Competetion Report

Team Name: **SEC**  
Participants: Qingyu Guo, Songqiang Xu, Linfeng Zhong, Renjie Wei  
Supervisor: Meng Li, Yuan Wang  
Affiliations: **Peking University**

# Introduction

In recent years, Large Language Models (LLMs) have revolutionized multiple research areas and driven the development of many advanced applications. In this technical report, we introduce the methods and results of our submitted design for the AICAS competition’s final. Our design aims to provide satisfactory LLM inference speed on edge devices. As a brief experimental result, with the specified LLM Qwen2.5-0.5B-Instruct, our design achieves up to **288.5 tok/s** in the prefill stage and **35.2 tok/s** in the decoding stage, which is sufficient enough for many latency-sensitive tasks.

# Challenges

Although LLM is powerful, it’s enormous computation and weight access have mostly limited its deployment to server side. As a suitable platform, FPGA manifests potential in deploying neural network accelerators, but deploying LLMs on FPGAs is still nearly impossible. Several challenges need to be solved in the deployment:

**Bandwidth Undertulization**: On the edge devices, the accessible memory bandwidth is mostly restrited to around 10 GB/s, therefore bandwidth is the most significant bottleneck. However, even under such a restriction, hardware accelerators cannot fully utilize the bandwidth and suffer from bandwidth utilization. Moever, without proper scheduling, the hardware modules can also be left idle, leading to suboptimal performance.

**Complicated Non-Linear Operators**: In the structure of LLMs, there are some complex non-linear operators that generally requires floating-point calculation. For example, the Qwen series, there are Rotary Positional Encoding (RoPE), the Softmax, the Root Mean Square Normalization (RMSNorm), and Sigmoid Linear Unit (SiLU). As they demonstrate high dynamic data range and also requries careful numeric calculation, implementing them on FPGA platform is a big challenge.

To solve the challenges, our design incorporate multiple novel design techniques and a comprehensive hardware architecture. To fully unveil the potential of LLM acceleration on FPGA platform, we did full-stack optimization, including network quantization/compression, and cooperative design of hardware modules. The details will be demonstrated in following sections.

# Overview

Our design is full-stack optimized. To achieve near-theoretical optimal throughput, we incorporated radical optimization means in our design.

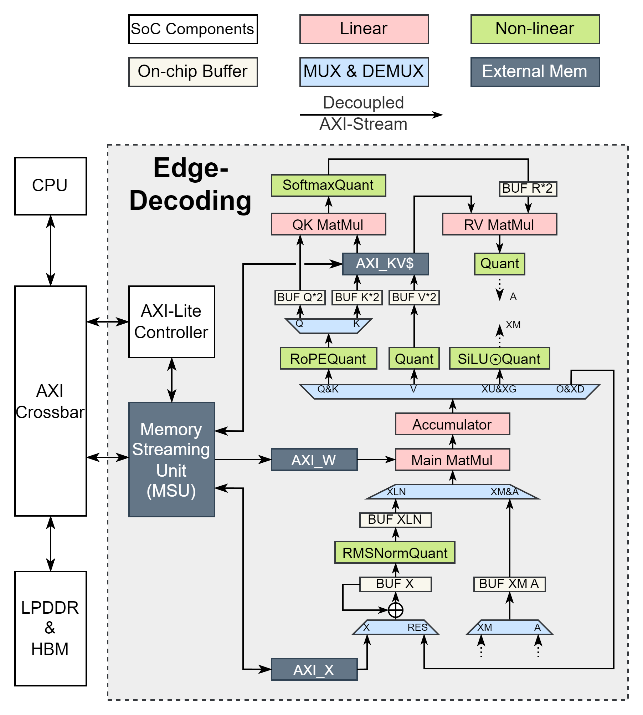


Fig. 1: Diagram of our design.

Our design stands out with the following features:

Domain-Specific Architecture for LLM: the data paths are specialized for the structural characteristics of LLMs, specifically, LLaMA-like models. A single MatMul module handles all static-weight GEMM, with feedback paths enabling efficient output reuse across successive operations. The non-linear operators are implemented with dedicated modules instead of a general non-linear engine. With this dedicated implementation, each module can exploit the unique calculation propertys of each network operator and therefore achieve idea efficiency.

