Edge QLoRA: Hardware Accelerated QLoRA for On-Device LLM Fine Tuning

DVCON India Design Contest 2025

Design Report for Stage 2 Submission

Submission Number: 6481

June 1, 2025

Group: Silicon Edge

Goonetilleke P
Jayakody J A K
Warushavithana N D

Department of Electronic and Telecommunication Engineering
University of Moratuwa, Sri Lanka

Contents

List of Tables

List of Figures

Abstract

Deploying and fine-tuning large language models (LLMs) on edge devices is challenging due to their high computational demands, memory requirements, and power consumption. While Low-Rank Adaptation (LoRA) reduces fine-tuning complexity, its memory footprint remains prohibitive for resource-constrained devices. Quantized LoRA (QLoRA) addresses this by combining 4-bit quantization with LoRA, significantly reducing memory usage while preserving model performance. However, efficient hardware acceleration is still needed to meet the real-time processing and energy constraints of edge applications. This work presents Edge QLoRA, an FPGA-accelerated implementation of QLoRA designed for on-device LLM fine-tuning. We propose a custom hardware architecture leveraging FPGA reconfigurability to optimize quantization, dequantization, and matrix operations—key components of QLoRA. Our design integrates systolic arrays for parallelized matrix multiplication, on-chip BRAM caching for efficient memory management, and power-optimized pipelines for inference and fine-tuning. Key contributions include: A memory-efficient FPGA implementation of QLoRA, reducing TinyLlama's (1.1B parameters) footprint to ¡300MB while enabling fine-tuning on edge devices, hardware-accelerated 4-bit quantization/dequantization with custom IP cores, improving computational throughput, and benchmarks demonstrating superior latency, power efficiency, and memory savings compared to GPU-based QLoRA, validated through simulations targeting a Vega processor-based FPGA. Our preliminary estimates (python based software implementations) indicate a 99% speedup in processing time (for quantiztion and dequantization) and we expect 75% reduction in model size compared to FP16 baselines, making LLM fine-tuning feasible on edge hardware. This approach bridges the gap between cloud-based LLMs and edge deployment, enabling privacy-preserving, low-latency applications in healthcare, IoT, and robotics. Future work will focus on expanding support for larger models and dynamic quantization strategies.

1 Introduction

Deploying large language models (LLMs) on edge or embedded devices is extremely challenging. These models demand substantial memory to store parameters and extensive computational power. This computational power is typically provided by GPUs, which are power hungry and expensive, removing them as a viable option for edge/embedded applications.

Furthermore, fine-tuning LLMs on edge devices is nearly impossible due to their resource constraints. While cloud-processing could address these issues, it adds latency and poses a privacy concern.

Recent advancements such as Low-Rank Adaptation (LoRA) provide a way to accelerate fine-tuning while reducing computational requirements. However, LoRA alone does not address the memory demands of LLMs. This limitation is tackled by Quantized LoRA (QLoRA), which applies quantization techniques to compress model parameters, drastically reducing memory usage and improving computational efficiency. By quantizing model weights to lower bit-width representations, QLoRA reduces memory footprints and accelerates matrix multiplication, the core operation in LLMs. This makes LLM deployment on edge devices more feasible. We aim to implement QLoRA on FPGA-based systems, leveraging their high computational throughput, low power consumption, and reconfigurability. Specifically, we will:

- Design custom IP cores for quantization and matrix multiplication, optimized for QLoRA.
- Minimize resource usage to fit the constraints of embedded systems.
- Evaluate performance gains in speed, memory efficiency, and power consumption compared to traditional GPU-based solutions.

Implementing QLoRA on FPGAs would enable efficient, low-latency deployment of LLMs on resource-constrained devices. This approach offers a promising alternative to cloud processing, enhancing privacy, robustness, and efficiency for various applications.

2 Background Research

2.1 QLoRA and Efficient Fine Tuning

Deploying large language models (LLMs) on edge devices is challenging due to their high memory requirements, computational complexity, and power consumption. Edge devices, such as embedded systems, typically lack the processing power and memory capacity needed for LLM inference and fine-tuning. GPUs, which are commonly used for LLM deployment, are power-hungry and expensive, making them impractical for most edge applications. Cloud processing offers a partial solution, but it introduces latency, dependency on external networks, and potential privacy concerns when transmitting sensitive data. Efficiently deploying and fine-tuning LLMs on edge devices could enhance privacy, reduce latency, and improve power efficiency, making it a valuable objective for various real-time, resource-constrained applications.

LoRA: Low Rank Adaptation for large language models (LLM) is a method to fine tune a large language model by only training two low rank matrices, task specific LoRA modules, while the pretrained model weights are freezed. This method drastically reduces the inference latency and input sequence length while retaining the high model quality. [1]. LoRA can be used in fine tuning LLM on edge devices but the main constraint is the inefficiency that is caused by the high memory usage of these large models.

QLoRA is a novel method for efficiently finetuning large language models by 4-bit quantization and using Low Rank Adapters (LoRA). This approach optimizes memory usage without affecting performance. QLoRA enables multi billion parameter models to be finetuned on a single GPU, while preserving the original 16-bit precision of the model weights. QLoRA incorporates three main features to achieve this performance: 4-bit normal float (NF4) data type which is optimal for the normally distributed model weights; double quantization to further reduce memory usage by quantizing the quantization constants, and paged optimizers to handle memory spikes [2].

