# $F^3$: An FPGA-Based Transformer Fine-Tuning Accelerator With Flexible Floating Point Format

Zerong He, Xi Jin, and Zhongguang Xu

***Abstract*—Transformers have demonstrated remarkable success across various deep learning tasks. However, their inference and fine-tuning require substantial computation and memory resources, posing challenges for existing hardware platforms, particularly resource-constrained edge devices. To address these limitations, we propose $F^3$, an FPGA-based accelerator for transformer fine-tuning. To reduce computation and memory overhead, this paper proposes a flexible floating point (FFP) format which consumes fewer resources than traditional floating-point formats of the same bitwidth. We adapt low-rank adaptation to FFP format and propose a fine-tuning strategy named LR-FFP which reduces the number of trainable parameters without compromising fine-tuning accuracy. At the hardware level, we design specialized processing elements (PEs) for the FFP format. The PE maximizes the utilization of DSP resources, enabling a single DSP to perform two multiply-accumulate operations per cycle. The PEs are organized into a systolic array (SA) to efficiently handle general matrix multiplication during fine-tuning. Through theoretical analysis and experimental evaluation, we determine the optimal dataflow and SA parameters to balance performance and resource consumption. We implement the architecture on the Xilinx VCU128 FPGA platform and $F^3$ achieves a performance of 8.2 TFlops at 250 MHz. Compared with CPU and GPU implementations, $F^3$ achieves speedups of 15.22× and 3.44×, respectively, and energy efficiency improvements of 70.52× and 9.44×.**

***Index Terms*—Transformer, fine-tuning, low-precision floating point, field-programmable gate array (FPGA), hardware accelerator.**

## I. Introduction

TRANSFORMER models [1] have emerged as a transformative paradigm in artificial intelligence, achieving state-of-the-art accuracy in tasks such as language understanding [2], [3], image processing [4], [5], [6], and generative modeling [7], [8]. Due to the substantial computation and memory requirements of training transformers from scratch, fine-tuning pre-trained models for specific downstream tasks has become the more practical and widely adopted approach. Fine-tuning transformer models demands personal data, and the computation and memory access overhead is much greater than that of the inference [9]. One common approach is to upload personal data to servers for fine-tuning, with the updated model subsequently returned to the user. However, this method poses serious risks to user privacy and is impractical in offline scenarios. Thus, fine-tuning the model directly on edge devices becomes an attractive option. Despite its appeal, fine-tuning on edge devices faces significant challenges due to resource and power constraints.

FPGAs have significant advantages in edge computing due to the flexible dataflow and high energy efficiency. Since the fixed-point number can achieve floating-point precision in neural network inference and fixed-point computation on an FPGA consumes fewer resources than floating-point computation, most neural network accelerators targeting FPGAs are designed for inference. Some recent work [10], [11], [12] employs low-precision fixed-point numbers to fine-tune transformers. However, during fine-tuning, they de-quantize the parameters to floating-point numbers. Because solving the problems of insufficient computing power and accuracy degradation simultaneously is difficult, few works have been done to enable transformer fine-tuning on FPGA platforms.

Floating-point numbers are essential to maintaining accuracy and ensuring the stability of the fine-tuning process. The challenge lies in bridging the gap between the limited floating-point computing power of FPGAs and the high computing power requirements of transformer fine-tuning. This paper reduces the computation and memory overhead for floating-point fine-tuning through two key approaches. First, we employ a parameter-efficient fine-tuning technique named low-rank adaptation (LoRA) [13] to minimize the number of trainable parameters. Second, we introduce a hardware-friendly floating-point format to enhance the floating-point computing power of FPGAs.

Numerous works have attempted to customize floating-point formats and operator algorithms to reduce hardware resource overhead. Brain Floating Point half-precision format (BF16) [14] and 8-bit floating point (FP8) [15], [16] have been proposed for deep neural network (DNN) training and applied on NVIDIA high-end GPUs. However, the resource consumption of the BF16 and FP8 operations on the FPGA is high. A key contributor to this overhead is aligning the exponent values of operands during floating-point operations. After exponent alignment, the bitwidth of operands increases significantly, and thus the resource consumption increases as well. For instance, an FP8 multiplier consumes at least 1 DSP

Received 4 December 2024; revised 13 February 2025; accepted 24 March 2025. Date of publication 31 March 2025; date of current version 16 June 2025. This work was supported by the Hefei Science of China Microelectronics Innovation Center. This article was recommended by Guest Editor J. Park. *(Corresponding author: Zhongguang Xu.)*

Zerong He and Zhongguang Xu are with the School of Microelectronics, University of Science and Technology of China, Hefei, Anhui 230026, China (e-mail: hezerong@mail.ustc.edu.cn; xuxu@ustc.edu.cn).

