Tech Report: FPGA Acceleration for Qwen

Jinwei Zhou[1], Chenhao Xue[2], Xiping Dong[2], Yi Ren[3], Jiaxing Zhang[2], Xinnan Lin[1,*], Guangyu Sun[2,*]
[1]*The Anhui Engineering Research Center of Vehicle Display Integrated Systems*,
Joint Discipline Key Laboratory of Touch Display Materials and Devices,
School of Integrated Circuits, Anhui Polytechnic University, Wuhu 241000, China
[2]*School of Integrated Circuits, Peking University*, Beijing, China
[3]*School of Software and Microelectronics, Peking University*, Beijing, China
jwzhou0915@163.com, {xch927027, gsun}@pku.edu.cn, {dxp, yiren20, zjx}@stu.pku.edu.cn, xnlin@mail.ahpu.edu.cn

***Abstract*—In recent years, large language models (LLMs) have emerged as significant milestones in the advancement of artificial intelligence. Driven by the growing demand for data privacy and energy efficiency, efficient LLM inference in edge scenarios demonstrates substantial application potential. However, the exceptional model sizes and computational demands pose significant challenge for efficient deployment of LLMs on edge devices. In this work, we propose a FPGA-based hardware-software co-optimization framework tailored for efficient edge LLM inference. Our approach encompasses three key components: 1) *W4A8 quantization*, which trades off acceptable accuracy degradation for huge acceleration potential; 2) *operator acceleration*, which effectively utilizes programmable logic (PL) for GEMM/GEMV acceleration, as well as processing system (PS) for the remaining high-precision operations; and 3) *operator scheduling*, which overlaps PS/PL execution to further improve resource utilization. Experimental results demonstrate that the proposed approach efficiently accelerates inference for the Qwen2.5-0.5B-Instruct model on the AMD Kria KV260 platform, achieving 187.9195 tok/s for prefilling and 9.7857 tok/s for decoding, with controllable accuracy loss.**

I. Introduction

In recent years, the rapid advancement of transformer-based large language models (LLMs) have revolutionized the field of artificial intelligence. These models demonstrate exceptional performance in a wide range of tasks, including text comprehension, content generation, and multimodal interactions. Consequently, LLMs have facilitated the emergence of innovative applications, ranging from personalized intelligent assistants to the Internet of Things (IoT),

Despite its substantial application potential, the deployment of LLMs at the edge still presents a significant challenge. Two primary methodologies have emerged for this purpose. The first approach leverages LLM inference services provided by cloud platforms. While this method offers superior performance, it is often constrained by critical concerns such as data privacy vulnerabilities, prohibitive inference costs, and inherent network latency, which often render cloud-based solutions suboptimal in many cases. The second approach involves the local deployment of lightweight LLMs directly on edge devices, yet this method comes with its own challenges: Firstly, the substantial model sizes of LLMs often exceed the storage capacity of edge devices. Secondly, the computational demands of these models typically surpasses the processing capabilities of edge devices. These challenges pose significant barriers to the efficient and practical deployment of LLMs at the edge, underscoring the need for innovative solutions to bridge this gap.

To address these challenges, this paper proposes an embedded FPGA-based acceleration framework to facilitate efficient edge LLM inference. The proposed framework employs a comprehensive hardware-software co-optimization strategy to achieve significant performance improvements. To address the storage challenge, we introduce W4A8 quantization to compress the weight and activation of LLMs, drastically reducing memory capacity and bandwidth requirements. To overcome the computation challenge, we harness both the programmable logic (PL) and processing system (PS) for high-performance operator implementation, and carefully orchestrate dataflow between PL and PS to maximize resource utilization. We validate the effectiveness of the proposed framework on AMD Kria KV260 platform [1], where it significantly improves the inference efficiency of Qwen2.5-0.5B-Instruct model [2]. The main contributions of this work are as follows:

- We propose a FPGA-based hardware-software co-optimization framework for efficient edge LLM inference.
- We adopt W4A8 quantization to compress the model size with negligible model accuracy degradation.
- We implement highly optimized CPU/FPGA kernels and carefully orchestrate PS/PL execution to accelerate LLM inference.
- Experiments demonstrate that the proposed framework achieves 187.9195 tok/s for prefilling and 9.7857 tok/s for decoding when running Qwen-2.5 0.5B on AMD Kria KV260 platform.