Instruction-less Design: The pipeline operates without instruction-driven control, enabling eager execution, where each stage initiates computation as soon as data arrives. This instruction-less design eliminates instruction decoding, reduces idle time, and improves bandwidth utilization. In an instruction driven design, both the decoding of the instruction and the sequential execution of instructions may incur bandwidth and hardware underutilization. With the instruction flow removed, our design exploit all possible chances of overlapped execution of multiple operators and schedule them as operator, therefore achieve nearly 100% bandwidth utilization.

Unified Memory Management: A single Memory Streaming Unit (MSU) handles all external memory access, optimizing bandwidth by consolidating weight loading into long burst transfers and efficiently managing non-byte-aligned scaling factor packing and unpacking. For example, the weights are quantized to 5-bit, which is not byte-aligned. We use an FSM to unpack the weights. This unified management extracts all memery access patterns into one module, reducing the number of external memory access interfaces, and consolidate memory access behaviors with specific design, therefore achieving near theoretical accessible bandwidth.

The overall design of our system is illustrated in Fig. 1. As shown, the architecture consists of multiple sub-modules, each responsible for a specific functionality within large language model (LLM) inference. For example, the QK Matmul module is specialized for computing the matrix multiplication between the query (Q) matrix and the transposed key (K) matrix. Non-linear operations are also custom-implemented in dedicated modules. All sub-modules communicate using standardized data handshake interfaces, and each is developed as a standalone HLS module for modularity and reuse.

One key component is the Main Matmul module, which handles all matrix multiplications involving static weights. The computation patterns of these matrix multiplications are extracted and hardcoded into the module to maximize efficiency. To decouple computation stages, we introduce an Accumulator module that separates the multiplication and accumulation phases. This is necessary because different matrix operations require different reduction lengths. For instance, generating Q involves reducing over the input channel dimension (896), while the Down projection reduces over the hidden dimension (4864).

By consolidating all static weight accesses into the Main Matmul module and fully pipelining it, we achieve an initiation interval of one cycle. The module is capable of performing 8×8×8 multiplications and accumulations per cycle, reaching a theoretical throughput of 408 GOP/s.

After accumulation, results are routed to different processing branches based on the LLM's computational graph. For example, during multi-head attention (MHA), Q, K, and V are generated in an interleaved fashion: Q and K are sent to the first branch for RoPE (Rotary Positional Encoding), V is sent to the second branch, and the output projection is handled by the fourth branch. In the MLP module, the Up and Gate projections are also interleaved and processed by the third branch, which performs the SiLU activation.

We also optimize for datapath reuse. For example, both the output projection and Down projection results share the same path in the MUX module, since both require addition with the input residual. To support this, the input residual is stored in a dedicated BUF X module. When the output or Down projection result becomes available, it is directly added to the stored residual in-place.

# Operator Fusion: LLM As One Operator

The core principle of our design is "LLM as one operator", which means we fuse all individual transformer sub-operators into a single, unified hardware module. By treating the entire transformer block as one operator, and embedding the unique topology of LLM network into the hardware, we are able to achieve several significant benefits.

First, this approach greatly reduces kernel invocation overhead. All computation is orchestrated internally within the hardware, minimizing the interaction between the processing system (PS) and the programmable logic (PL). Unlike traditional designs that require repeated CPU involvement to launch each hardware function, our design only requires the CPU to configure a few control registers. Once the start register is written, the entire computation proceeds autonomously. This fully decoupled architecture eliminates CPU-induced stalls and minimizes invocation latency.

The hardware interface includes eight control registers: (1) two registers define the start and end decoder layers, (2) four registers define memory addresses: input activation, weight address range (lower and upper bounds), and output activation, (3) one register specifies the current token position (for positional encoding), (4) one start register triggers the computation.

This flexible configuration supports scenarios like the prefill stage, where the output (e.g., from the lm-head) can be optionally skipped. The decoder layer control registers make such bypasses straightforward to implement.

Second, by fully fusing the LLM into a single hardware operator, we significantly improve memory bandwidth efficiency. During the decoding phase, the entire computation is scheduled as a streaming pipeline. This allows for always-on-chip decoding, eliminating most off-chip memory accesses for intermediate tensors such as RMSNorm outputs, SiLU activations, and more.

Furthermore, the use of pipelined modules and multiple FIFOs ensures continuous hardware operation. The only potential stall comes from the AXI memory interface. In our simulation, we observe up to 99% bandwidth utilization, demonstrating the effectiveness of our streaming and fully pipelined design.