Xi Jin is with the School of Physical Science, University of Science and Technology of China, Hefei, Anhui 230022, China (e-mail: jinxi@ustc.edu.cn).

Digital Object Identifier 10.1109/JETCAS.2025.3555970

and 100 look-up tables (LUTs) on the Xilinx FPGA. To reduce the resource consumption of the processing element (PE) on the FPGA, Block Floating-Point (BFP) format is proposed. BFP compresses multiple floating-point numbers by aligning their exponential values to one number and truncating mantissa. Lian et al. [17] and Zhao et al. [18] employ BFP on their FPGA-based DNN accelerators. However, as we can see in Section III-B, this data format is not suitable for model training due to the transpose operation. This paper proposes a flexible floating point (FFP) format for transformer fine-tuning. FFP not only has the advantages of BFP, but is also applicable to model training.

We adapt LoRA to the FFP format and propose an FPGA-friendly fine-tuning strategy called LR-FFP. We observe that operations with a high impact on accuracy, such as softmax and low-rank matrix multiplication (LRMM), have relatively low time complexity. Therefore, high-precision format BF16 is adopted for these operations. General matrix multiplication (GEMM) dominates the latency in transformer fine-tuning and performing GEMM in FFP format can achieve significant performance gains. FFP format increases FPGAs' floating-point computing power by 8× and LoRA reduces the number of trainable parameters by two orders of magnitude. Based on FFP format, we design a resource-efficient PE and these PEs are organized into systolic arrays (SA) to perform GEMM. Due to the specificity of the FFP format and PE, our SAs have different characteristics from conventional ones. We conduct a theoretical analysis to evaluate the performance of SAs at different sizes and dataflow. By comparing with existing low-precision floating-point PEs, we demonstrate the advantages of our scheme in terms of resource consumption and performance. The low-rank weight matrices introduced by LoRA have very small dimensions, which makes the SA quite inefficient. To solve this problem, we design two specialized types of SAs, LR-SA and GEMM-SA. LR-SA is used for low-rank matrices and GEMM-SA for large-scale matrices. In summary, the contributions of this paper are as follows:

1) We introduce a novel floating point format FFP and propose an FPGA-friendly fine-tuning strategy for FPGA-oriented transformer fine-tuning. We test different transformer models and neural language processing tasks. Experiments show that when the number of trainable parameters is reduced by two orders of magnitude, the accuracy degradation is still within 1%. Our strategy makes it possible to bridge the computing power gap between the FPGA and transformer fine-tuning.
2) We design a PE that enables one DSP to perform two 8-bit FFP (FFP8) multiply-accumulate (MAC) operations in one cycle. Unlike other double MAC schemes, our approach not only supports signed-number MAC, but also enables cascading of an arbitrary number of DSPs. We organize PEs into an SA and conduct a theoretical analysis to determine the optimal configuration and dataflow for transformer fine-tuning.
3) We propose F$^3$, an FPGA-based transformer fine-tuning accelerator. F$^3$ is equipped with two specialized SAs: LR-SA for LRMM and GEMM-SA for GEMM. LR-SA is designed with high precision and flexibility but a smaller array size, while GEMM-SA features a larger array size with lower precision and flexibility. This design can save hardware resources without impacting performance and accuracy.
4) We implement F$^3$ on the Xilinx VCU128 FPGA platform and evaluate the performance and energy efficiency. F$^3$ achieves a performance of 8.2 TFlops at 250 MHz. Compared with CPU and GPU implementations, F$^3$ achieves speedups of 15.22× and 3.44× in performance, and 70.52× and 9.44× in energy efficiency, respectively.

## II. Background and Related Work

### A. Fine-Tuning Transformers With LoRA

Existing parameter-efficient fine-tuning schemes include adapter tuning [19], prefix tuning [20], prompt tuning [21], and LoRA [13]. Adapter tuning inserts trainable layers into the original transformer model, and only the inserted layers are updated during fine-tuning. Prefix tuning and prompt tuning incorporate a set of virtual token embeddings to the input token embeddings of certain transformer layers, and only the virtual token embeddings are updated. These techniques reduce computation and memory overhead by limiting the number of trainable weights, but they increase inference overhead. On the contrary, LoRA reduces the fine-tuning overhead without making any changes to inference. Therefore, we employ LoRA in our fine-tuning scheme.

For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA constrains its update by representing the latter with a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll min(d, k)$. During training, $W_0$ is frozen, while $A$ and $B$ are trainable parameters.

