

F-BFQ: Flexible Block Floating-Point Quantization Accelerator for LLMs

Jude Haris, José Cano

School of Computing Science, University of Glasgow, Scotland, UK

Abstract—Large Language Models (LLMs) have become increasingly prominent for daily tasks, from improving sound-to-text translation to generating additional frames for the latest video games. With the help of LLM inference frameworks, such as *llama.cpp*, which support optimizations such as KV-caching and quantization, it is now easier than ever to deploy LLMs on edge devices. Quantization is fundamental to enable LLMs on resource-constrained edge devices, and *llama.cpp* utilizes block floating point (BFP) quantization to drastically reduce the bit width of weights and input tensors, the memory footprint, and the computational power required to run LLMs. LLMs are typically quantized with mixed BFP quantization across the model layers to reduce the loss of model accuracy due to quantization. Therefore, to efficiently accelerate across the layers of BFP-quantized LLMs, specialized accelerators need to support different BFP variants without reconfiguration.

To address this issue, we propose a Flexible Block Floating-Point Quantization (F-BFQ) accelerator, which can dynamically switch between two BFP quantization variants and perform matrix multiplication (MatMul) operations. Our initial F-BFQ accelerator design, deployed on the AMD Kria board, reduces inference time by $1.4\times$ on average over the Arm NEON-based CPU execution across three BFP quantized LLMs while achieving 5.2 tokens per second (~ 3.9 words per second).

I. INTRODUCTION

Large language models (LLMs) such as the Llama [1] and GPT [2], [3] family of models have revolutionized the ability of Artificial Intelligence (AI) systems to understand and generate human language in terms of text, audio, or video. LLMs are an emerging class of machine learning (ML) models that are built by learning from huge text-based datasets. With the innovation in model architecture and training methods, and through the help of the popularity of online services like ChatGPT [4], the field of LLMs is evolving rapidly. The number of users is also growing rapidly due to the countless applications and use-cases from classification [5], code generation [6], translation [7] to healthcare [8].

Cloud-based LLM services like Gemini [9] have become the go-to method for daily users to access to LLMs. However, as the availability of open-source LLMs and datasets has increased, especially over the last few years, the need for edge-based, localized access and execution of LLMs has become more sought after due to concerns over security and data privacy. The latest community-driven pushes have facilitated easy access to LLMs and rapid prototyping of new models and optimizations, enabling efficient LLM inference on edge devices. The GPT-Generated Model Language [10] (GGML), a tensor library for ML specialized to enable high performance for LLMs on commodity hardware, is at the forefront of these

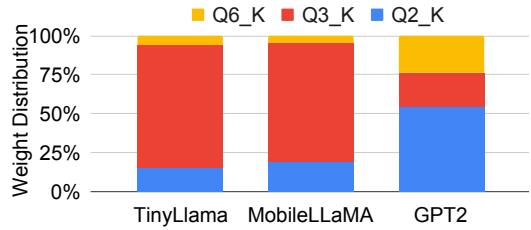


Fig. 1: Weights quantization distribution percentage breakdown of the three LLMs under study.

pushes. Furthermore, the *llama.cpp* inference framework [11], which is based on the GGML library, is specialized towards running LLMs on edge devices, supporting LLM inference on commodity CPUs, GPUs and NPUs.

Unfortunately, LLMs can be very computationally demanding, even for inference. In addition, due to their large memory footprint, they require high memory capacity and bandwidth. These properties of LLMs make them challenging to execute on resource-constrained edge devices. For example, running LLMs on mobile phones or Internet-of-Things (IoT) devices is sometimes impossible due to memory constraints. To improve performance at the edge, FPGA-based (Field Programmable Gate Arrays) accelerators [12], [13] have been developed to outperform standard CPU-based inference, but the problem of model size and inference memory footprint is still a limiting factor due to limited memory of the on-chip accelerator.

Quantization is one of the key optimizations that is used to run LLMs on resource-constrained edge devices [14]. Many quantization schemes have been developed and adopted to enable reduction of LLM sizes. *llama.cpp* has specifically embraced the block floating point (BFP) quantization scheme, which enables blocks of data to be quantized in a group-like manner to further reduce the bit width. To ensure minimal LLM accuracy reduction during quantization, statistical analysis is typically used to determine the level of quantization that can be applied without losing model accuracy. To achieve the best trade-off between LLM accuracy and reduction in size, the level of quantization applied is chosen on a per-layer basis, depending on the weights of each layer.