2.2 Challenges in Implementing Large Language Models in Edge Devices

Deploying large language models (LLMs) on edge devices presents several critical challenges due to their computational demands and resource constraints. Edge devices, such as embedded systems, often lack the necessary processing power, memory LLM inference and fine tuning. Their low-power CPUs struggle with the extensive computational complexity, while memory constraints hinder the storage and execution of large-scale models [3]. Additionally, power efficiency is a significant concern, as these devices typically operate on battery power or limited energy sources. The high energy consumption of LLMs, caused by continuous matrix multiplications and memory access, leads to excessive power draw and thermal management issues that can compromise reliability .

[4] Connectivity limitations such as reliance on cloud-based processing introduces latency, dependency on external networks, and potential security risks when transmitting sensitive data . Furthermore, latency and real-time processing requirements pose additional challenges for edge applications like autonomous systems. Another main issue in implementing LLMs in on edge devices is the inefficiency of on device training/ fine tuning which is normally done on cloud rather than on device. edge [5].

Mitigating These Challenges with QLoRa on FPGA To address these challenges, we propose a QLoRa approach implemented in FPGA based on the Vega processor core enabling efficient inferencing and fine-tuning of LLMs on edge devices while optimizing power consumption and computational efficiency. By leveraging FPGA-based acceleration, QLoRa can significantly reduce latency in inference and facilitate fine tuning while reducing the model size, making LLM deployment feasible even in constrained edge environments.

2.3 FPGA for LLMs

FPGAs can address the above limitations through their customizability. Unlike the fixed architectures of CPUs and GPUs, FPGAs can be configured to create hardware that is tailored to meet the specific computational and memory demands of LLMs on embedded/edge devices.

A few examples include:

- Parallizing computations, which would speed up matrix multiplications, one of the most computationally intensive operations in LLMs[6]
- Optimization of memory and power utilization [6]

Which would address concerns such as:

- Performance (by eliminating the need for CPU based inference)
- Data-Privacy (by eliminating the need for cloud-based processing)
- Resource limitations on embedded/edge devices by reducing utilization

Further, FPGA-based LLM accelerators show promising results:

- FTRANS (2020) achieved 16x model size reduction and 27x CPU performance gains [7]
- MHA(2020) reported 14.6x speedups [8]
- Tzanos(2022) achieved 2.3x BERT speedup [9]
- NPE (2021) reduced power consumption by 4-6x [10]

2.4 Transformer Architecture of Large Language Models

First introduced by Google in their paper *Attention Is All You Need*, many LLMs follow the Transformer architecture, which is based on attention mechanisms. The architecture follows an encoder-decoder structure with internal layers for multi-head self-attention and the feed-forward network. The attention function maps queries and key-value pairs to an output which includes performing of matrix multiplication (matmul), scaling, and soft max functions [11]. The transformer model is illustrated below in Figure 1 with linear and non linear operations coloured in red and blue [12].

Transformer architecture is the inference mechanism of an LLM, in which the highest latency and resource utilization are required by the vast amount of matmul operations. Hence, any major improvements to latency should be applied to improving efficient matmul operations to observe overall improvement of the process. Moreover, accelerating other mathematical functions, including softmax, ReLU, and Normalizing, may provide additional improvements in performance.

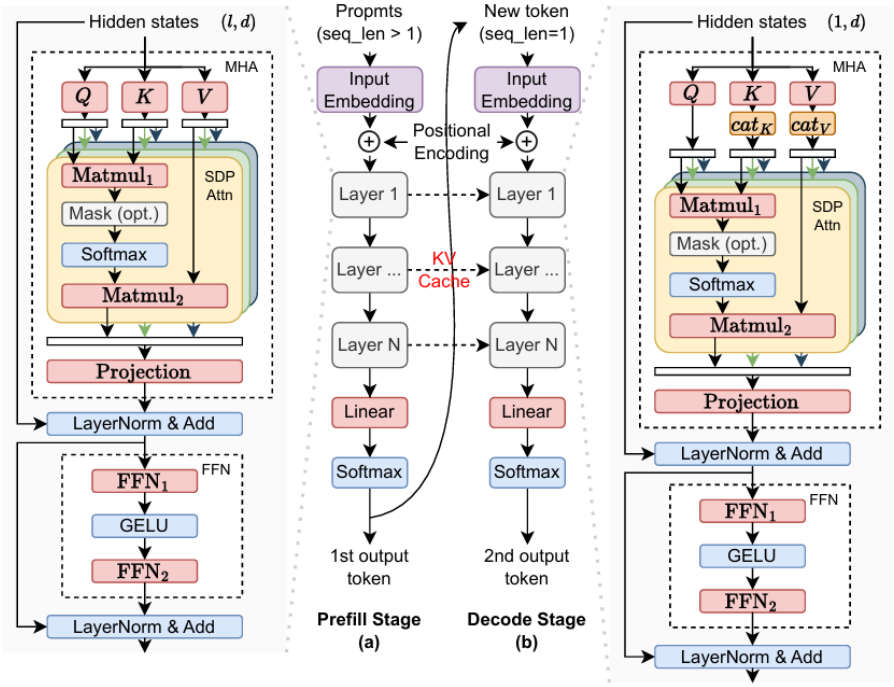

Figure 1: Transformer Model Architecture [12]

3 Goal and Objectives

3.1 Goal

Design and develop hardware accelerator on FPGA for QLoRA fine-tuning of LLMs on edge devices.