A complete training process consists of three phases: forward pass (FP), backward pass (BP) and weight update (WU). Given an input matrix $X$ and a pre-trained weight matrix $W_0$, the FP phase can be expressed in Equation 1.

$$Y = X \times (W_0 + B \times A) \tag{1}$$

In the BP phase, we need to calculate the error and gradient. The error and gradient are the derivatives of $X$ and weights with respect to loss $L$, respectively. Since $W_0$ is frozen, only the gradients of A and B need to be calculated. The BP phase can be expressed in the following equations:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \times (W_0 + BA)^T \tag{2}$$

$$\frac{\partial L}{\partial A} = (B^T \times X^T) \times \frac{\partial L}{\partial Y} \tag{3}$$

$$\frac{\partial L}{\partial B} = X^T \times \left(\frac{\partial L}{\partial Y} \times A^T\right) \tag{4}$$

where $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial Y}$ are errors, $\frac{\partial L}{\partial A}$ and $\frac{\partial L}{\partial B}$ are gradients. In the WU phase, we use AdamW optimizer [22] to update low-rank weight matrices $A$ and $B$. AdamW is widely adopted in transformer training due to its faster convergence and better generalization performance compared to Adam [23]. Since the weight matrices $A$ and $B$ have small dimensions, the overhead of WU is very low.

TABLE I
DETAILS OF FP8 FORMATS

| | E4M3 | E5M2 |
|---|---|---|
| **Exponent bias** | 7 | 15 |
| **Infinities** | N/A | $S.11111.00_2$ |
| **NaN** | $S.1111.111_2$ | $S.11111.\{01,10,11\}_2$ |
| **Zeros** | $S.0000.000_2$ | $S.00000.00_2$ |
| **Max normal** | $S.1111.110_2 = 1.75 \times 2^8$ | $S.11110.11_2 = 1.75 \times 2^{15}$ |
| **Min normal** | $S.0001.000_2 = 2^{-6}$ | $S.00001.00_2 = 2^{-14}$ |
| **Max subnorm** | $S.0000.111_2 = 0.875 \times 2^{-6}$ | $S.00000.11_2 = 0.75 \times 2^{-14}$ |
| **Min subnorm** | $S.0000.001_2 = 2^{-9}$ | $S.00000.01_2 = 2^{-16}$ |

### B. 8-Bit Floating Point Format

Floating point numbers contain the sign bit ($s$), exponent ($e$) and mantissa ($f$). The value of a floating point number can be expressed as $(-1)^s \cdot (1.f) \cdot 2^{e-b}$, where exponent bias $b$ depends on the exponent bitwidth $W_e$, i.e., $b = 2^{W_e-1} - 1$. NVIDIA proposes two types of FP8 format, E4M3 and E5M2 [16]. E4M3 has 4 exponent bits and 3 mantissa bits, while E5M2 has 5 exponent bits and 2 mantissa bits. The details of these formats are listed in Table I. E5M2 follows IEEE 754 conventions for representation of special values (i.e., infinities and nan) while E4M3 does not. E5M2 can represent a larger data range and E4M3 has higher precision.

### C. FPGA-Based Multiply-Accumulator

To enhance the fixed-point computing power of FPGAs, Xilinx proposes a method [24] that enables a single DSP to compute two signed 8-bit MACs in one cycle. This method packs input $a$ and $b$ into a single operand and separates the multiplication results between $a$, $b$ and input $c$ into $a \times c$ and $b \times c$. Product terms $ac$ and $bc$ are kept in the high and low bits of the result. However, with only 2 bits remaining between the lower and upper product terms, the number of product terms each DSP can accumulate is limited.

Floating-point operations on the FPGA usually consume lots of logic resources, so many works have proposed low-precision floating-point formats and corresponding hardware architectures. Xilinx introduces a 7-bit small floating point (E3M3) format [25] and implements a multiplier with just 9.5 LUTs on FPGAs. Since the result of multiplication is truncated, the precision of MAC is insufficient. Wu et al. [26] utilizes E3M4 (FP8) format to accelerate DNN inference on the FPGA. They significantly reduce the consumption of DSPs by implementing four 4-bit multiplications with one DSP. However, the accumulation of intermediate results requires a lot of exponent alignment logic and adder trees. Lian et al. [17] and Zhao et al. [18] apply BFP format to DNN inference. This format groups data into blocks and all elements in a block share the same exponent. BFP significantly reduces the number of exponent alignments. The bitwidth of mantissa in BFP is very low so that Xilinx's method mentioned above can be used to improve the computing efficiency of DSPs. However, neural network training involves matrix transposition and blocks with fixed shapes can not match the dataflow of BP.