Figure 1 provides a breakdown of the distribution of weight parameters across three LLMs, GPT2 [2], MobileLLaMA [15] and TinyLlama [16] quantized using *llama.cpp*-based BFP quantization. These models use three different variants of BFP quantization ($Q2_K$, $Q3_K$, and $Q6_K$) for the matrix multi-

plication (MatMul) operations. Hence, to efficiently accelerate these quantized models, an accelerator that supports multiple BFP quantization variants is needed.

In this work, we propose a new Flexible Block Floating-Point Quantization (F-BFQ) accelerator that supports multiple BFP quantization variants to perform the MatMul operation with LLMs. Our initial F-BFQ accelerator design focuses on supporting the $Q2_K$ and $Q3_K$ variants as a proof of concept. We designed and evaluated our accelerator using the SECDA-LLM [17] platform across three different LLMs. The contributions of this work are as follows:

- *F-BFQ*, a new Flexible Block Floating-Point Quantization accelerator that supports $Q2_K$ and $Q3_K$ BFP MatMul operations.
- *Dynamic Super-Block Vector Processor Unit*, a hardware module design that processes $Q2_K$ and $Q3_K$ operations concurrently, an is able to dynamically switch data loading and output accumulation depending on the required quantization variant per layer/operation.
- Evaluation of our initial accelerator across three LLMs (GPT2 [2], MobileLLaMA [15] and TinyLlama [16]), achieving an average speedup of $1.4\times$ over the baseline execution of the Arm NEON-based CPU with an average of 5.2 tokens per second (~ 3.9 words per second).

II. BACKGROUND AND RELATED WORK

A. Large Language Models

Large Language Models (LLMs) are a class of machine learning models built upon the Transformer architecture [18] and pre-trained on massive language corpora. Typically, these models are adapted to specific downstream tasks, such as question-answering, via fine-tuning on task-relevant datasets. LLMs are characterized by their large parameter counts which increases memory footprint; for instance, LLaMA models start with 7 billion parameters [19]. Many LLMs operate in an auto-regressive manner, predicting the next tokens (or words) based on previously cached context. This approach, known as key-value (KV) caching [20], enhances performance but incurs linearly scaling memory overhead.

Quantization techniques are widely adopted to reduce the parameter count and enable the deployment of LLMs on resource-constrained edge devices. For example, 8-bit quantization has been shown to maintain accuracy while reducing model size by up to $4\times$ [21], [22]. Researchers have also explored more aggressive strategies, such as 4-bit quantization, to further shrink models without compromising accuracy [23]. In this work we look at a more promising approach, block floating point (BFP) quantization, which has been evaluated against traditional integer quantization methods [24] for its potential advantages in efficiency and performance.

B. BFP Quantization in *llama.cpp*

llama.cpp [11] is a pure C/C++ library with minimal external dependencies for enabling LLMs inference on a wide range of edge devices that support GCC/Clang. Currently, *llama.cpp* supports a wide set of LLMs, including some multi-modal

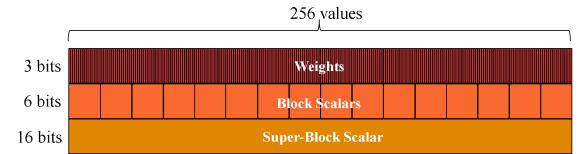


Fig. 2: $Q3_K$ super-block Data Format.

and custom-defined models. With the GPT-Generated Unified Format model format (GGUF) used by *llama.cpp*, it is possible to represent the weights of an LLM with as few as 1.5 bits using BFP quantization. These quantized weights enable users to run LLMs on resource-constrained edge devices such as the Raspberry Pi and the Pixel phone [25].