3.2 Objectives

- Efficient integration of the accelerator with the VEGA processor.
- Achieve limited memory usage.
- Demonstrate real-world applicability of the accelerator.
- Accuracy/Memory efficiency/power efficiency/speedup

4 Design Process

4.1 Refined Solution

This project focuses on optimizing FPGA resource utilization and computational efficiency through three interlinked components:

1. Matrix Multiplication Acceleration
2. Quantization
3. Dequantization

These components are critical in enabling low-latency inference of large language models (LLMs) under tight FPGA resource constraints. Each has been refined both architecturally and through careful application of High-Level Synthesis (HLS) pragmas to maximize performance and minimize hardware usage.

4.1.1 Matrix Multiplication Acceleration

Matrix multiplication is a core computational primitive in LLMs, especially in attention mechanisms and linear layers. However, direct implementations on FPGAs face significant challenges due to the limited BRAM and DSP resources available. To address this, we adopted a **two-tiered tiling approach** that efficiently utilizes BRAM by loading matrix sub-blocks instead of entire matrices.

- Initially, we referred to an **open-source HLS implementation [13]** of tiled matrix multiplication which only tiled matrix B. While functional, this approach led to **excessive BRAM usage**, far exceeding the capacity available on our target FPGA.
- To resolve this, we extended the design to include **tiled loading of matrix** A as well. This ensured that both matrices are loaded in manageable chunks, drastically reducing BRAM overhead.
- Furthermore, we systematically applied HLS directives:
  - `#pragma HLS pipeline` and `#pragma HLS unroll` for instruction-level and loop-level parallelism.
  - `#pragma HLS array_partition` to enable simultaneous access to multiple elements of on-chip arrays.
  - AXI interfaces were optimized using `m_axi` and `s_axi` to enhance memory throughput.
- The resulting `mmult_accel` module exhibits significantly lower BRAM usage compared to the original design, while sustaining high throughput via efficient memory access patterns.

4.1.2 Quantization and Dequantization

While tiling addresses runtime memory constraints, the model's total parameter storage requirement remains a bottleneck. FPGAs generally do not have sufficient DRAM to store the full-precision parameters of LLMs. To mitigate this, we adopted a quantization strategy inspired by **QLoRA** [2], which reduces model size without sacrificing much accuracy.

4.1.3 NF4 Quantization

We used QLoRA's NF4 (Normal Float 4-bit) format to compress model weights. NF4 preserves distribution-aware quantization while reducing storage to just 4 bits per parameter. Given that NF4 is not natively supported in HLS, we:

- Designed a custom quantization module to convert fixed-point-8 values to NF4.
- Built the corresponding dequantization module to recover approximate fixed-point values during inference.
- Implemented fixed-point arithmetic in HLS for the dequantized values, avoiding costly floating-point operations.

Fixed-Point Arithmetic in `q_2_6` Format
To further reduce FPGA resource consumption, especially the usage of floating-point units (which are scarce and resource-heavy), we adopted a fixed-point format denoted as `q_2_6`, with:

- 2 integer bits
- 6 fractional bits

This format strikes a balance between precision and resource usage. It is used throughout the data path after dequantization and before re-quantization. Both the quantizer and dequantizer were tailored to this fixed-point representation, and verified using test benches.

HLS Optimizations for Quant/Dequant Modules
The quantization and dequantization modules also benefited from targeted HLS optimizations:

- Loop pipelining and unrolling to enhance throughput.
- Array partitioning to enable parallel access to weights and activations.
- Efficient handling of lookup tables and shift operations for fast conversion between fixed-point and quantized formats.

4.1.4 Summary of Refinement

In summary, our refined solution addresses three major challenges in FPGA-based LLM acceleration:

- **Memory usage** is minimized via tiled matrix loading and NF4 quantization.
- **Computation latency** is reduced through HLS-level pipelining and parallelism.
- **Hardware resource efficiency** is achieved by shifting from floating-point to fixed-point arithmetic using the `q_2_6` format.

These optimizations are critical for deploying large-scale models in real-time embedded environments where resource constraints are strict.

4.2 Functional Specification

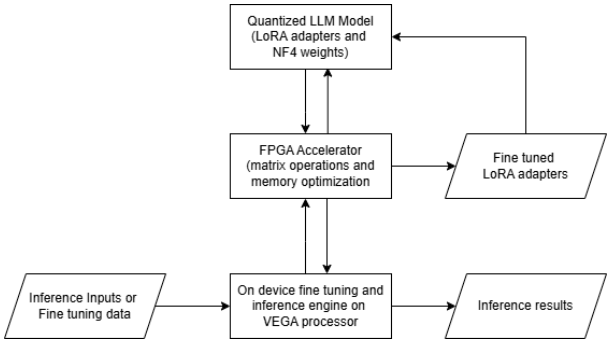

Figure 2: Block Diagram

The functional specification of each IP block is summarized in Table 1.

IP Block	**Input**	**Output**	**Functionality**
Matrix Multiplier	Matrices A, B (`int8_t`)	Matrix C (`int64_t`)	Computes $C = A \times B$ using blocking and tiling strategies optimized for FPGA parallelism.
Quantizer	Fixed-point weights	4-bit NF4 codes (`ap_uint<4>`)	Converts fixed-point weights to NF4 codes using a two-level block quantization scheme.
Dequantizer	4-bit NF4 codes, quant scales	Fixed-point weights (`ap_fixed<8,2>`)	Reconstructs weights from NF4 codes using a two-level scaling hierarchy.