### D. Hardware Accelerators for Transformer Models

Existing FPGA-based transformer accelerators typically target inference tasks. Zhuang et al. [27] propose a spatial sequential hybrid architecture to accelerate vision transformer inference. They leverage the AMD ACAP VCK190's heterogeneous components, including FPGA and AIE vector cores, and map matrix-multiply layers onto the AIE part. However, most relatively low-end FPGA platforms use DSPs as the primary computing resource and require DSP-targeted optimization solutions. Zeng et al. [28] propose a configurable sparse DSP chain to support different sparsity patterns in transformer-based Large Language Model (LLM) inference. Unlike transformer training, the bottleneck for LLM is limited memory bandwidth and low bandwidth utilization. Chen et al. [29] map LLMs to AMD U280 FPGA and employ a fully pipelined design. For different models and hardware platforms it is necessary to re-adjust the dataflow and allocate hardware resources so that the latency of each layer is comparable.

Transformer training requires high computing power, which makes on-device training very challenging. Shao et al. [30] apply 8-bit piecewise integer (PINT) [31] to represent and compute GEMM in transformer training. PINT can significantly reduce memory requirements and achieve the same performance for different tasks as FP32. However, each PE decodes the operands and shifts the result. As the size of the PE array increases, resource consumption grows significantly. Tuli et al. [9] propose a dynamic and unstructured pruning technique to skip ineffectual MAC operations in transformer training. Although this technique induces 50% sparsity in weights and activations and 90% sparsity in gradients, it leads to irregular memory access patterns and workload imbalance. Yu et al. [32] adapt LoRA to FP8 and analyze the accuracy of transformer inference and training. They propose an architecture to accelerate transformer fine-tuning, but they don't evaluate the performance of the whole system.

## III. FPGA-FRIENDLY FINE-TUNING STRATEGY

Compared with low-precision transformer inference, low-precision fine-tuning receives less attention. Yu et al. [32] have demonstrated that FP8 format can match the accuracy of BF16 format. However, the resource consumption of FP8 operations on the FPGA is much higher than that of INT8 operations. Therefore, we modify FP8 format so that the resource consumption is comparable to that of INT8 format.

### A. Observation on 8-Bit Floating Point Fine-Tuning

Sun et al. [15] use hybrid FP8 formats to train DNNs. They find that errors have a wider dynamic range compared with weights and activations. Therefore, weights and activations are quantized in E4M3 format, while errors are quantized in E5M2 format. We fine-tune a four-layer BERT [2](BERT-Small) and a distilled version of BERT (DistilBERT [33]) on the general language understanding evaluation (GLUE) [34] dataset and CoNLL dataset [35] using the same quantization strategy.

We observe that the dynamic range of E4M3 and E5M2 formats is underutilized during the training process. Figure 1