# Improve Utilization With Pipeline

The second key principle of our design is the use of deep pipelining to hide computation bubbles and maintain high hardware utilization. By exploiting data locality and organizing operations into well-structured pipelines, the accelerator minimizes idle cycles and ensures smooth execution. Most stalls are avoided, except for a few causality-related bubbles due to dependencies in computation order.

For example, in the Multi-Head Attention (MHA) module, there are 24 independent attention heads. Once the RMSNorm output is ready, all heads can be processed in parallel. As shown in Fig. 2, we demonstrate the overlapped execution of four attention heads. Each head consists of multiple stages—such as QKV generation, QK Matmul, Softmax, and RV Matmul—which are handled by dedicated hardware modules. Their independence enables us to pipeline these stages across heads and overlap computation.

To ensure continuous data flow and eliminate potential stalls, we employ ping-pong buffers for the intermediate tensors: Q, K, V, and R. These double-buffered structures allow the system to write and read data concurrently across pipeline stages. Both linear operators (e.g., matrix multiplications) and non-linear functions (e.g., RoPE and Softmax) are included in the pipeline to maintain steady throughput, especially for the central GEMM module, which is a key performance bottleneck.

The only observable stalls are small and predictable: one introduced by the initial RMSNorm computation and another due to the final head flush. Apart from these, the pipeline remains fully utilized throughout execution, as illustrated in Fig. 2.

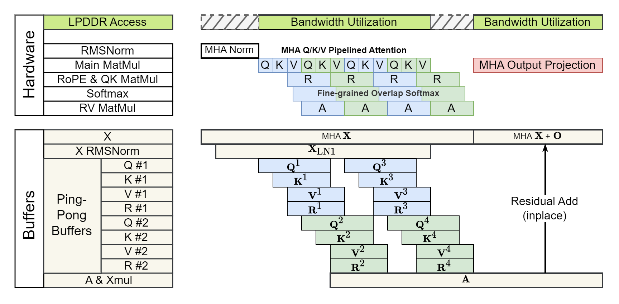


Fig. 2: Pipeline of MHA.

In the MLP module, there are two branches: Up and Gate. In the canonical computation order, these two tensors are generated sequentially, which means the on-chip buffer must be large enough to store both tensors simultaneously. In our design, we fuse the 3 operation (Up, Gate and SiLU) in the granularity of output channel parallelism, greatly reducing on-chip buffer cost.

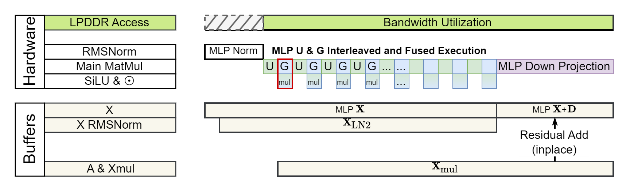


Fig. 3: Pipeline of MLP.

For linear operators with static weights (including QKV generation, Output Projection, Up Projection, Gate Projection, and Down Projection) and dynamic weights (including QK and RV), we designed three GEMM modules to handle the computation as shown in Fig. 1. Our pipeline design keeps the Main GEMM module always busy.

# Higher Resource Efficiency With Integer-Only Quantization

The third principle of our design is “Integer-Only Quantization”. To improve the throughput of a specified model on edge devices, quantization is essential. In most cases, it becomes a significant burden if the quantized model still contains floating-point operations, as this requires FPUs in the hardware. Therefore, we applied thorough integer-only quantization in our design.

Our integer-only quantization method includes the quantization of both linear and non-linear operators. For non-linear functions, we apply lookup table (LUT) methods to convert floating-point and transcendental functions into integer-only tables. The width and depth of these tables are shown in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
|  | #Segments | #Entries | Bits |
| SiLU | 1 | 512 | 16 |
| Rsqrt | 4 | 512 | 21 |
| Exp | 1 | 256 | 9 |
| Recip | 4 | 64 | 17 |

Table. I: Non-linear lookup tables.

Our lookup table method is capable of handling a high output dynamic range using segmented design techniques. With a single lookup table, the output dynamic range is limited by the table's output scaling factor. To improve the dynamic range for functions with large output variations (e.g., Rsqrt, ranging from 2^-3 to 2^12), we split the input tensor range into segments and create separate lookup tables for each segment.

