JAIST Repository
https://dspace.jaist.ac.jp/

Title	LLM エッジ推論のFPGA実装及び最適化に関する研究
Author(s)	何, 嘉翔
Citation	
Issue Date	2025-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/20049
Rights	
Description	Supervisor: 田中 清史, 先端科学技術研究科, 修士 (情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

A Study on FPGA Implementation and Optimization of LLM Edge Inference

2330002 HE Jiaxiang

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

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

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

At the core of most modern LLMs lies the Transformer architecture, which revolutionized sequence modeling through the use of self-attention mechanisms and feed-forward networks. These innovations enable the parallel computation of token dependencies, significantly enhancing scalability and performance. Key operations within a Transformer block include matrix multiplications for linear projections (e.g., Query, Key, and Value), attention score calculation (via additional matrix multiplications and softmax), and element-wise operations for activation functions and normalization layers. These operations represent the fundamental computational kernels that must be efficiently mapped onto hardware for inference acceleration.

With the computational workload defined, the next step is selecting an appropriate hardware platform. For edge computing scenarios, traditional FPGAs often require external processors to handle control flow and system-level functions. System-on-Chip FPGAs (SoC FPGAs) address this limitation by integrating a general-purpose processor (typically an ARM Cortex-A series core) with reconfigurable FPGA fabric on a single chip. This architecture supports a hardware-software co-design paradigm: the embedded CPU can manage tasks such as model loading, I/O, and inference orchestration, while the FPGA fabric accelerates compute-intensive LLM kernels. This integrated approach reduces inter-chip communication overhead, improves latency, and typically offers lower power consumption compared to multi-chip solutions.

OpenCL serves as the unifying programming model for the heterogeneous SoC FPGA platform. It enables the separation of computation into "host" code, executed by the embedded CPU, and "kernel" code, executed on the FPGA fabric. This abstraction simplifies development by allowing high-level control flow on the CPU while delegating compute-intensive operations to custom FPGA accelerators. Moreover, OpenCL ' s hardware-agnostic interface promotes portability and modularity in design.

While OpenCL defines the programming model, High-Level Synthesis (HLS) tools—such as AMD ' s Vitis HLS—enable the practical implementation of OpenCL kernels on FPGAs. HLS allows developers to describe hardware logic in high-level languages like C/C++, significantly reducing the complexity and time associated with traditional HDL-based design. HLS compilers generate register-transfer level (RTL) code automatically, inferring key hardware constructs such as pipelining, parallel data paths, and memory access patterns from the high-level code. Additionally, developers can apply directive-based optimizations (e.g., loop unrolling, pipelining, and dataflow partitioning) to guide the synthesis process and exploit parallelism effectively.

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

The primary purpose is to create a fully standalone edge AI device for LLM inference. This capability is crucial for various intended use cases where real-time, on-device processing is paramount. To meet these demands, the

core performance specifications for the device are: Response Time (less than 1 second), Context Length (200 characters or more) and Inference Speed (2 words per second).

The tinyllamas model is selected for its lightweight architecture, which aligns well with our requirements in terms of model size and input sequence length, as well as its high implementability on resource-constrained edge devices. The AMD Xilinx Kria™ KV260 development board is chosen as the hardware platform. The KV260's SoC FPGA integrates an Arm CPU-based Processing System (PS) and Programmable Logic (PL). To manage diverse tasks and ensure flexible development, a Linux OS is employed. The model implementation leverages AMD's PetaLinux in Symmetric Multiprocessing (SMP) mode. Xilinx Runtime (XRT) acts as an open-source software stack, managing the FPGA devices and mediating communication between the host application and the custom accelerators on the FPGA.

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

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

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

The FPGA hardware configuration is built using C-language-based source code and compiled with the Vitis IDE's High-Level Synthesis (HLS) tool. All six accelerator kernels are integrated into a single XCLBIN file. This compiled file is then copied to a specified location on the evaluation board's

MicroSD card. Upon execution of the host application, the XCLBIN file is read, and the PL of the FPGA is configured.

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

The matmul_kernel is decomposed into four internal functions: stream_ input_vector and stream_input_matrix for data input, compute_matrix_vector_ multiply for computation, and write_result_to_memory for data output. In the initial implementation, the DATAFLOW pragma is applied to these functions to enable function-level parallelization. Additionally, loop unrolling (UNROLL) is attempted on the inner loop of the compute_matrix_vector_multiply function's nested loop (matrix_multiply_execute). However, a critical limitation is identified: due to the matmul_kernel's design for versatility across different vector and matrix sizes, the HLS tool could not determine the exact number of loop iterations at compile time. This often resulted in the UNROLL pragma being ignored, hindering the desired performance gains.

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

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

This work establishes a foundation for deploying LLMs in resource-constrained environments and identifies key optimization strategies for future research. The results indicate strong potential for FPGA-based LLM infer-

ence in applications requiring local processing, power efficiency, and cost-effectiveness, while highlighting opportunities for further improvements through quantization, mixed-precision arithmetic, and advanced memory management techniques.