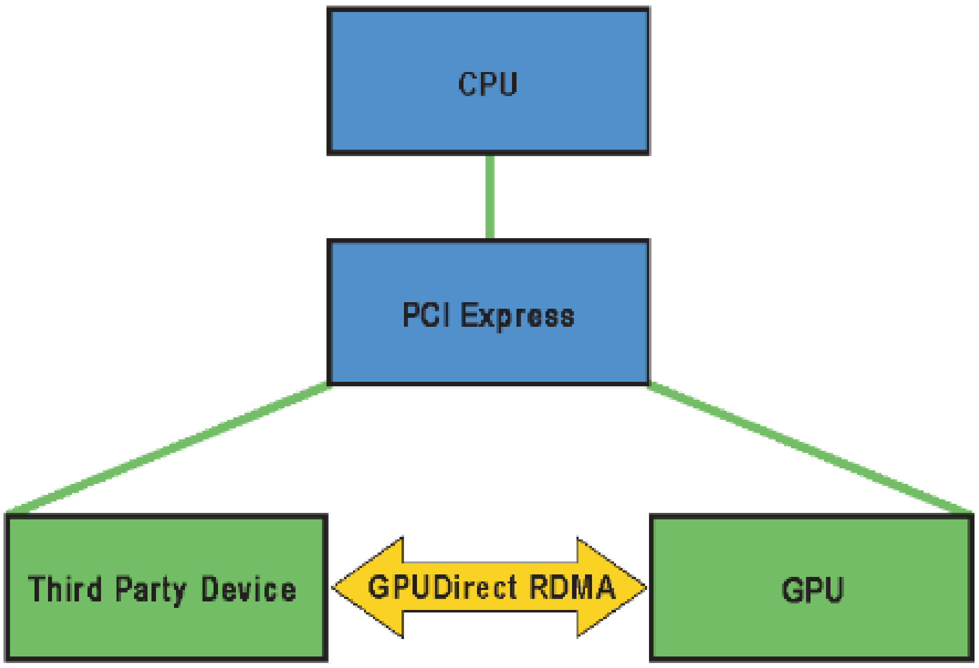

Figure 9: GPU Direct memory shared design

7.3.2 CUDA

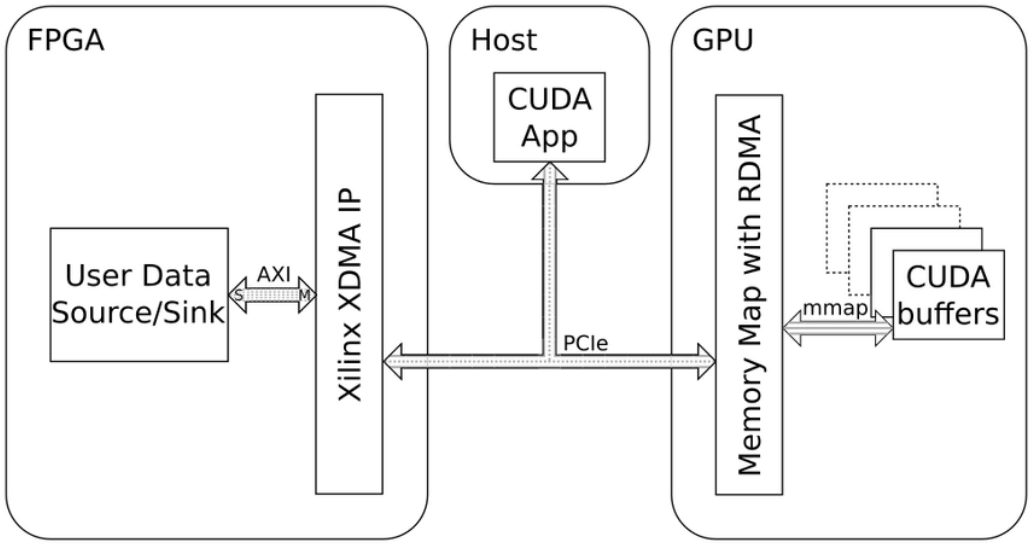

Figure 10: CUDA third party

References

[1] Julián Caba, Fernando Rincón, Jesús Barba, José Antonio de la Torre, and Juan Carlos López. Fpga-based solution for on-board verification of hardware modules using hls. *Electronics*, 9(12), 2020.

[2] Georgi Gerganov and Contributors. llama.cpp: Llm inference in c/c++. `https://github.com/ggml-org/llama.cpp`, 2023. Accessed: 2025-07-22.

[3] Turing Inc. Swan: A lightweight language model execution environment using fpga. `https://github.com/turingmotors/swan`, 2024. Accessed: 2025-07-22.

[4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.