For linear operators, we apply a mixed quantization strategy for activations and weights. To support a wide dynamic range for outliers, we use Grouped PoT (Power of Two) quantization with a group size of 8 for activations. PoT means that the secondary quantization of a group's scaling factor is restricted to powers of two. For weight quantization, we apply APoT (Additive Power of Two), where two shift-based scaling factors are used to reduce the reconstruction loss of the weights. As a result, the computation of GEMM can be fused as follows:

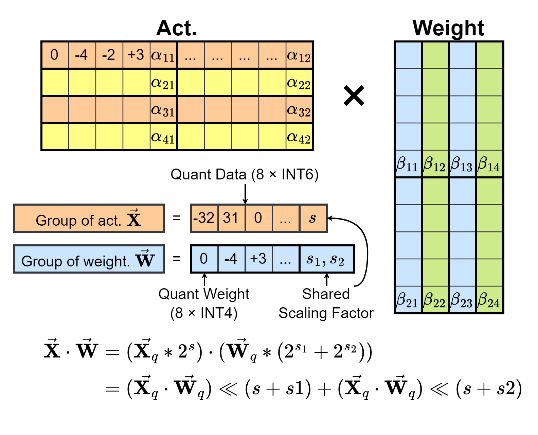


Fig. 4: Fused GEMM of PoT Act and APoT Weight.

# Embedding Table Pruning

During the initial version of our design, we identified the embedding table as a major bottleneck for model compression. Although Qwen-2.5-0.5B is a relatively small model, it retains a very large vocabulary size of 151,936, consistent with other models in the same series. This results in a substantial memory footprint: even after compressing each embedding entry to 12 bits, the embedding table still occupies 194.7 MB.

For comparison, the quantized weights of all decoder layers total 268 MB (at 5-bit precision), and the lm-head occupies 93.3 MB. Hence, the embedding table alone contributes significantly to the overall model size, motivating the need for further optimization.

To address this, we perform embedding table pruning by removing rarely used entries. Specifically, we conduct vocabulary usage analysis across a range of datasets representative of English, Chinese, and programming languages. The datasets include WikiText, AG News, IMDB, Hello-SimpleAI/HC3-Chinese, and CodeSearchNet. The token frequency distributions, shown in log scale, follow a clear power-law distribution: a small subset of tokens accounts for the majority of usage.

Based on this observation, we retain only the top 25% most frequent tokens, reducing the embedding table size to 48.67 MB. Additionally, we leverage a key feature of Qwen-2.5-0.5B: the embedding table and lm-head share weights (i.e., they are tied during training). To take full advantage of this, we perform online quantization at runtime—dynamically generating the quantized lm-head from the pruned and compressed embedding table. The online quantization step also includes weight permutation to further enhance compression efficiency and preserve model accuracy.