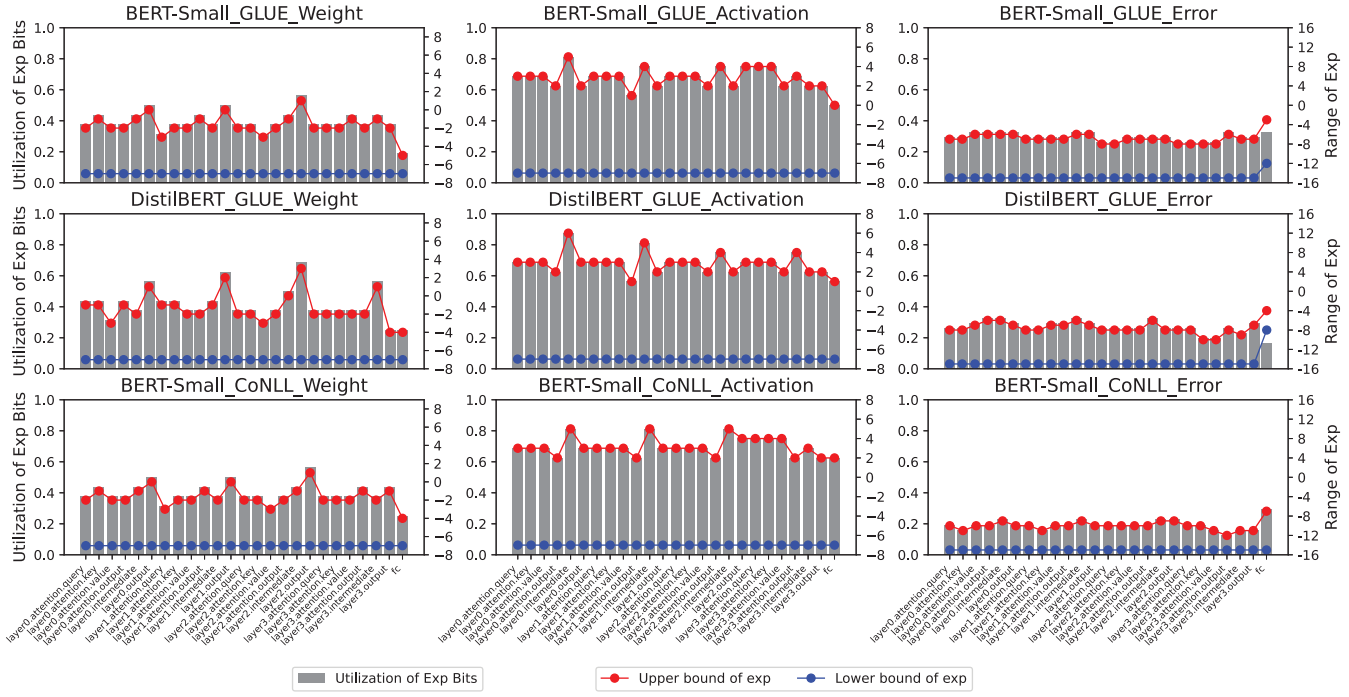



Fig. 1. The utilization of exponent bits and the range of exponents of different layers during the training process. Weights and activations are quantized in E4M3 format. Errors are quantized in E5M2 format. The exponents of subnormal values for E4M3 and E5M2 are regarded as $-7$ and $-15$, respectively. For DistilBERT, we only show the first four encoder blocks and the last two layers.

shows the exponent range of parameters in FP8 format. Except for special values, the exponent ranges of E4M3 and E5M2 are $-7 \sim 8$ and $-15 \sim 15$, respectively. Since the range of parameters doesn't match the range of FP8 format, the exponent bits are underutilized. For example, the maximum value of the weights in the first layer (i.e., *layer*0.*attention.query*) is $0B00101110$ in E4M3 format. The upper bound of the exponent can be calculated as $e - bias = 5 - 7 = -2$. The minimum value of weights is smaller than that allowed by E4M3 format, so the lower bound of exponent is the minimum exponent of E4M3 (i.e., $-7$). We define the utilization of exponent bits as the ratio of the exponent range of parameters and the exponent range of FP8 format. For instance, the utilization for weights in the first layer of BERT-Small_GLUE is $(-2-(-7)+1)/(8-(-7)+1) = 37.5\%$. As shown in Figure 1, the average utilization of exponent bits for weights and errors is quite low, at $39.4\% \sim 41.6\%$ and $19.4\% \sim 29.3\%$, respectively.

### B. Details of FFP Format

Due to the low utilization of the dynamic range of FP8 format and the high overhead of FP8 operations on the FPGA, we propose the FFP8 format. First, we divide an input matrix into blocks. The range of parameters in each block is much smaller than that in the original matrix. Next, we modify the exponent bias to better match the range of parameters. We determine the bias based on the maximum parameters of each block. This adjustment ensures that the maximum exponent of the FFP8 format aligns exactly with the maximum exponent of the parameters. For example, if the weights are quantized in E4M3 format, the maximum weights of a layer $X_{max}$ and corresponding exponent bias $b$ satisfy the inequality $1.75 \times 2^{15-b} \geq X_{max}$. Exponent bias can be calculated by $b = 15 - \lceil log_2 \frac{X_{max}}{1.75} \rceil$.

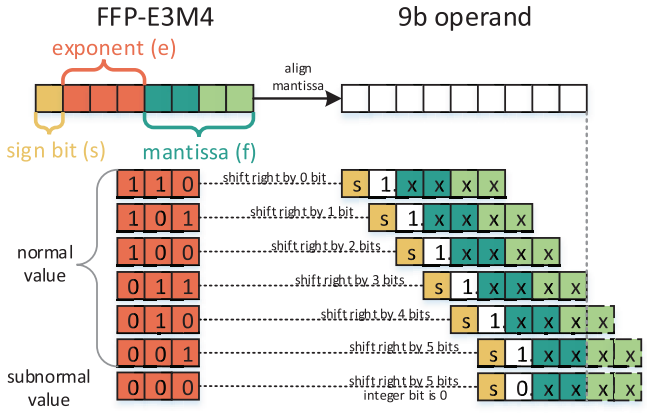



Fig. 2. Exponent alignment for FFP-E3M4 format.