The remaining of this paper is organized as follows: Section II introduces model quantization. Section III introduces the implementation of GEMM and GEMV kernels on PL. Section IV introduces the acceleration of the remaining operators on PS. Section V details the operator scheduling between PL and PS. Section VI gives the evaluation results. Finally, section VII concludes this paper.

II. Model Quantization

The original Qwen-2.5 0.5B model in `float32` format requires a minimum of 2GB of memory, occupying a substantial portion of the KV260 platform's limited 4GB total memory. To reduce the memory footprint, we propose to compress the model weights using neural network quantization [3]. Furthermore, to enhance computational efficiency, we also quantize the input activations of linear projection layers. The weight-activation quantization enables the implementation of integer-only matrix multiplication kernels on FPGA, which demands significantly fewer hardware resources compared to their `float`-based counterparts.

Specifically, we adopt the post-training quantization algorithm AWQ [4], leveraging the open-source implementation from LLMC [5]. While AWQ was originally designed for weight-only quantization, LLMC extends this framework to support weight-activation quantization, aligning well with our objectives. AWQ introduces an equivalent transformation in linear layers to preserve salient weights during quantization, which optimize the smoothing factor $\mathbf{s}$ through grid search to minimize the objective in Eq.(1):

$$\min_{\mathbf{s}} ||Q_W(\mathbf{W} \cdot \mathbf{s})Q_X(\mathbf{s}^{-\mathbf{1}} \cdot \mathbf{X}) - \mathbf{W}\mathbf{X}||, \tag{1}$$

Technical report of team *PKUZZZ* for attending IEEE AICAS 2025 Grand Challenge - Large Model Hardware System Design Track.
*Corresponding supervisors.

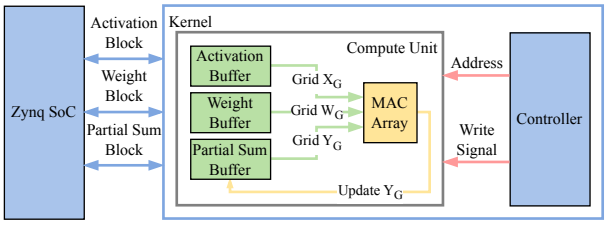

Fig. 1. The architecture of FPGA accelerator.

where $Q_W(\cdot)$ is 4-bit channel-wise quantization, and $Q_X(\cdot)$ is 8-bit token-wise dynamic quantization. Both $Q_W(\cdot)$ and $Q_X(\cdot)$ determine the scaling factor with the maximum absolute value in the quantization group. We randomly sample 128 sequences of length 512 from WikiText2 dataset [6] as calibration data to profile the statistics of activation $\mathbf{X}$. After quantization, LLMC yields a transformed model in `.safetensors` format, where all the smoothing factors $\mathbf{s}$ have been fused to the weights of neighboring operators (e.g. LayerNorm layers, previous linear layers). Therefore, we can directly quantize the transformed model and export quantized weights and the scaling factors into the `.bin` file.

III. FPGA Acceleration

Our profiling of the original `llama.c` implementation reveals that the bottlenecks of LLM inference primarily arise from linear layer computations. To mitigate this limitation, we propose offloading linear operations to specialized FPGA kernels to enhance computational throughput. We develop a parameterized RTL kernel template capable of executing blocked integer matrix multiplication. Given the differing compute-to-memory ratios between the prefilling and decoding stages, we deploy tailored GEMM and GEMV kernels by optimizing the parameter configurations for each phase.

A. Kernel Architecture

Our FPGA kernel consists of three main modules: on-chip buffers, a MAC array, and a controller. The specific architecture is illustrated in Fig. 1, and the following sections elaborate on the implementation of each module.

1) On-Chip Buffer: The FPGA kernel employs three dual-port buffers to improve on-chip data reuse. These buffers are interfaced with DRAM through AXI interconnect to enable data exchange with the PS. Each buffer is dedicated to caching a specific matrix *block*:

- Activation Buffer: Stores a block $X_B \in \mathbb{R}^{B_M \times B_K}$ from activation matrix X;
- Weight Buffer: Stores a block $W_B \in \mathbb{R}^{B_K \times B_N}$ from weight matrix W.
- Accumulation Buffer: Stores a block $Y_B \in \mathbb{R}^{B_M \times B_N}$ that cache the partial-sum of the final result $Y = WX$. Each partial sum is stored with 32b, which is sufficient for the target model.