llama.cpp [11] supports BFP quantization variants of 1.5, 2, 3, 4, 5, 6, and 8 bits along with some additional quantization techniques to recover accuracy. These variants are typically denoted as Qx_y , where x represents the number of bits per weight and y denotes the type of quantization. For example, Figure 2 shows the $Q3_K$ [26] BFP format, where a superblock (SB) represents 256 weights and is partitioned into 16 blocks. Each block contains 16 weights and a block-scaling factor BSF of 6-bits. Each SB also has a super-scaling factor SSF of 16 bits. The BSF and SSF values are used to rescale the weights and ensure minimal accuracy loss. By summing up the total number of bits required for the weights and the scaling factors and then dividing by the number of weights, we can determine that the BFP format requires ~ 3.5 bits per weight, which is a significant reduction compared to the typical 32-bit floating-point format used in LLMs. The $Q2_K$ [26] format differs from $Q3_K$, as it contains 2-bit weights, 4-bit ‘minimum’ and ‘scalar’ values for each block (total 8-bits per block), and 16-bit ‘minimum’ and ‘scalar’ values for the superblock (total 32-bits per superblock). Overall, it requires around ~ 2.6 bits per weight. Note that the $Q8_K$ format is used for input tensors, where each SB contains 256 input values and a single SSF of 16 bits, thus requiring ~ 8 bits per input.

C. BFP Acceleration

Some previous works focus on supporting and accelerating BFP quantization for DNNs. FlexBlock [27] proposes a new accelerator for DNN training with multiple BFP support. By dynamically switching to lower BFP bit widths, it allows up to $16\times$ more multiple-accumulate (MAC) operations than a static bit-width solution. In terms of inference, Bucket Getter [28] is a novel architecture for DNN inference that utilizes multiple smaller accumulators within their custom floating point adder to support different scales of BFP quantization operations.

F-BFQ focuses on LLM inference with the GGUF BFP format, which is widely used in the *llama.cpp* framework.

III. F-BFQ ACCELERATOR ARCHITECTURE

Utilizing our SECDA-LLM [17] platform we were able to quickly and efficiently design our Flexible Block Floating-Point Quantized (F-BFQ) accelerator architecture. The goal of

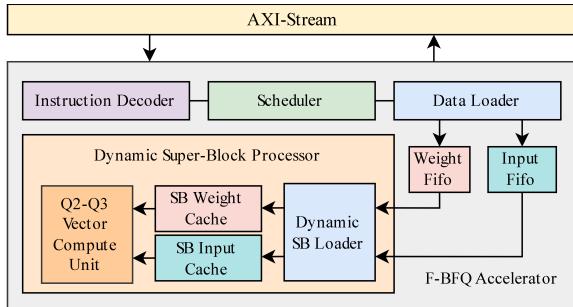


Fig. 3: Overview of our F-BFQ accelerator design for $Q2_K$ and $Q3_K$ MatMul operations.

TABLE I: Micro-ISA Opcode Set.

Opcode	Description
0x01	Configure DSBP (Sets configuration registers)
0x02	Load Weights (Configures Data Loader to load Weights)
0x04	Load Input (Configures Data Loader to load Inputs)
0x08	Schedule MatMul Operation (Activates DSBP)
0x10	Store Output (Sends Output Data to Main Memory)

the initial design is to provide a template architecture which supports two variants of BFP quantization.

The rest of this section provides: i) an overview of the F-BFQ Architecture; ii) descriptions of the hardware modules; iii) in-depth look into the dynamic super-block processor; iv) and a discussion about the software driver, including the opcode generation and tiling.

A. Overview

Our F-BFQ accelerator design aimed to create a scalable architecture that could support efficient processing of BFP-quantized MatMul operations. The proposed accelerator architecture, shown in Figure 3, contains an *instruction decoder*, a *data loader*, a *scheduler* and the *Dynamic Super-Block Processor (DSBP)*, along with data FIFOs that are used to temporary store weight and input data. The following sections describe the details of the main hardware components of the accelerator.

1) *Instruction Decoder*: loads and decodes opcode-based instructions from the AXI-Stream (the data interface to main memory) and then communicates the instruction throughout the rest of the accelerator. Table I shows the micro-ISA opcode set for the accelerator. The opcodes are generated by the accelerator driver and are sent via the AXI-stream. Note that some opcodes are immediately followed by operand data, which the accelerator expects once the instruction is decoded. For example, after opcode ‘0x02’, a set of weight SBs are expected and parsed by the data loader.

2) *Data Loader*: parses the incoming data stream and maps the weight and input SBs into their respective data FIFOs; the number of SBs loaded is configured with the ‘0x01’ opcode. It also partitions consecutive data elements in depth dimension across N FIFOs, enabling parallel access to data so that the *DSBP* can compute N operations simultaneously without stalling the computation pipeline.

