Accelerating LLM Inference on FPGA

LLM Forward Pass on Xilinx Alveo U250

Motivation

▓▓▓ Why This Problem?

LLM inference during autoregressive decoding is **memory-bandwidth bound**.

- Each generated token requires a full forward pass through all layers.
- Every weight matrix is read from DRAM once per token — **no reuse**.
- Compute units often sit idle waiting for data.

Why not just use a GPU?

- INT4/INT2 quantization support is still maturing and often needs software emulation
- High power draw (300–700W) for continuous inference.

Why FPGA?

- Direct control over memory access patterns — stream weights exactly once.
- Hardware-level precision configurability (FP32, INT8, or lower).
- Significantly lower power consumption.

Project Objective

Implement and run the **complete forward pass** of Qwen-3 0.6B on the Xilinx Alveo U250 FPGA.

Specifically:

1. Write all transformer kernels (RMSNorm, MatMul, RoPE, GQA Attention, Softmax, FFN) in Vitis HLS.
2. Verify correctness against a CPU reference at every stage.
3. Generate a working bitstream and measure real hardware throughput.
4. Apply INT8 quantization to improve bandwidth utilization.

> This is a from-scratch HLS implementation — no pre-existing FPGA inference framework was used.

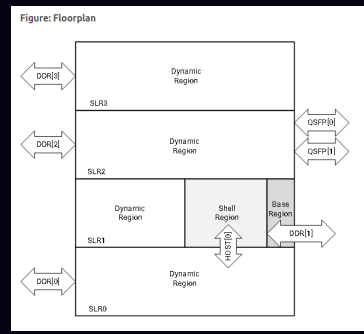
Background

Target Hardware: Xilinx Alveo U250

Key resources

Resource	Available
LUTs	~1.4 M
FFs	~2.9 M
BRAM	2,388
URAM	1,068
DSPs	10,656
DDR4	4 × 16 GB

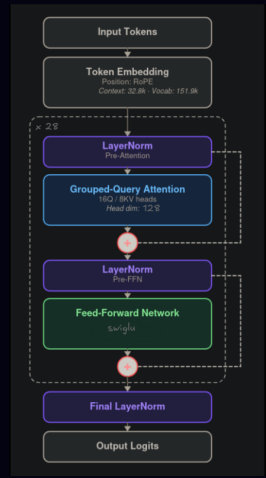
- PCIe Gen3 ×16 (~16 GB/s peak to host)
- 4-SLR die: SLR0 – SLR3
- Qwen-3 0.6B weights: ~2.4 GB (FP32)



Qwen-3 0.6B: Model Architecture

28 decoder layers, each running:

1. **RMSNorm** → normalize activations
2. **Q / K / V projection** → three MatMuls
3. **RoPE** → rotary positional encoding on Q, K
4. **GQA Attention** → grouped-query (16Q / 8KV heads, dim 128)
5. **Softmax** → over attention scores
6. **Output projection** → MatMul back to model dim
7. **RMSNorm** → pre-FFN normalization
8. **FFN** → SwiGLU (gate + up projection, element-wise multiply, down projection)



Every weight tensor is loaded from DDR4 **once per token per layer**. 28 layers × ~85 MB weights = **~2.4 GB read per generated token**.

▓▓▓ Related Work

Work	Key Idea
FlightLLM on-chip decode	Introduces a configurable sparse DSP chain and an scheme for LLM inference.
BitNet b1.58 ternary	Proposes a 1.58-bit LLM variant where every weight is ({-1, 0, +1}).
MatMul-Free LLMs layers	Completely eliminates MatMul operations in both dense and self-attention mechanisms using ternary weights.
TerEffic 1.6-bit ternary	Designs a hardware-software co-design specifically for weights, featuring a custom Ternary Matrix Unit (TMU).