Table 1: Functional specification of key IP blocks.

The design supports matrix dimensions up to and exceeding 768×768 and ensures efficient memory usage via AXI interfaces and burst transfers.

4.3 SoC Design (SoC level block diagram with interfaces/sensors used)

The overall SoC integrates the three IP blocks within the FPGA fabric, interconnected via AXI buses for high-bandwidth memory access. The system architecture includes:

- **External DDR Memory:** Used for storing input matrices, quantized weights, and output matrices.
- **AXI Interconnect:** Facilitates efficient data transfer between memory and IP blocks, supporting burst reads and writes.
- **IP Blocks:** Matrix Multiplier, Quantizer, and Dequantizer, all implemented using HLS for rapid development and optimization.

- **Control Logic:** Manages data movement and orchestrates the execution sequence of the IP blocks.

4.3.1 Accelerator Design Implementation

Matrix Multiplier: The matrix multiplier employs a three-level memory hierarchy (DDR, BRAM, registers) and implements multi-level blocking, processing matrix blocks of 32 rows/columns with inner 4×4 tile computations. The design is fully pipelined with an initiation interval (II) of 1 and features complete loop unrolling for maximum parallelism. Input and output data types are `int8_t` and `int64_t`, respectively. HLS pragmas are extensively used to optimize array partitioning and memory interface bandwidth.

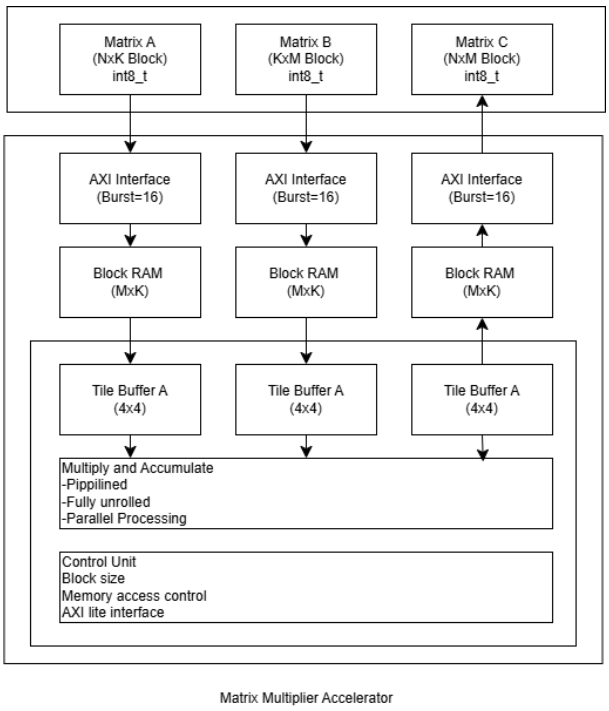

Figure 3: Block Diagram

Quantizer: The quantizer realizes NF4 quantization using a two-level block hierarchy: 64 elements per block at level 1 and 256 elements per block at level 2. Fixed-point weights are converted to 4-bit codes using a lookup table of NF4 values, with internal computations in fixed-point (Q2.6) format. The design is optimized for pipelining, array partitioning, and AXI burst memory transfers using HLS pragmas.

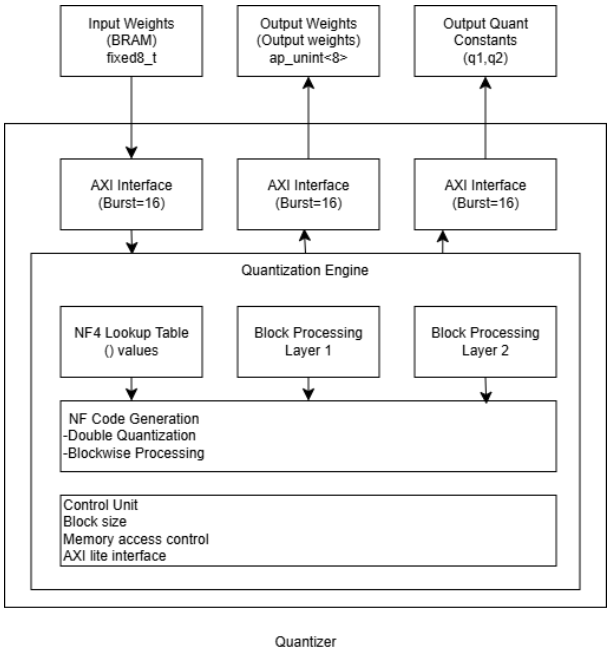

Figure 4: Block Diagram

Dequantizer: The dequantizer mirrors the quantizer's two-level structure to reconstruct original fixed-point weights, applying scaling at both block levels with stored quantization constants. It utilizes BRAM buffers and array partitioning for parallel processing and supports AXI burst read/write operations with loop unrolling and pipeline optimizations.