3) *Scheduler*: tiles the MatMul problem according to the dimensions of the target layer and controls the *DSBP*. Additionally, it synchronizes and accumulates the output data produced by the *DSBP* and sends the results back to main memory via the AXI-Stream.

B. Dynamic Super Block Processor

The Dynamic Super Block Processor (DSBP) is the core processor within the F-BFQ accelerator. While our current design contains a single DSBP, the F-BFQ accelerator is designed to be modular and scalable with multiple DSBPs. The DSBP consists of four main components: Dynamic SB Loader, SB Weight Cache, SB Input Cache and the (Q2-Q3) Vector Compute Unit. Figure 4 contains a detailed view of the DSBP architecture and its components.

The Dynamic SB Loader is responsible for loading data from the Input and Weight FIFOs into the respective local SB cache. First, the ‘fifo reader’ reads the data, then according to the type (input/weight) and the variant ($Q2_K$, $Q3_K$ or $Q8_K$), the ‘bit-slicer’ slices the data packets into the different parameters, and finally the ‘data mapper’ stores the data into the correct buffers within the SB caches.

For example, when $Q3_K$ SB weight data is being read, the bit-slicer and the data mapper partition and store the SB data into ‘w_scales’, ‘w_low’ and the ‘w_high’ buffers within the SB weight cache according to the $Q3_K$ SB data format shown in 2. The input and weight caches are stored in partitioned BRAM buffers, enabling the vector compute unit to access consecutive data in parallel.

The vector compute unit (VCU), in our case the Q2-Q3 VCU, supports both $Q2_K$ and $Q3_K$ SB vector operations. To efficiently handle both variants while maintaining low resource overhead, the VCU contains a common ‘vector engine’ that can perform the dot product operation required for matrix multiplication. The remaining scaling operations, which are dependent on the quantization variant, are handled by ‘Q2/Q3 Scalar Units’. Finally, a multiplexer (‘Mux’) is used to accumulate the correct output value in the accumulator register (‘Acc’). This accumulator register is then read and saved back to main memory.

C. F-BFQ Driver

The accelerator driver is a key component to enable seamless operation between the SECDA-LLM [17] backend and the accelerator. Our F-BFQ driver is configured to receive all MatMul operations that use $Q2_K$ or $Q3_K$ weights and $Q8_K$ inputs. The driver is responsible for generating and sending opcode instructions through an AXI-Stream data transfer to control the accelerator: i) First, the driver receives the necessary metadata to configure the accelerator from the SECDA-LLM backend; using this, it sends the ‘0x01’ opcode to configure the DSBP with the corresponding dimension of the MatMul operation. Additionally, the driver updates the ‘weight_type’ control register which decides if the accelerator is in $Q2_K$ or $Q3_K$ mode; ii) Second, the accelerator checks if the input matrix is small enough to be stored inside the

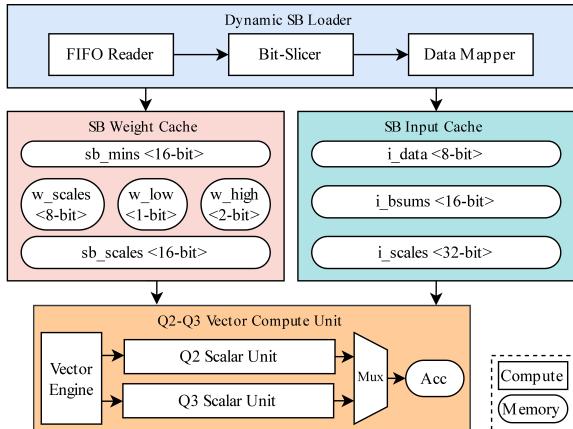


Fig. 4: Detailed view of the Dynamic Super Block Processor.

TABLE II: Specifications of the KV260 board. We highlight the FPGA hardware resource used in terms of absolute value and % of the total available. The FPGA runs at 200MHz.