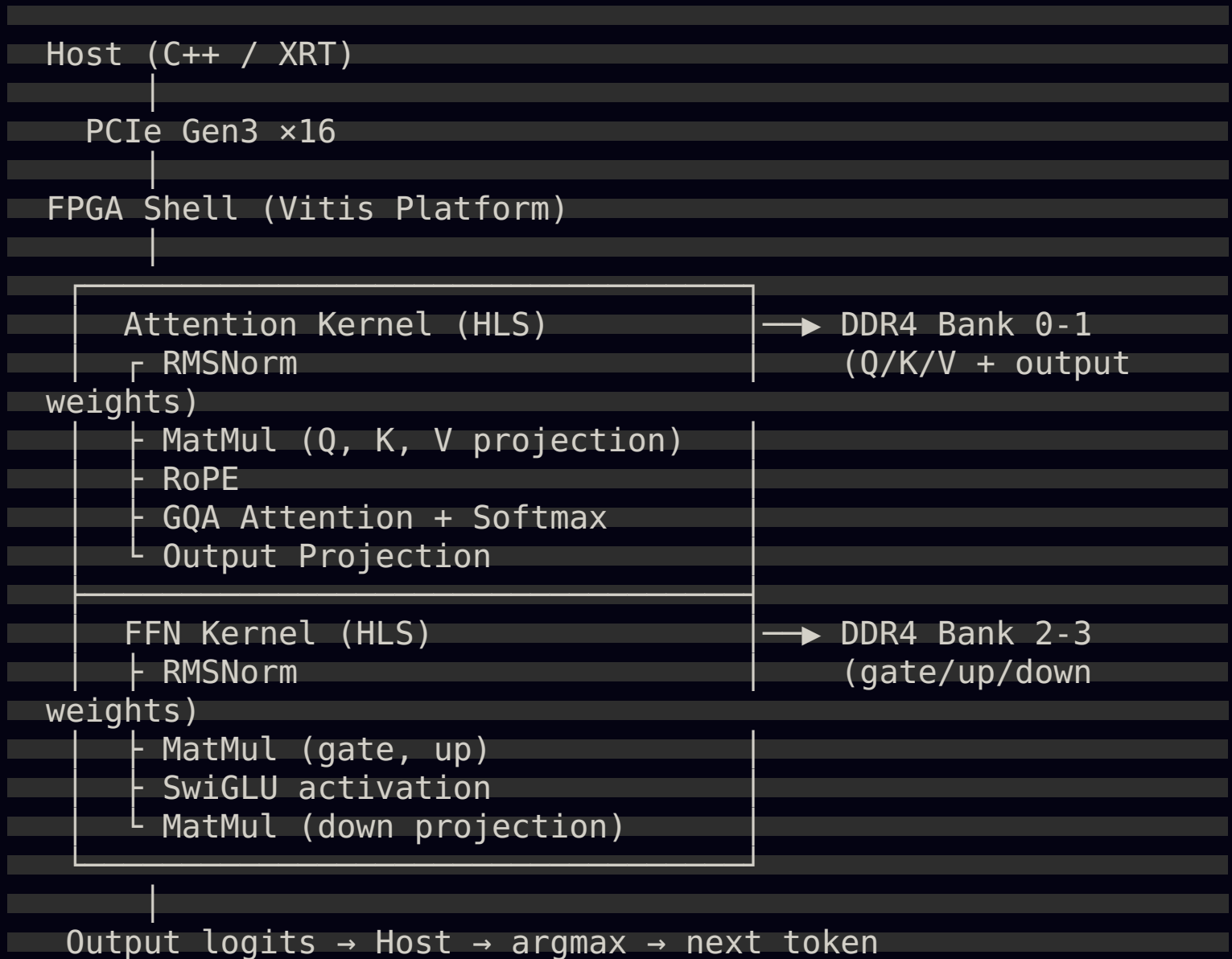
Common lesson: Raw FLOPS don't matter. Memory access patterns and bandwidth utilization determine throughput.

This project applies that insight to a **standard dense transformer** on the U250 — no sparsity, no custom weight formats — to establish a baseline and understand the real bottlenecks.

Design & Implementation

System Architecture

End-to-end execution flow:



▓▓ HLS Kernel Design & Optimizations

Operations are isolated into separate HLS functions, connected via `hls::stream`.

Key HLS Pragmas:

- `DATAFLOW`: Pipelines the full layer so operations run concurrently.
- `PIPELINE II=1`: Maximizes throughput per loop iteration.
- `ARRAY_PARTITION`: Parallelizes memory reads for dot-product accumulation.

Crucial Optimizations Implemented:

- **Tree-Based Reductions:** Used in Softmax and MatMuls to optimize the critical path and hit the II=1 target.
- **Hardcoded Logic:** RoPE and SwiGLU were hardcoded directly into the pipeline to save resources and simplify control flow.
- **Column-Major Streaming (Broadcasting):**
  - *The "Adder Tree" Problem (Row-Major):* Streaming a row of A and a column of B requires a massive spatial adder tree to sum all products in one cycle, costing high LUTs and causing congestion.
  - *The "Broadcasting" Advantage (Column-Major):* By streaming a column of B and broadcasting it to Processing Elements holding rows of A, the reduction becomes temporal (adding to a register over cycles). This drastically saves resources and eases routing.

▓▓ Hardware Challenges: The Routing Problem

The problem was never resource utilization — it was routing.

- Global utilization was **under 30%** across LUT, FF, DSP.
- However, Vivado reported **routing congestion level 6–7** and failed to place all nets.
- Root cause: Multiple `matmul` instances with high fanout spread across SLR2 and SLR3.
- SLR crossing wires under heavy load caused localized net congestion.

How it was mitigated:

1. **Increased Initiation Interval** on MatMuls: Reduced register fanout, bringing congestion down to level 5–6.
2. **Explicit 2-SLR Split:** Attention kernel explicitly placed in SLR2, FFN kernel in SLR3.
3. **Reduced Loop Unroll Factors:** Lowered local resource density in hotspot regions.
4. **Redistributed DDR Bank Assignments:** Balanced memory traffic per SLR.