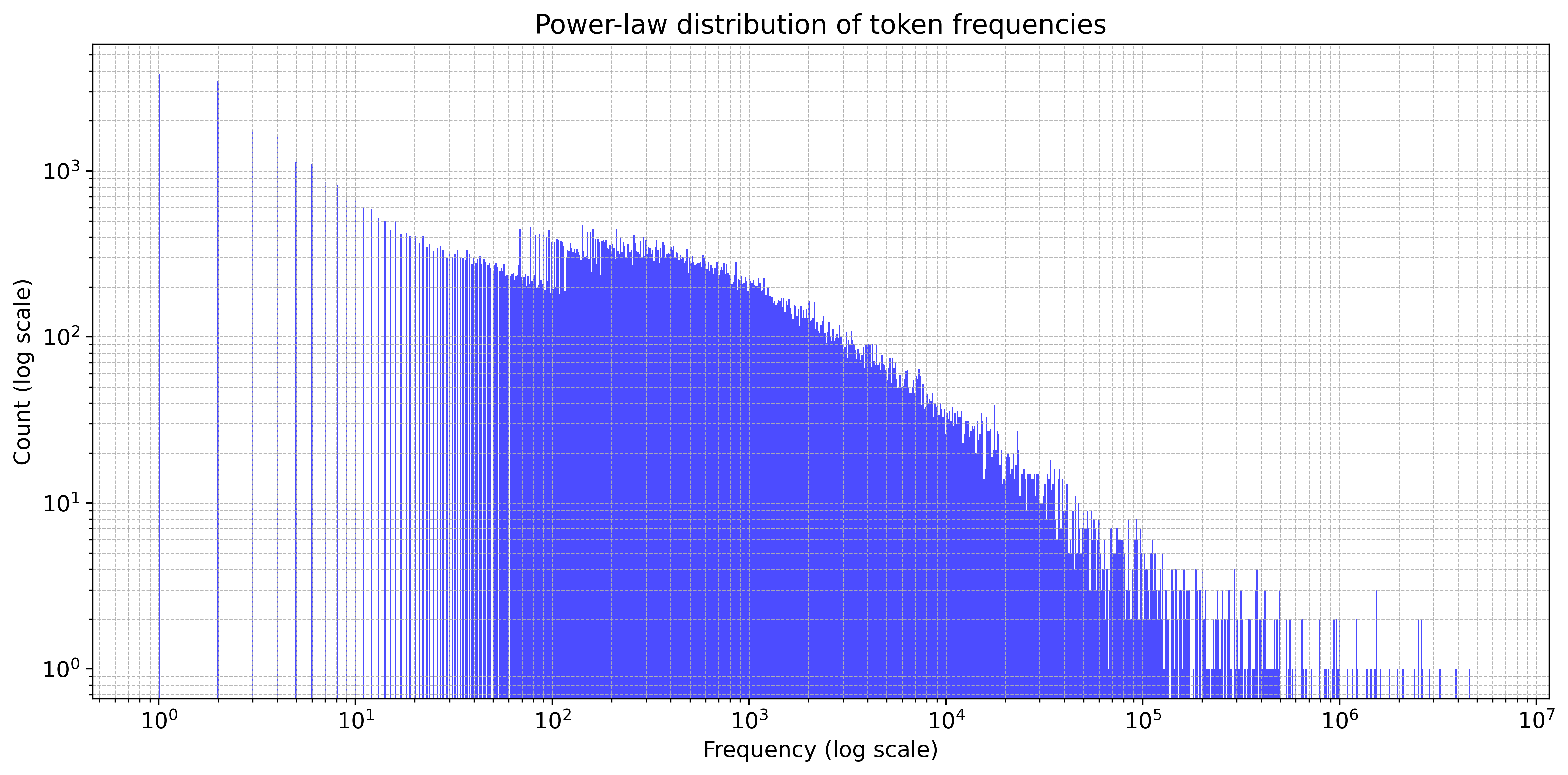


Fig. 5: Bandwidth under-utilization in simulation.

# Memory layout

In our FPGA implementation using the IP Integrator flow, we encountered a hardware constraint: the provided development board (KV260) supports only a 128-bit AXI slave port, which is not directly compatible with our quantized weight layout. In our quantization scheme, weights are grouped with a group size of 8, and the model is executed with a token parallelism of 8. Each weight tile thus consists of 8x(8\*5+6)=320 bits, where: the first 8 is the token parallelism; the second 8 is the group size; 5 is the bitwidth of each quantized weight; 6 is the bitwidth of the pair of power-of-two (PoT) scales per group.

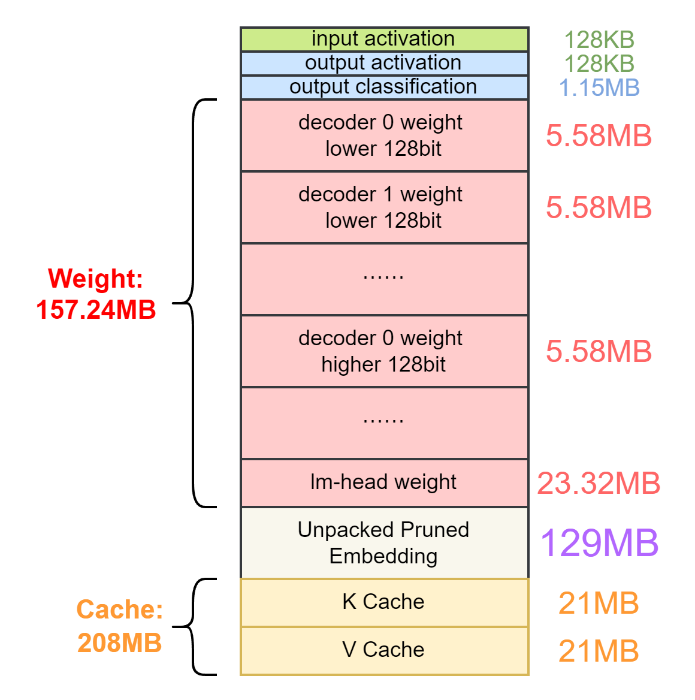


Fig. 6: Bandwidth under-utilization in simulation.