2) MAC Array: The computational logic is implemented with an array of multiply-accumulate (MAC) units based on inner product. The MAC array fetches a smaller subset of input data (termed as *grid*) from on-chip buffers and performs multiply-accumulate computations. Specifically, it fetches an activation grid $X_G \in \mathbb{R}^{G_M \times G_K}$, a weight grid $W_G \in \mathbb{R}^{G_K \times G_N}$, and a partial sum grid $Y_G \in \mathbb{R}^{G_M \times G_N}$. Upon fetching the input data, the MAC array computes $Y_G \leftarrow Y_G + W_G X_G$, and writes the updated Y_G back to accumulation buffer. Our MAC array consists of two main components: the multiplication engine and the adder tree. To maximize FPGA resource utilization, we merge two multiplication operations into a single DSP within the multiplication engine. Additionally, we divide the adder tree into p pipeline stages to meet timing constraints.

3) Controller: Due to on-chip resource constraint, the on-chip buffer cannot hold the entire matrix of linear layers. In this case, the controller needs to manage the block-wise data transfer between on-chip buffers and off-chip DRAM. Prior to computation, the controller is configured with the number of blocks and the base addresses of the matrices. During computation, It autonomously determines which blocks to read from DRAM and whether to write the final results in the accumulation buffer back to DRAM. Notably, all matrices adopt the customized data layout detailed in Section III-C, which reduces the hardware overhead of calculating block addresses.

TABLE I
HYPERPARAMETER CONFIGURATION OF GEMM AND GEMV KERNELS.

	Param.	Description	GEMM	GEMV
MAC Array	G_M	#rows of X_G / #rows of Y_G.	8	1
	G_K	#columns of X_G / #rows of W_G.	32	32
	G_N	#columns of W_G / #rows of Y_G.	8	8
	p	adder tree pipeline stages	5	5
Buffer	B_M	#rows of X_B / #rows of Y_B.	8	1
	B_K	#rows of X_B / #rows of W_B.	2	2
	B_N	#rows of W_B / #rows of Y_B.	16	16

B. Kernel Implementation

The blocked matrix multiplication kernel is highly parameterized. For both prefilling and decoding stages, it can be configured to facilitate efficient linear operations. TABLE I summarizes the hyperparameters of the FPGA kernels, along with the selected values for GEMM and GEMV operations. To further optimize performance, several key techniques were employed:

1) Double Buffering: Double buffering was implemented to overlap memory transfers with computation, effectively hiding memory latency and maximizing hardware utilization. This technique significantly improved throughput by ensuring that while one buffer is being processed, the other is being loaded with data.

2) Frequency Optimization: To maximize compute capability, we set a higher clock frequency for the kernel. However, the AXI interconnect operates at a maximum frequency of 250 MHz. To address this, we place the MAC array and buffer in a fast clock domain running at 300 MHz, while keeping the AXI master interface and controller in a slow clock domain at 250 MHz. We use asynchronous FIFO to transfer data between the two clock domains.

3) Block Size Selection: The block size is designed to optimally fit our kernel. We utilize 1024 DSPs in the GEMM kernel, allowing it to process 2048 elements per cycle. Therefore, we set the grid size G_M, G_K, G_N to 8, 32, and 8, respectively. A larger G_K is chosen because it requires fewer FFs to store intermediate results, as the partial sum is 32 bits. Both the activation buffer and weight buffer are 4 KB in size, leading us to set B_M, B_K, B_N to 8, 2, 16. For the GEMV kernel, G_M and B_M are set to 1, resulting in a sequence length of 1.

C. Hardware and Software Interface

We wrap the FPGA kernels into C functions `gemm` and `gemv`, which can seamlessly replace the CPU-based matrix multiplications.

1) Data Communication: To enable data transmission between the PS and the PL, we create intermediate buffers using Xilinx Runtime Library (XRT). The XRT buffers are visible to both host program and the on-chip controller, and can be accessed via memory read/write operations. They serve as intermediate proxies between C buffers (created by `calloc`) and on-chip buffers.

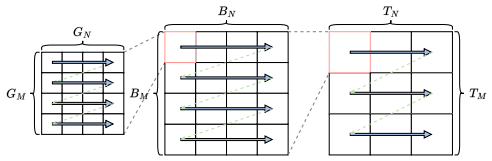

Fig. 2. Data layout: A matrix is reshaped into a six-dimensional tensor of shape $(T_M, T_N, B_M, B_N, G_M, G_N)$. After transposition, the tensor is flattened into a contiguous one-dimensional array following the arrow-indicated traversal order, ensuring efficient data layout.