To reduce the resource consumption of the FPGA, we restrict the bitwidth of the fixed-point operand after exponent alignment to 9 bits. We will explain the reason for this choice in Section IV. To restrict the fixed point operand to 9 bits, the mantissa of the corresponding floating-point number can be shifted by at most 8 bits during alignment (disregarding the sign bit), therefore the exponent bitwidth cannot exceed 3 bits. Based on the above analysis, we propose FFP-E3M4 format. Table II lists the detail of FFP-E3M4 and Figure 2 illustrates the exponent alignment for FFP-E3M4 format. The two least

TABLE II
DETAILS OF FFP-E3M4 FORMAT

| | |
|---|---|
| **Exponent bias** * | $b = 6 - \lceil log_2 \frac{X_{max}}{1.9375} \rceil$ |
| **Infinities** | $S.111.0xxx_2$ |
| **NaN** | $S.111.1xxx_2$ |
| **Zeros** | $S.000.0000_2$ |
| **Max normal** | $S.110.1111_2 = 1.9375 \times 2^{6-b}$ |
| **Min normal** | $S.001.00xx_2 = 2^{1-b}$ |
| **Max subnorm** | $S.000.11xx_2 = 0.75 \times 2^{1-b}$ |
| **Min subnorm** | $S.000.01xx_2 = 0.01 \times 2^{1-b}$ |

* $X_{max}$ is the maximum value of parameters in a block.

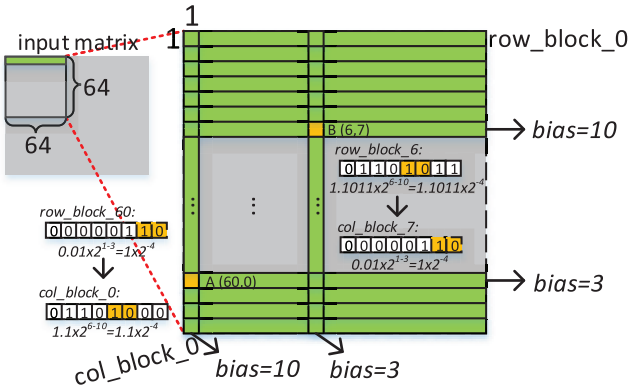



Fig. 3. The input matrix is quantized based on a row vector block and a column vector block.

significant bits of the mantissa may be rounded off due to the exponent alignment.

Fine-tuning involves matrix transposition. If the block is a vector, the method will lose its effectiveness for the following reason. Matrix multiplication $A \times B$ is done by calculating the dot product of the row vectors in $A$ and the column vectors in $B$. Even if all elements in a column vector of $B$ belong to the same block, after $B$ transposes, any column vector of $B$ must contain elements from different blocks. If the block is a square matrix, to match an SA of the same size, the size of the block will increase to the square of a vector block. The larger the block size, the lower the accuracy.

As illustrated in Figure 3, we modify the traditional vector block-based quantization method. We still use a vector block (a 64-dimensional vector in our experiment). If the input matrix is quantized and stored based on row vector blocks, it will be dynamically quantized based on column vector blocks after matrix transposition. We find experimentally that a 2-bit mantissa is sufficient to maintain fine-tuning accuracy with a block size of 64. A 4-bit mantissa can ensure that each number has at least 2 valid bits in its mantissa whether it is quantized based on a row vector block or a column vector block. Figure 3 gives an extreme case. Number A is quantized and stored based on a row vector block. It is the row vector's minimum number and the column vector's maximum number. As the minimum number, the two least significant bits of A (i.e., 10) are eliminated during exponent alignment. This information becomes important when A becomes the maximum number of the column vector block. In Figure 3, this information is

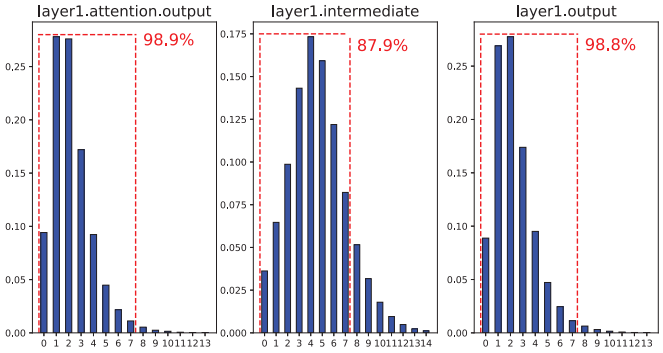



Fig. 4. Histogram of the exponent difference of errors in different layers. The x-axis represents the difference between a number's exponent and the corresponding block's maximum exponent. The y-axis is the statistical distribution of the exponent difference. The red dashed box represents the exponent range of FFP-E3M4.