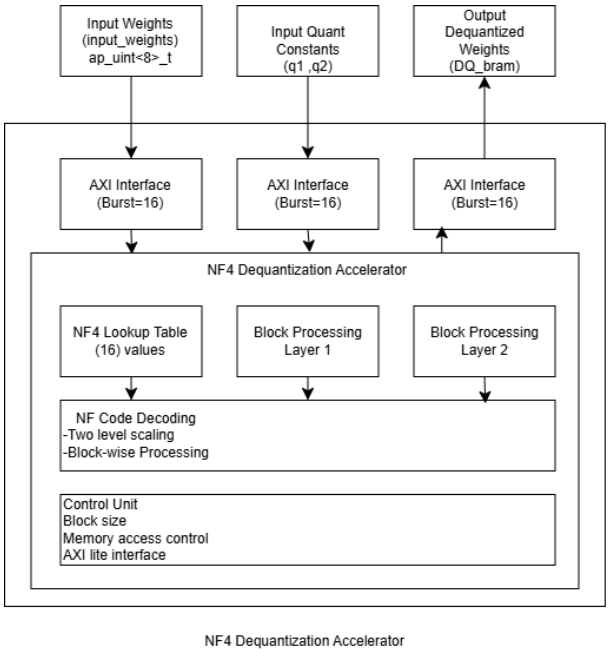

Figure 5: Block Diagram

4.4 Test Plan/Test cases

4.4.1 Matrix Multiplication Accelerator Testbench

Objective The goal of the `mmult_accel_tb.cpp` testbench is to validate the functional correctness and performance of our IP. It specifically targets scenarios common in transformer architectures, where a single input matrix is multiplied by multiple weight matrices (e.g., for Q, K, and V projections).

Testing Methodology Each test case is defined by a tuple (N, K, M) representing matrix dimensions:

- Matrix A: $N \times K$
- Matrix B: $K \times M$
- Output Matrix C: $N \times M$

To mimic the transformer's projection behavior, three different weight matrices (`B_q`, `B_k`, `B_v`) are initialized with deterministic pseudo-random values for reproducibility.

Testing Workflow

1. For each test case, memory is allocated and populated for matrices A, B, and output C.
2. The software reference implementation computes a ground truth result (`C_sw`).
3. The hardware accelerator is invoked to load matrix A into on-chip memory.
4. Hardware outputs (`C_hw`) are compared element-wise against the software result.
5. Timing is measured to compute GFLOPs.

Performance Metrics The testbench measures throughput in GFLOPs (giga floating-point operations per second), calculated as:

$$\text{GFLOPs} = \frac{2 \cdot N \cdot M \cdot K}{\text{Execution Time (s)} \cdot 10^9}$$

Here, $2 \cdot N \cdot M \cdot K$ accounts for both the multiplication and accumulation operations in the matrix product.

Validation and Reporting The testbench logs the best-performing test case in terms of GFLOPs, and prints detailed mismatch reports if any element in `C_hw` deviates from `C_sw`. In case of failure, the testbench highlights the specific index and value mismatch for debugging.

4.4.2 Dequantization Accelerator Testbench

Objective The goal of the `dequant_accel_tb.cpp` testbench is to verify the functional correctness and accuracy of our dequantization IP. This IP reconstructs fixed-point values from quantized data using multiple quantization levels.

Testing Methodology

- Input quantized data `Q_bram` of size $Q\_SIZE = 65536$ elements is initialized with values drawn from a standard normal distribution clipped to [-1, 1] converted to a fixed point representation.
- Quantization outputs include:
  - `output_weights` (size $Q\_SIZE/2$)
  - `output_q1` (size Q_SIZE/LAYER1_BLOCK_SIZE)
  - `output_q2` (size Q_SIZE/(LAYER1_BLOCK_SIZE $\times$ LAYER2_BLOCK_SIZE))
- The dequantization function uses these outputs to reconstruct fixed-point values into `DQ_bram`.
- Root Mean Square Error (RMSE) between original fixed-point inputs and dequantized outputs is computed to assess accuracy.

Testing Workflow

1. Initialize `Q_bram` with pseudo-random fixed-point values sampled from a clipped normal distribution.
2. Perform quantization via `quantize_nf4_q2_6()` to generate compressed weight representations and quantization constants.
3. Run `dequantize_nf4_q2_6()` to reconstruct the original data from quantized weights and constants.
4. Calculate RMSE.
5. Print timing information for quantization and provide sample outputs from weights, quantization constants, and the original versus dequantized values for visual inspection.

Performance Metrics Timing of the quantization step is recorded in milliseconds. RMSE provides a numerical measure of dequantization accuracy, defined as

$$\text{RMSE} = \sqrt{\frac{1}{Q\_SIZE} \sum_{i=0}^{Q\_SIZE-1} (Q\_bram[i] - DQ\_bram[i])^2}.$$

Validation and Reporting The testbench outputs:

- Quantization execution time.
- Sample bytes from `output_weights` and values from `output_q1` and `output_q2` arrays.
- RMSE between the original and dequantized data to confirm the quality of reconstruction.
- Side-by-side comparison of a few original and dequantized values to facilitate debugging.

4.5 Simulation Results of Accelerator (Including Waveforms)

We performed simulation of the developed accelerator using Vitis HLS. This included C simulation, C/RTL co-simulation, and waveform generation to verify the functionality of the IP cores.

The C simulation confirmed that the matrix multiplication IP passed all test cases, producing results that perfectly matched the reference outputs. For the quantization and dequantization IPs, we evaluated the combined quant-dequant flow in a single test bench. The Root Mean Square Error (RMSE) observed for this was 0.323.

Following the assumptions made in QLoRA, that LLM weights typically follow a natural distribution [2], we used a natural distribution to generate random inputs for the dequantizer. Specifically, the input values were drawn from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.3$.