> *Takeaway:* A design fitting in synthesis doesn't guarantee Vivado can route it. Localized congestion on multi-SLR FPGAs is a primary bottleneck.

▓▓ Verification Strategy

Correctness was strictly verified in **four stages**:

1. **Unit Testbenches:** Each kernel tested independently with synthetic inputs. No numerical divergence from CPU reference (FP32).
2. **Full Pipeline C-Simulation:** All kernels connected in dataflow order. Single-query forward pass run end-to-end. Logits matched exactly.
3. **Hardware Emulation (HW-Emu):** RTL-level simulation of the full kernel pipeline. Verified XRT buffer allocation and kernel enqueue.
4. **On-Hardware Execution:** Bitstream deployed to the U250. End-to-end text generation verified over PCIe.

Results

▓▓ Performance & Latency

System Performance (Qwen-3 0.6B, FP32):

Metric	Measured Value
End-to-End Throughput	**~5.9 tokens/sec**
CPU Baseline(Intel Core i7-1255U)	~5.2 tokens/sec
DDR4 Bandwidth Utilized	~13.2 GB/s
Board Power (FPGA only)	~4.3 W
Total System Power	~16.3 W

Kernel Latency (Per Layer):

Kernel	Average Latency	Best Case	Worst Case
Attention Kernel	12.31 ms	12.31 ms	0.12 sec
FFN Kernel	18.49 ms	18.46 ms	18.51 ms

▓▓ Hardware Utilization Breakdown

Global Device Utilization:

Resource	Used	Available	Utilization %
LUT	404,287	1,726,208	23.4%
FF	599,789	3,452,416	17.3%
BRAM	903.5	2,688	33.6%
URAM	216	1,280	16.8%
DSP	1,897	12,288	15.4%

Kernel-Level Budget Breakdown:

Kernel	LUT	Registers	BRAM	URAM	DSP
Attention	153,904	202,725	327	160	1,290
FFN	65,478	79,946	167	56	594

> The disparity between low global utilization (<35%) and the inability to route without congestion highlights the challenges of spatial architecture design.

Quantization

▓▓ INT8 Quantization

Motivation: If memory bandwidth is the bottleneck, make each weight smaller. INT8 weights are **4× smaller** than FP32. Moving the same number of bytes yields 4× more weights.

What was quantized:

- matmul kernels only. All other operations (Softmax, RMSNorm) remain FP32 to preserve precision.
- **Scheme:** Symmetric per-tensor quantization.

```
scale = max(|W_fp32|) / 127
W_int8 = round(W_fp32 / scale)

// Compute path:
acc += X_int8 × W_int8    →  int32 accumulator
result = int32_acc × scale_W × scale_X    →  back to FP32
```

Current status:

Stage	Status
C-simulation	✓ Verified — output matches FP32 reference
C-synthesis	✓ Completed
HW-emu	✓ Verified
Bitstream generation	✗ In progress — routing congestion

▓▓ INT8 Bitstream: Current Status

The quantized design encounters the **same routing problems** as FP32, since the kernel structure remains identical — only the datapath width has changed.

Approach currently being implemented: A 3-SLR split.

Kernel Group	Target SLR
Q-projection (Attention)	SLR1
K/V projection + Attention scores	SLR2
FFN (gate/up/down) + Output projection	SLR3

Expected impact once bitstream is generated:

Metric	FP32 (Measured)	INT8 (Expected)
Weight memory per token	~2.4 GB	~0.8 GB
DDR Bandwidth required	~13.2 GB/s	~3.8 GB/s
Throughput	~5.9 TPS	**> 5.9 TPS**

> *Note:* The 3-SLR placement is currently hanging during Vivado implementation and is under active debugging.

Analysis & Conclusion

Summary

Accomplishments:

- All core kernels implemented in HLS (RMSNorm, MatMul, GQA, RoPE, Softmax, SwiGLU).
- Unit testbenches, full C-simulation, and HW-emulation verified.
- **FP32 bitstream generated and running on the U250.**
- FP32 throughput measured: **~5.9 TPS at ~4.3W** (matching CPU throughput at a fraction of the power).
- INT8 quantization verified through emulation.

Key Takeaway: The design is fully functional on hardware. The dominant constraint is memory bandwidth utilization and localized routing congestion — perfectly reflecting the fundamental challenges of single-token LLM decode on spatial architectures.

▓▓ Future Work

Short-term:

- Resolve 3-SLR placement congestion in Vivado to successfully generate the INT8 bitstream.
- Measure actual throughput improvement with INT8 on hardware.

Long-term:

- **On-chip KV cache:** Avoid re-reading cached K/V vectors from DDR on every token.
- **Weight prefetching:** Overlap layer `N+1` weight reads with layer `N` compute.
- **Batched Inference:** Move from single-token to multi-token decode to increase arithmetic intensity.

Thank you.