To reconcile the 320-bit weight tiles with the 128-bit AXI bus, we introduce a specialized weight reordering sub-module within the Matrix-Scale Unit (MSU). This sub-module uses the Least Common Multiple (LCM) of the AXI width and tile size as its encoding period, and is capable of extracting complete 320-bit tiles from a stream of 128-bit AXI words.

In contrast to our initial competition version, we now employ two AXI ports to increase available bandwidth. Operating at 375 MHz, this setup provides a total peak memory bandwidth of 11.1 GB/s. To take advantage of both AXI channels, we split each 320-bit tile into two halves—upper and lower 160 bits—and assign each half to a separate AXI port. The reordering sub-module then reconstructs full tiles by concatenating the words retrieved from both ports.

The complete memory layout is illustrated in the figure. Decoder weights are split and interleaved such that each AXI port reads from contiguous memory blocks. Additionally, the embedding table is unpacked prior to being written into memory to support the runtime format. The total size of all weights—including those of the decoders and the lm-head—is 291.16 MB. During the prefill stage, 267.84 MB (excluding the lm-head) must be loaded into memory for each inference.

To reduce bandwidth pressure, all weights are stored in a compact, quantized format. Weight unpacking and reordering are performed on-chip during execution to ensure smooth and efficient data feeding to the compute modules.

# Results

We discuss the experimental results as follows. Some key metrics of the accelerator is demonstrated in Table. II.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LUTs | BRAMs | URAMs | DSPs | Freq |
| 85.6k | 123.5 | 11 | 602 | 375MHz |

Table II. Key metrics of the accelerator.

Throughput: In the measurements, our design achieved a prefill speed of up to 155.3 tokens/s and a decoding speed of 9.3 tokens/s. The prefill speed is significantly higher than the decoding speed because we provide a token parallelism of 8, meaning that 8 tokens are processed in one pass. Additionally, in the prefill stage, LM\_HEAD is not computed, which reduces the required weight loading by approximately 1/4. It is worth noting that we approached the theoretical maximum prefill speed at this frequency. (theoretical: 375MHz \* 128bit ≈ 5.58 GB/s, measured: 11.431MB \* 24 layers \* 155 tok/s / 8 token parallelism ≈ 5.07 GB/s).

Compression Rate: We achieved a compression rate of 0.4444 with our quantization methods. The primary obstacle preventing us from achieving a higher compression rate is the embedding table. While most model weights are compressed to only 5.75 bits, the embedding table remains at 10 bits, exceeding 160 MB in size. We will improve the quantization of embedding table in the future.

Accuracy: On the given GLUE WNLI dataset, we achieved an accuracy of 66.20%, which is even higher than that of the floating-point Qwen model. The primary reason is that the dataset is too small, making the accuracy highly sensitive to hyperparameter changes. In some cases, the quantized model may even achieve better accuracy than the floating-point model.

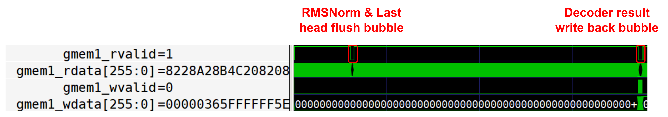


Fig. 7: Bandwidth under-utilization in simulation.

Fig. 7 shows the bandwidth utilization in the simulation. There are only two predictable bubbles in memory access (gmem1\_rvalid). On average, the memory interface experiences only 1% under-utilization.

# Conclusion

We designed a comprehensive neural network accelerator to enhance LLM inference performance on the edge device KV260. Our design incorporates a wide range of optimization techniques to maximize efficiency and eliminate performance bottlenecks.

First, we applied operator fusion to streamline computation and reduce overhead. We then mapped the LLM model topology onto an efficient hardware pipeline, minimizing underutilization of both compute resources and memory bandwidth. To further improve resource efficiency, we adopted integer-only quantization for both linear and non-linear operators. Additionally, we pruned the embedding table to significantly reduce model size and enhance compression.

In our experiments, the accelerator achieved a prefill speed of 288.5 tokens per second and a decode speed of 35.2 tokens per second. Given the constrained resources of the KV260 platform, these results demonstrate that our design is capable of delivering real-time LLM inference for a wide range of edge applications.