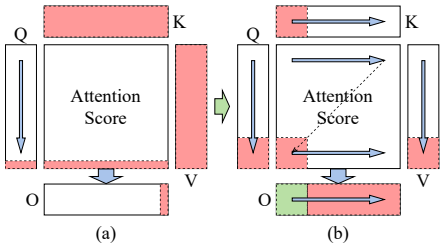

Fig. 3. The original method (a) used GEMV to compute the vector for each token, resulting in low computational intensity. In contrast, our approach (b) utilizes GEMM, which allows for higher data reuse and significantly increases computational intensity.

2) Data Layout: To achieve efficient data exchange between PS and PL, we adopt a customized data layout for quantized matrix in C buffer as illustrated in Fig. 2. A 2-dimensional matrix is organized in a block-by-block manner, i.e. all elements within a block reside contiguously in memory, enabling efficient data transfer between C buffer and on-chip buffer. Similarly, each block is organized in a grid-by-grid manner to achieve efficient on-chip data transfer.

3) Quantization & Dequantization: Finally, we use CPU to compute input activation quantization and accumulated output dequantization within the wrapped C function. Notably, the quantized matrices are stored in the customized layout, whereas the original matrices are still stored in row-wise layout. The difference in data layout necessitates data rearrangement operations. In our implementation, this process are fused into the quantization & dequantization operations.

IV. CPU Acceleration

After offloading GEMM/GEMV operations to programmable logic, the remaining operations executed on CPU become the system bottlenecks. To address this issue, we propose customized CPU operators to alleviate the bottleneck, including block-sparse attention, exponential function approximation, and position embedding precomputation.

A. Block-Sparse Attention

During the prefilling stage, the original `llama.c` attention implementation calculates the $QK^\top$ multiplication token-by-token using GEMV. This requires repeatedly reading the K matrix during computation, which creates a memory bottleneck and leaves computing power underused. To address this, we employ FlashAttention [7], which reduces memory accesses through tiled GEMM and SoftMax computations, as shown in Fig. 3. Moreover, we utilize `sgemm` and `strmm` routines of OpenBLAS [8] to implement block-wise computation, thereby fully leveraging the CPU's computational capacity.

To further accelerate attention, we adopt StreamingLLM [9], which leverages the observed dominance of initial and recent tokens in attention weight distributions within LLMs. By selectively preserving these critical tokens, our implementation achieves faithful approximation of the original SoftMax function while reducing computational complexity from $O(n^2)$ to $O(n)$, where n denotes sequence length.

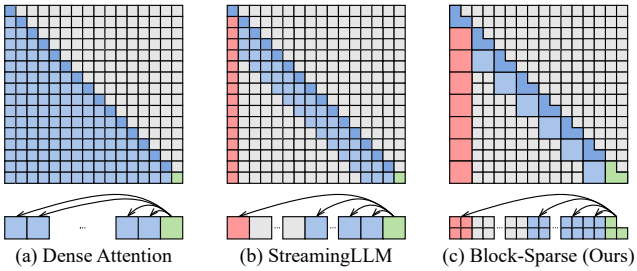

Fig. 4. Composing the blocking of FlashAttention and the sparsity of StreamingLLM, we focus only on the initial tokens and the most recent tokens.

```
float fast_exp(float x) {
    int z = x * 0xb8aa3b + 0x3f7a68c7;
    float y = *(float*) &z;
    return y;
}
```

Listing 1. Efficient Exponential Approximation

We combine FlashAttention with StreamingLLM to develop Block-Sparse attention, as shown in Fig. 4. Our approach preserves the first block and the latest two blocks in attention computation, improving computational density while reducing overall processing overhead. Using a block size of 32 achieves minimal accuracy degradation while enhancing performance.

B. Exponential Function Approximation

Exponential function is instrumental in introducing nonlinearity to transformer-based LLMs, particularly in operations such as SoftMax and SwiGLU [10]. The C standard math library provides off-the-shelf implementation `expf` for `float32`. While offering high precision, the computational efficiency of `expf` is suboptimal. To address this issue, we adopt Schraudolph's algorithm [11] to approximate the original exponential function. The algorithm reformulates the computation of e^x into $2^{z_i} \cdot 2^{z_f}$, where z_i and z_f are the integer and fractional part of $z = \frac{x}{\ln 2}$. The first term 2^{z_i} can be derived from the exponent field of z. The second term 2^{z_j} can be approximated using a polynomial of degree n over the interval $[0, 1)$. Listing 1 shows the C code of approximated exponential function `fast_exp`, where 2^{z_j} is approximated using first-degree polynomial. `fast_exp` reduces the computation to merely one `float32` multiplication and addition while maintaining a maximum relative error of 3% [12]. Empirically, we find `fast_exp` meets the precision requirements of lightweight LLM inference, introducing negligible impact on model accuracy.