Device	CPU	DRAM	BRAM	DSP	FF	LUT
KV260	ARM-A53	4GB	234 (81%)	175 (14%)	14K (6%)	36K (30%)

input buffer without tiling; if so, the input matrix is sent to the accelerator. Otherwise, output stationary tiling is used to tile and send the weight and input data to the accelerator using the ‘0x02’ and ‘0x04’ opcodes, respectively. Once the data blocks are sent to the accelerator, the ‘0x08’ opcode is sent to the accelerator to start executing the MatMul operation; iii) Finally, once the accelerator processes all the data, the ‘0x10’ opcode is sent to the accelerator to transfer the output data to main memory.

IV. EVALUATION

To evaluate our F-BFQ accelerator design and understand the performance improvements, we fully utilize the integration with SECDA-LLM [17] to perform end-to-end LLM inference.

A. Experimental Setup

We deploy and evaluate our accelerator design on the AMD KV260 [29] board. Table II provides some hardware details along with the resource utilization of our accelerator design (in %). We deploy three LLMs to evaluate our accelerator, and Table III provides key details about them, such as the number of MatMul layers, number of parameters, and model size. These LLMs contain various levels of BFP quantization but always contain layers quantized to Q_3_K and Q_2_K . We use the *llama-cli* program from the *llama.cpp* framework to cross-compile for our targeted CPU architecture (ARMv8) with NEON vector instructions enabled alongside our accelerator driver. Note that we run all our experiments 10 times each and take the average to present the results.

B. Results

We evaluated the accelerator performance in terms of inference time versus the CPU baseline on the KV260 board. Table IV summarizes the results of these experiments. The

TABLE III: Specification of the LLMs used. Note the number of Q_2_K and Q_3_K MatMul layers within each model.

Model	Q_2_K Layers	Q_3_K Layers	Parameters	Size (MBs)
GPT2 [2]	25	24	163M	77
TinyLlama [16]	45	110	1.1B	460
MobileLLaMA [15]	49	120	1.4B	560

TABLE IV: Results for LLM inference using our F-BFQ accelerator design in terms of overall execution time (seconds), speedup compared to the CPU baseline, and tokens per second across the three LLMs under study.

Model	Hardware	Overall (s)	Speedup	token/s
GPT2	CPU	1.85	1.00	8.31
	FBFQ	1.58	1.17	12.18
MobileLLaMA	CPU	21.78	1.00	0.69
	FBFQ	14.40	1.51	1.44
TinyLlama	CPU	17.59	1.00	0.86
	FBFQ	11.49	1.53	1.82

total time and speedup relative to the CPU available are presented in the ‘Overall’ column. Note that our input prompt passed to the LLMs contained 6 tokens and that we requested each model to generate 10 tokens during our experiments. The ‘Overall’ time represents the total time for both prompt processing and token generation; similarly, we consider both aspects for the ‘token/s’ columns.

Overall, we achieve a speedup of $1.4\times$ on average across the three LLMs, reaching up to 12.2 tokens/s for the GPT2 model. We see that the GPT2 model, which is the smallest of the three, results in the lowest performance gains. After further model analysis, which involved calculating the number of operations required for each layer, we conclude that this is due to the lower computational intensity of the model, especially during the token generation phase, which leads to high data bandwidth requirements. We plan to support more quantization variants, such Q_4_K - Q_8_K , within F-BFQ and enable consecutive MatMul layers to be processed within the accelerator before sending the output data back to main memory to reduce the data transfer bandwidth bottleneck.

V. CONCLUSION

We proposed a new flexible block floating point (BFP) quantization accelerator, F-BFQ, for matrix multiplication operations within LLMs. F-BFQ supports two BFP quantization variants by switching operational modes via opcodes generated by the F-BFQ driver. We designed and implemented the accelerator architecture within the SECDA-LLM platform [17] and evaluated it across three LLMs. Compared to an Arm NEON-optimized CPU baseline, we obtained an average speedup of $1.4\times$, while averaging 5.2 tokens per second. Future work will support other BFP variants to accelerate all MatMul operations within a given BFP-quantized LLM.

ACKNOWLEDGMENT

This work was supported by the EU Project dAIEDGE (GA Nr 101120726) and the Innovate UK Horizon Europe Guarantee (GA Nr 10090788).