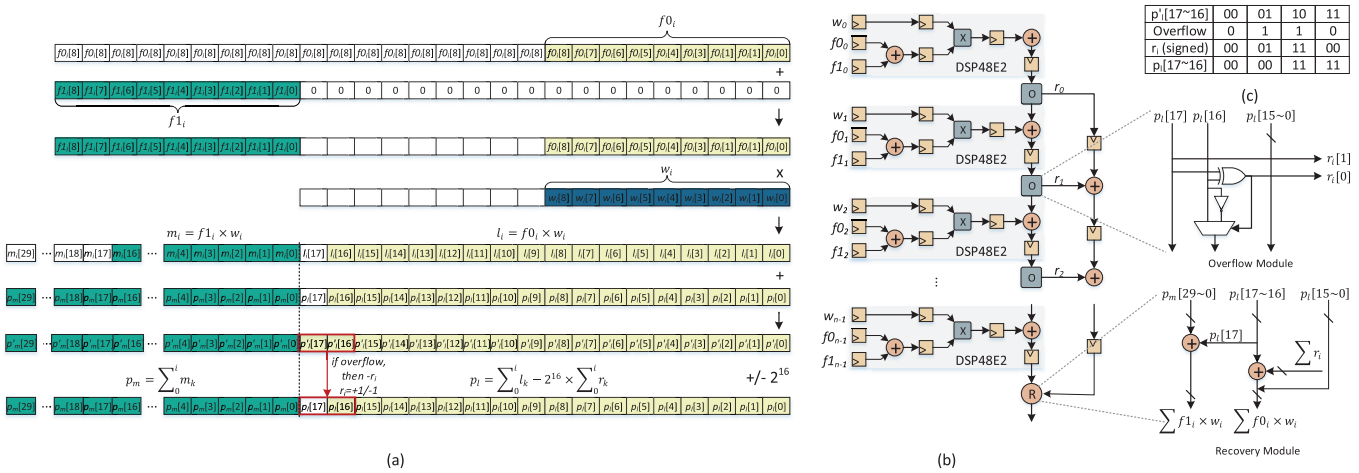
recovered when A is quantized based on the column vector block. We save the quantized matrix and the exponent bias of all row and column vector blocks.

We count the exponent difference of errors in some layers of BERT-Small. As shown in Figure 4, the exponent of most errors falls in the exponent range of FFP-E3M4 (indicated by the red dashed box). In Figure 1, we observe that the exponent range of weights and activations is smaller than that of the errors. Therefore, FFP-E3M4 format can provide sufficient precision for most of the parameters in transformer fine-tuning. There are two main reasons. First, since we quantize each vector block separately, the distribution of parameters is more centralized. Second, the dynamic range of the FFP format can be flexibly adjusted to accommodate different data distributions. The overhead of our quantization method lies in re-quantizing the intermediate results into FFP format. Our method requires recomputing the exponent bias for intermediate results, which is an extra overhead compared with fixed-point quantization. This step is handled by the Encoder in Figure 8. The main operation is finding the maximum value of a vector and the comparator is the primary module.

### C. LR-FFP Fine-Tuning Strategy

We propose an FPGA-friendly mixed-precision fine-tuning strategy LR-FFP. LR-FFP reduces hardware overhead while ensuring the model converges. Specifically, by using LoRA, LR-FFP significantly reduces the overhead of gradient computation and WU. These operations, along with nonlinear functions, have low time complexity and their precision has a significant impact on the convergence of the model. LR-FFP adopts FP32 and BF16 formats for these operations. This strategy guarantees model convergence at the cost of adding very little overhead. GEMM is the computation bottleneck of transformer fine-tuning and FFP8 format is applied.

The number of operations for BP and WU is more than that of FP during transformer training [9]. Therefore, we use LoRA to reduce this overhead. In the forward pass of LoRA (Equation 1), we calculate $W_0 + B \times A$ in BF16 and quantize the result to FFP8. The gradients of trainable matrices $A$ and $B$ (Equation 3 and 4) are also calculated using BF16. As




| $p'_i$[17~16] | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| Overflow | 0 | 1 | 1 | 0 |
| $r_i$ (signed) | 00 | 01 | 11 | 00 |
| $p_i$[17~16] | 00 | 00 | 11 | 11 |

Fig. 5. (a) How two 9-bit signed MACs work in one DSP slice. (b) Hardware architecture. (c) Truth Table of the overflow term and accumulation term.

mentioned in Section II-A, the dimensions of $A$ and $B$ are much smaller than that of the weight matrix $W_0$. LoRA reduces the time complexity of gradient computation to $4r/d$ ($r \ll d$) of the original. For example, the hidden dimension $d$ of BERT is 768 and the LoRA rank $r$ is usually set to 8. This means that the time complexity is reduced by two orders of magnitude. Even though we quantize $A$ and $B$ into BF16, the associated computation doesn't consume too much hardware resources.