C. Position Embedding Precomputation

Rotary Position Embedding (RoPE) [13] is a widely-adopted technique for enabling long-context LLM inference. It encodes absolute token positions using a rotation matrix to preserve relative position dependencies within the self-attention mechanism. However, computing the rotation matrix involves calculating sine and cosine functions, which suffers from poor efficiency on edge CPUs. To address this issue, we propose to replace redundant online computations of sine and cosine functions with a lookup mechanism. Observing that the same rotation matrix can be reused across multiple attention heads, we compute it once and store the results for future reuse. This technique is advantageous as memory access is typically faster than calling `sinf` and `cosf` on edge CPUs. In our implementation, we precompute the rotation matrices for a maximum sequence length of $S_{\text{max}} = 4096$ tokens, and replace `sinf` and `cosf` function calls in the original RoPE function with accessing the buffered results.

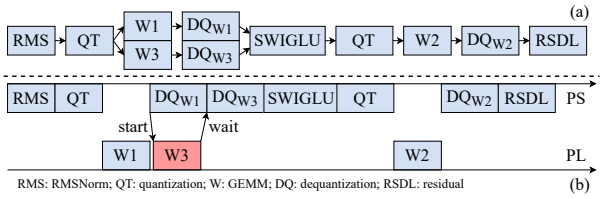

Fig. 5. Operators graph (a) and scheduling (b) for FFN.

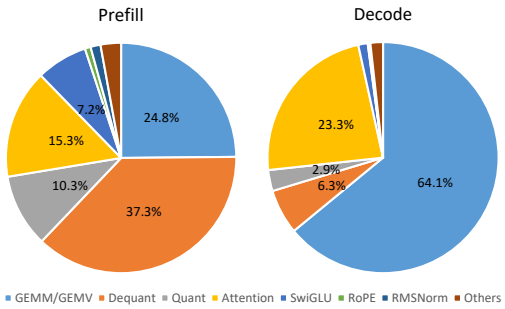

Fig. 6. Operator execution time breakdown.

V. Operator Scheduling

To further enhance performance, we analyzed the operator graph during inference and implemented asynchronous scheduling for parallelizable GEMM/GEMV operations. Taking the Feed-Forward Network (FFN) in each transformer layer illustrated in Fig. 5 as an example, we identified two independent operators: (1) `W1`, the GEMM/GEMV operation with weight W_3, and (2) `DQ_W1`, the dequantization of activations generated by weight W_1. These operations exhibit no data dependency and reside on the PL and PS sides respectively. Consequently, we asynchronously schedule these operators. Specifically, the PS side *starts* the PL-side `W1` operator via a `start` function before executing `DQ_W1`. After completing `DQ_W1`, the PS side then *waits* for PL-side completion using a `wait` function. Similarly, we also implement asynchronous GEMM/GEMV for weights W_K and W_V in the Multi-Head Attention (MHA) of each transformer layer.

VI. Evaluation

A. Evaluation Setup

We evaluate the efficacy of the proposed framework using Qwen2.5-0.5B-Instruct [2] on AMD Kria KV260 platform [1]. We use the official test programs provided by the AICAS 2025 Grand Challenge Committee [14] to evaluate model accuracy and throughput. Model accuracy is measured on two distinct subsets of GLUE WNLI benchmark [15] (preliminary round and final round). Throughput is calculated by dividing the total number of tokens processed by the cumulative time required for the prefilling and decoding stages, respectively.

B. Compression Ratio

Owing to the 4-bit weight quantization, the compressed model in `.bin` format achieves a significant 74.79% size reduction compared to the original model in `.safetensors` format. As we adopt channel-wise quantization granularity, the scaling factors introduce negligible storage overheads.

TABLE II
Model accuracy evaluated on preliminary round benchmark upon progressive inclusion of different optimization methods.