- For a normal distribution, 99% of the data lies within $\pm 2.576\sigma$.
- Therefore, expected data range: Min ≈ -0.7728, Max $\approx +0.7728$

To evaluate the accuracy of the quant-dequant pipeline, we compute the percentage error relative to the expected standard deviation:

$$\text{Percentage Error} = \left(\frac{0.323 - 0.3}{0.3} \right) \times 100\% = 7.84\%$$

This level of error is within tolerable limits for quantized LLM inference and suggests that the quantization and dequantization IPs are functioning as intended.

C Simulation Results

```
Running test case 3: N = 96, K = 128, M = 128
   Computing Q projection...
   Q projection time: 0.0885861 s, GFLOPs: 0.0355104
[DEBUG] test_pass = 1
   Q projection Passed. -84475 : -84475
No mismatches found in validation loop.
   Computing K projection...
   K projection time: 0.0857691 s, GFLOPs: 0.0366767
   K projection Passed.
   Computing V projection...
   V projection time: 0.0859615 s, GFLOPs: 0.0365946
   V projection Passed.
Test case 3 Passed.
Running test case 4: N = 64, K = 768, M = 768
   Computing Q projection...
   Q projection time: 4.11009 s, GFLOPs: 0.0183688
   Q projection Passed. 236188 : 236188
No mismatches found in validation loop.
   Computing K projection...
   K projection time: 4.1444 s, GFLOPs: 0.0182167
   Computing V projection...
   V projection time: 2.44582 s, GFLOPs: 0.0308679
Test case 4 Passed.
All test cases Passed.
Performance Summary:
  Average GFLOPs: 0.0242798
  Best GFLOPs: 0.0366767 (N=96, K=128, M=128)
```

Figure 6: C Simulation of Matrix Multiplication IP

```
Byte 0: 0x1c
Byte 1: 0x53
Byte 2: 0xc5
Byte 3: 0x58
Output Q1 constants:
Q1[0] = 0.734375
Q1[1] = 0.96875
Q1[2] = 1
Q1[3] = 0.640625
Output Q2 constants:
Q2[0] = 1
Q2[1] = 1
Q2[2] = 1
Q2[3] = 1
Root Mean Square Error (RMSE) between original and dequantized: 0.323527
Comparison:
0.46875 0.25
-0.453125 -0.703125
0.046875 0
0.03125 0
-0.09375 -0.109375
0.125 0.140625
0.046875 0
0.109375 0
-0.71875 -0.703125
```

Figure 7: C Simulation of Quantizer + Dequantizer IPs

C/RTL Co-Simulation Results

```
    K projection Passed.
    Computing V projection...
    V projection time: 0.0003216 s, GFLOPs: 0.0254726
    V projection Passed.
Test case 3 Passed.
Running test case 4: N = 8, K = 64, M = 64
    Computing Q projection...
    Q projection time: 0.0002879 s, GFLOPs: 0.227635
[DEBUG] test_pass = 1
    Q projection Passed. -8176 : -8176
No mismatches found in validation loop.
    Computing K projection...
    K projection time: 0.0002811 s, GFLOPs: 0.233141
    K projection Passed.
    Computing V projection...
    V projection time: 0.0002831 s, GFLOPs: 0.231494
    V projection Passed.
Test case 4 Passed.
All test cases Passed.
Performance Summary:
  Average GFLOPs: 0.0528486
  Best GFLOPs: 0.233141 (N=8, K=64, M=64)
```