REFERENCES

- [1] H. Touvron *et al.*, “Llama: Open and efficient foundation language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2302.13971>
- [2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [3] T. B. Brown *et al.* Language Models are Few-Shot Learners. [Online]. Available: <http://arxiv.org/abs/2005.14165>
- [4] P. P. Ray, “ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope,” vol. 3, pp. 121–154. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S266734522300024X>
- [5] X. Sun *et al.* Text Classification via Large Language Models. [Online]. Available: <http://arxiv.org/abs/2305.08377>
- [6] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation,” vol. 36, pp. 21558–21572. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash_43e9d647cccd3e4b7b5baab53f0368686-Abstract-Conference.html
- [7] B. Yao, M. Jiang, D. Yang, and J. Hu, Benchmarking LLM-based Machine Translation on Cultural Awareness. [Online]. Available: <http://arxiv.org/abs/2305.14328>
- [8] J. Clusmann *et al.*, “The future landscape of large language models in medicine,” vol. 3, no. 1, pp. 1–8. [Online]. Available: <https://www.nature.com/articles/s43856-023-00370-1>
- [9] Google, “What gemini is and how it works,” <https://gemini.google/overview-gemini-app.pdf>, 2024, accessed: 2025-10-14.
- [10] “GGML,” <https://github.com/ggerganov/ggml>, 2024.
- [11] “llama.cpp,” <https://github.com/ggerganov/llama.cpp>, 2024.
- [12] 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 (FPGA ’21)*. New York, NY, USA: Association for Computing Machinery, Feb. 2021, p. 227.
- [13] 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)*. Las Vegas, NV, USA: IEEE, Sep. 2020, pp. 84–89.
- [14] P. Gibson, J. Cano, E. Crowley, A. Storkey, and M. O’boyle, “DLAS: A Conceptual Model for Across-Stack Deep Learning Acceleration,” vol. 22, no. 1, pp. 1:1–1:28. [Online]. Available: <https://dl.acm.org/doi/10.1145/3688609>
- [15] X. Chu *et al.*, “Mobilevlm : A fast, strong and open vision language assistant for mobile devices,” 2023. [Online]. Available: <https://arxiv.org/abs/2312.16886>
- [16] P. Zhang, G. Zeng, T. Wang, and W. Lu, TinyLlama: An Open-Source Small Language Model. [Online]. Available: <http://arxiv.org/abs/2401.02385>
- [17] J. Haris, R. Saha, W. Hu, and J. Cano, “Designing Efficient LLM Accelerators for Edge Devices,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.00462>
- [18] A. Vaswani *et al.*, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [19] H. Touvron *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.
- [20] W. Kwon *et al.*, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [21] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, “Q8BERT: Quantized 8Bit BERT,” in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. IEEE Computer Society, Dec. 2019, pp. 36–39.
- [22] L. J. Wan and et al., “Software/Hardware Co-design for LLM and Its Application for Design Verification,” in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Incheon, Korea, Republic of: IEEE, Jan. 2024, pp. 435–441.
- [23] X. Shen *et al.*, “EdgeQAT: Entropy and Distribution Guided Quantization-Aware Training for the Acceleration of Lightweight LLMs on the Edge,” <http://arxiv.org/abs/2402.10787>, 2024, accessed: Apr. 18, 2024.
- [24] B. D. Rouhani *et al.*, “Microscaling Data Formats for Deep Learning,” <https://arxiv.org/abs/2310.10537>, 2023, accessed: October 19, 2023.
- [25] Simon Willison, “Alpaca: A New Programming Language for Data Analysis,” <https://simonwillison.net/2023/Mar/13/alpaca/>, 2023, accessed: March 13, 2023.
- [26] G. Gerganov, “GGUF Quantizations,” <https://huggingface.co/docs/hub/en/gguf#quantization-types>.
- [27] S.-H. Noh, J. Koo, S. Lee, J. Park, and J. Kung, “FlexBlock: A Flexible DNN Training Accelerator With Multi-Mode Block Floating Point Support,” vol. 72, no. 9, pp. 2522–2535. [Online]. Available: <https://ieeexplore.ieee.org/document/10059122>
- [28] Y.-C. Lo and R.-S. Liu, “Bucket Getter: A Bucket-based Processing Engine for Low-bit Block Floating Point (BFP) DNNs,” in *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1002–1015. [Online]. Available: <https://dl.acm.org/doi/10.1145/3613424.3614249>
- [29] AMD. Kria KV260. AMD. [Online]. Available: <https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html>