Metric	None	+ Vanilla W4A8	+ AWQ W4A8	+ Block-Sparse Attn.	+ Exp. Approx.
Accuracy	64.79%	54.93%	57.75%	56.34%	56.34%

TABLE III
Speedup of optimized operators and overall throughput

Operator	Linear Layer	Self-Attention	SwiGLU	RoPE	Overall
Prefill	75.1×	13.0×	4.2×	5.9×	65.9×
Decode	4.0×	1.3×	4.2×	5.9×	3.8×

TABLE IV
FPGA resource utilization breakdown

Module	LUT	LUTRAM	FF	BRAM	DSP
KV260 platform	9801(8.36%)	1495(2.59%)	24477(10.45%)	58(40.28%)	0
GEMM kernel	49970(42.67%)	5541(9.62%)	57704(24.63%)	71(49.31%)	1027(82.29%)
GEMV kernel	13026(11.12%)	2800(4.86%)	17274(7.37%)	0	131(10.49%)
Overall	72797(62.16%)	9836(17.08%)	99455(42.46%)	129(89.58%)	1158(92.79%)

C. Model Accuracy

For the proposed optimization methods that may affect model accuracy, we ablate their impact using the preliminary round benchmark, with results presented in TABLE II. We observe that AWQ [4] demonstrates better precision preservation compared with directly applying W4A8 quantization to the original model (Vanilla W4A8). Additionally, the block-sparse attention and exponential function approximation have negligible impact on model accuracy. When integrating all the techniques, our framework achieves 48.03% accuracy on the final round benchmark.

D. Throughput

TABLE III presents the speedup of optimized operators and overall throughput. Our framework achieves 187.9195 tok/s at the prefilling stage and 9.7857 tok/s at the decoding stage, delivering a $65.9\times/3.8\times$ speedup compared to the original `llama.c` implementation. The primary acceleration stems from offloading GEMM/GEMV operations to PL, achieving $75.1\times/4.0\times$ speedup at the prefilling/decoding stage. Additionally, we effectively accelerate bottleneck operators on PS, achieving up to $13.0\times$ speedup. The breakdown of operator execution time is shown in Fig. 6, which reveals that PS and PL consume a comparable time.

E. FPGA Resource Utilization

Table IV summarizes resource utilization. The KV260 platform employs an AXI interconnect linking Zynq SoC with GEMM/GEMV kernel and control logic. In these kernels, LUT and FF form the adder tree, DSP handles multiplication, LUTRAM buffers data from DRAM via FIFO and BRAM store activation, weight, and partial sum blocks.

VII. Conclusion

This paper introduces a comprehensive hardware-software co-optimization framework for efficient LLM inference on embedded FPGA platforms. The proposed approach encompasses W4A8 quantization, operator acceleration on FPGA and CPU, and operator scheduling optimization, enabling significant performance improvement for deploying Qwen2.5-0.5B-Instruct model on AMD Kria KV260 platform.

Acknowledgement

This work is supported in part by the Key Laboratory Founding of Anhui Province (No. 202205p12030001), and in part by the Start-Up Funding of Anhui Polytechnic University (Grant No. 2023YQQ003).

REFERENCES

[1] Xilinx, *Kria KV260 Vision AI Starter Kit Data Sheet (DS986)*, 2024, retrieved from AMD/Xilinx documentation. [Online]. Available: https://docs.amd.com/r/en-US/ds986-kv260-starter-kit

[2] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei *et al.*, "Qwen2. 5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[3] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[4] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 87–100, 2024.

[5] R. Gong, Y. Yong, S. Gu, Y. Huang, C. Lv, Y. Zhang, D. Tao, and X. Liu, "Llmc: Benchmarking large language model quantization with a versatile compression toolkit," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024, pp. 132–152.

[6] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[7] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.

[8] OpenMathLib, "Openblas: An optimized blas library," http://www.openblas.net, 2025, accessed: 2025-03-19.

[9] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," *arXiv preprint arXiv:2309.17453*, 2023.

[10] N. Shazeer, "Glu variants improve transformer," *arXiv preprint arXiv:2002.05202*, 2020.

[11] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.

[12] L. Moroz, V. Samotyy, Z. Kokosiński, and P. Gepner, "Simple multiple precision algorithms for exponential functions [tips & tricks]," *IEEE Signal Processing Magazine*, vol. 39, no. 4, pp. 130–137, 2022.

[13] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[14] I. AICAS, "Ieee aicas 2025 grand challenge - llm hardware system design," https://tianchi.aliyun.com/competition/entrance/description, 2025, accessed: [Insert Date].

[15] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.