Figure 8: C/RTL Co-Simulation of Matrix Multiplication IP

```
Output Weights (first 8 bytes):
Byte 0: 0x82
Byte 1: 0x80
Byte 2: 0x82
Byte 3: 0x88
Output Q1 constants:
Q1[0] = 0.734375
Q1[1] = 0.96875
Q1[2] = 1
Q1[3] = 0.640625
Output Q2 constants:
Q2[0] = 1
Q2[1] = 1
Q2[2] = 1
Q2[3] = 1
Root Mean Square Error (RMSE) between original and dequantized: 0.30005
Comparison:
-0.078125 0
0.46875 0
-0.453125 0
0.046875 0
0.03125 0
-0.09375 0
0.125 0
0.046875 0
0.109375 0
-0.71875 0
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

Figure 9: C/RTL Co-Simulation of Quantizer + Dequantizer IPs

Waveforms Generated on Vivado by Co-Simulation

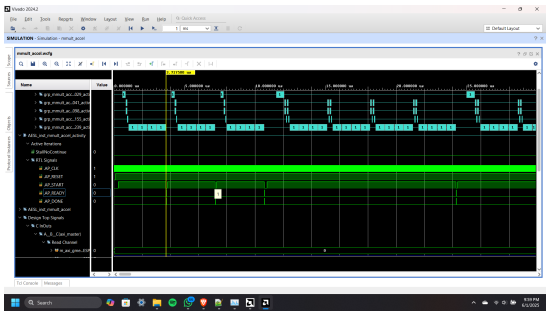

Figure 10: Waveform of Matrix Multiplication IP

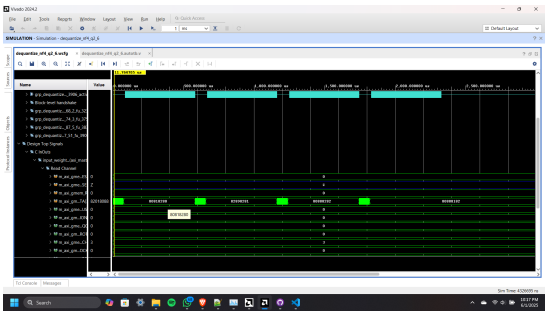

Figure 11: Waveform of Quantizer + Dequantizer IPs

4.6 Synthesis and Resource Utilization

For synthesis, the hardware target device was the `xc7k325tfbg676-2L` FPGA.

Implementation Reports
The following figures present the implementation reports of the three IP cores used in this project.

Resource Utilization Summary Based on the implementation reports, the resource utilization for each IP core is summarized in Table 2. All values are given in percentages (%).

Resource	Matrix Multiplication	Quantizer	Dequantizer	Total
LUT	3	1	1	5
LUTRAM	1	-	-	1
FF	2	1	1	4
BRAM	4	-	-	4
DSP	20	-	-	20
IO	60	33	26	119
BFG	3	3	-	6

Table 2: FPGA Resource Utilization for Each IP Core (Percentages)

As presented in Table 2, the overall resource utilization across the IP cores is relatively low, with the exception of I/O usage. The high I/O utilization is likely due to overhead introduced during the High-Level Synthesis (HLS) to Register Transfer Level (RTL) conversion process. We believe that by implementing the design directly in RTL, this overhead can be significantly reduced.

Due to time constraints, we were unable to complete the RTL implementation within the current project timeline. However, we plan to address this in future work by migrating the design to a full RTL-based implementation, optimizing I/O usage and improving overall efficiency.

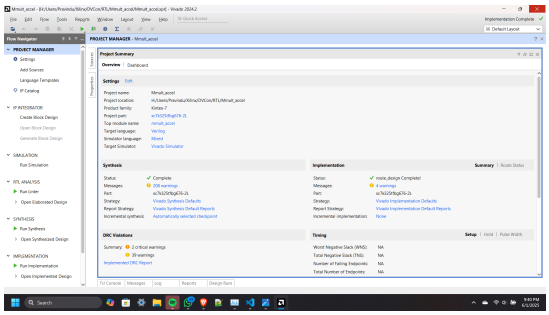

Figure 12: Matrix Multiplication IP Implementation Report - 1

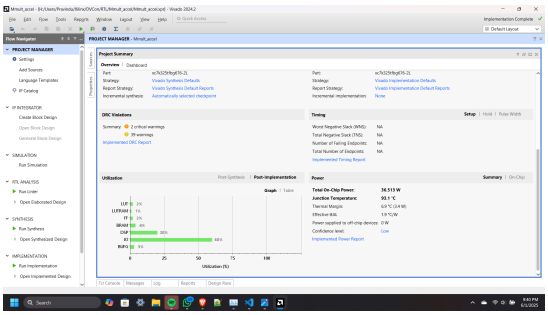

Figure 13: Matrix Multiplication IP Implementation Report - 2

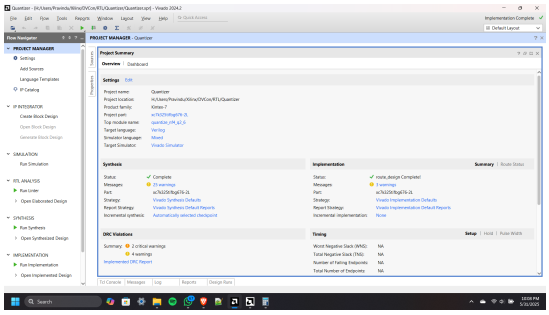

Figure 14: Quantization IP Implementation Report - 1

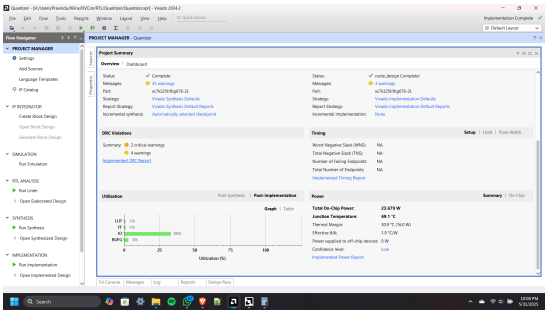

Figure 15: Quantization IP Implementation Report - 2

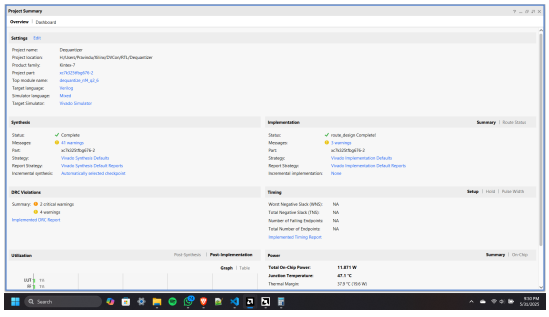

Figure 16: Dequantization IP Implementation Report - 1

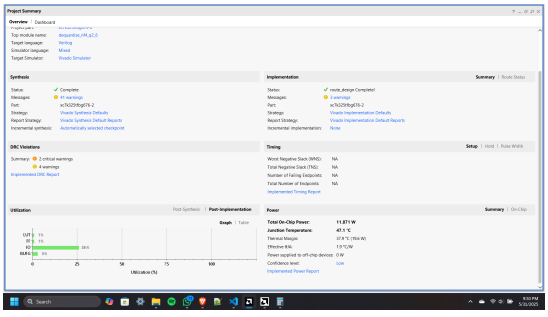

Figure 17: Dequantization IP Implementation Report - 2

5 Results and Discussion

5.1 Positive Outcomes

Our simulation results indicate that all three hardware accelerators—matrix multiplication, quantization, and dequantization—function correctly and efficiently under the current design parameters. Specifically:

- We achieved very low utilization of critical FPGA resources, including BRAMs, DSPs, flip-flops (FFs), and look-up tables (LUTs).
- Although additional optimizations through loop unrolling, pipelining, and array partitioning are feasible, we chose smaller parameter settings to allow scalability for larger matrix sizes. This ensures that our accelerator design remains adaptable for more complex LLM workloads in future iterations.

5.2 Challenges Observed

Despite the overall success, several limitations were encountered:

- **High I/O utilization** was observed, which may become a bottleneck when scaling to system-level deployments or integrating multiple IPs.
- Our current implementation is based on High-Level Synthesis (HLS), which inherently introduces overhead during translation to RTL. Given the project's timeline—alongside ongoing academic and examination obligations—we prioritized rapid prototyping via HLS. Nevertheless, we understand that manually written RTL would yield finer-grained control over hardware resources and performance.

5.3 Future Improvements

To address the above challenges and improve performance, we plan the following enhancements:

- Develop custom RTL modules for each accelerator to replace the synthesized HLS versions. This will enable more precise control over dataflow, timing, and interface logic, and significantly reduce both resource and I/O overhead.
- Integrate the accelerator IPs into a broader high-level inference pipeline for LLMs, supporting end-to-end data movement, quantization, computation, and dequantization in hardware.

Overall, while this stage of the project demonstrates proof-of-concept feasibility, future iterations will focus on refining the hardware implementation to achieve production-grade efficiency and integration.

6 Conclusion

This work presented Edge QLoRA, an FPGA-accelerated implementation of QLoRA designed to enable efficient on-device fine-tuning of large language models.
Key achievements include:

- Successful development of three optimized IP cores (matrix multiplier, quantizer, and dequantizer) with low resource utilization (5% LUTs, 4% FFs, and 20% DSPs)
- Validation of functional correctness through extensive simulation, achieving 7.84% RMSE in the quant-dequant pipeline
- Demonstration of accuracy of IPs through test benches.

While the current HLS-based implementation shows promise, future work will focus on:

- Migrating to a full RTL implementation to reduce I/O utilisation and improve performance
- Expanding support for larger models
- Integrating the IP cores into a complete LLM inference pipeline

Edge QLoRA represents a significant step toward enabling privacy-preserving, low-latency LLM applications on resource-constrained edge devices. The results suggest that FPGA acceleration can effectively bridge the gap between cloud-based LLMs and practical edge deployment in healthcare, IoT, and robotics applications.

7 References

[1] E. J. Hu, Y. Shen, P. Wallis, *et al.*, *Lora: Low-rank adaptation of large language models*, 2021. arXiv: 2106.09685 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2106.09685.

[2] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, *Qlora: Efficient finetuning of quantized llms*, 2023. arXiv: 2305.14314 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2305.14314.

[3] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, *A review on edge large language models: Design, execution, and applications*, Sep. 2024. DOI: 10.48550/arXiv.2410.11845.

[4] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023. DOI: 10.1109/JPROC.2022.3226481.

[5] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023. DOI: 10.1109/JPROC.2022.3226481.

[6] C. Kachris, "A survey on hardware accelerators for large language models," *Applied Sciences*, vol. 15, no. 2, p. 586, Jan. 2025, ISSN: 2076-3417. DOI: 10.3390/app15020586. [Online]. Available: http://dx.doi.org/10.3390/app15020586.

[7] B. Li, S. Pandey, H. Fang, *et al.*, "Ftrans: Energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '20, Boston, Massachusetts: Association for Computing Machinery, 2020, pp. 175–180, ISBN: 9781450370530. DOI: 10.1145/3370748.3406567. [Online]. Available: https://doi.org/10.1145/3370748.3406567.

[8] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, 2020, pp. 84–89. DOI: 10.1109/SOCC49529.2020.9524802.

[9] G. Tzanos, C. Kachris, and D. Soudris, "Hardware acceleration of transformer networks using fpgas," in *2022 Panhellenic Conference on Electronics Telecommunications (PACET)*, 2022, pp. 1–5. DOI: 10.1109/PACET56979.2022.9976354.

[10] H. Khan, A. Khan, Z. Khan, L. B. Huang, K. Wang, and L. He, "Npe: An fpga-based overlay processor for natural language processing," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21, Virtual Event, USA: Association for Computing Machinery, 2021, p. 227, ISBN: 9781450382182. DOI: 10.1145/3431920.3439477. [Online]. Available: https://doi.org/10.1145/3431920.3439477.

[11] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[12] H. Chen, J. Zhang, Y. Du, *et al.*, "Understanding the potential of fpga-based spatial acceleration for large language model inference," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 1, pp. 1–29, Dec. 2024, ISSN: 1936-7414. DOI: 10.1145/3656177. [Online]. Available: http://dx.doi.org/10.1145/3656177.

[13] R. Li and S. Chen, *Design and implementation of an fpga-based hardware accelerator for transformer*, 2025. arXiv: 2503.16731 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2503.16731.