We quantize activations and errors in FFP8 format and save weights in BF16 format. Intermediate results of special functions (i.e., layer normalization, softmax and activation functions) are quantized in BF16. The FP phase can be expressed as follows:

$$Q_{FFP8}(Y) = Q_{FFP8}(X) \times Q_{FFP8}(Q_{BF16}(W_0) + Q_{BF16}(B) \times Q_{BF16}(A)) \quad (5)$$

To ensure the model converges, the gradients $\frac{\partial L}{\partial A}$ and $\frac{\partial L}{\partial B}$ are represented by FP32 in WU. The BP phase is expressed as follows. Due to the small dimension of the gradients, this strategy doesn't add much overhead to WU.

$$Q_{FFP8}\left(\frac{\partial L}{\partial X}\right) = Q_{FFP8}\left(\frac{\partial L}{\partial Y}\right) \times Q_{FFP8}(W_0 + BA)^T \quad (6)$$

$$Q_{BF16}\left(\frac{\partial L}{\partial A}\right) = (Q_{BF16}(B^T) \times Q_{FFP8}(X^T)) \times Q_{FFP8}\left(\frac{\partial L}{\partial Y}\right) \quad (7)$$

$$Q_{BF16}\left(\frac{\partial L}{\partial B}\right) = Q_{FFP8}(X^T) \times \left(Q_{FFP8}\left(\frac{\partial L}{\partial Y}\right) \times Q_{BF16}(A^T)\right) \quad (8)$$

In Table III, we compare LR-FFP with recent work on transformer fine-tuning. The motivation of QLoRA [10], QA-LoRA [11] and LR-QAT [12] is to reduce memory consumption for LLM fine-tuning. They combine LoRA with quantization-aware training. The frozen pre-trained weight $W_0$ is quantized and saved in low-precision fixed-point format. Compressing $W_0$ can significantly reduce memory overhead and make it possible to train large models on a single GPU. However, $W_0$ is de-quantized to floating-point numbers (BF16) during fine-tuning. De-quantization operation, FP and BP involve a lot of high-precision floating-point computation which is unfriendly to edge devices. Yu et al. [32] quantize all the parameters to FP8 format, but the MAC for FP8 numbers still consumes a lot of resources. In LR-FFP, the parameters involved in the computation are quantized to FFP8 format. With our dedicated PEs, we can significantly alleviate the computation bottleneck for transformer fine-tuning on edge devices.

TABLE III
COMPARISON OF DIFFERENT FINE-TUNING APPROACHES

| | Fine-tuning | | | | Inference |
|---|---|---|---|---|---|
| | $W_0$ | $A/B$ | FP * | BP * | $W_0 + AB$ |
| **QLoRA [10]** | NF4 | BF16 | BF16 | BF16 | FP16 † |
| **QA-LoRA [11]** | INT4 | BF16 | BF16 | BF16 | INT4 |
| **LR-QAT [12]** | INT8 | BF16 | BF16 | BF16 | INT8 |
| [32] | FP8 | FP8 | FP8 | FP8 | FP8 |
| **LR-FFP** | BF16 | BF16 | FFP8 | FFP8 | FFP8 |

* FP: forward pass; BP: backward pass.
† For inference, the FP16 parameters are quantized to INT4 parameters by post-training quantization.

## IV. SYSTOLIC ARRAY DESIGN

In this section, we design two types of SAs using DSPs on the FPGA, GEMM-SA and LR-SA. We first introduce the architecture of PEs in the GEMM-SA and then determine the optimal configuration of GEMM-SA. Finally, we present details of LR-SA.

### A. Processing Element for FFP8 Format

The FFP8 format limits the bitwidth of operands after exponent alignment. Inputs and weights in FFP8 format are pre-aligned to the maximum exponents of each block. Therefore, the alignment logic is not needed in the PE and we can use the fixed-point PE to perform MAC. Let's first consider the bitwidth of weights and activations (FFP-E3M4) after exponent alignment. The DSP48E2 slice in Xilinx FPGA has an $18 \times 27$-bit multiplier and a 48-bit accumulator. Let $n$ represent the width of the operand after exponent alignment. To enable a single DSP48E2 to perform two signed multiplications

(i.e., $a \times c$ and $b \times c$) simultaneously, two conditions must be met [36]. First, the two operands $a$ and $b$ must be separated by at least $n$ bits. Second, the output register must be at least $4n$ bits. We pack $a$ and $b$ into a 27-bit operand and the output register is 48 bits, so $n$ cannot exceed 9. To minimize the accuracy degradation, we set $n$ of FFP-E3M4 to 9.

In Figure 5, we pack two 9-bit signed MACs in a single DSP48E2 slice. Before being packed by the pre-adder in the DSP (Figure 5), $f0_i$ and $f1_i$ are extended by sign extension and zero padding, respectively. The range of operands is limited to $[-255, 255]$, so there is no overflow in the pre-adder. The product terms $m_i$ and $l_i$ are 17 bits. With only 1-bit remaining between the lower and upper product terms, accumulating multiple product terms may result in the overflow. To prevent this, we restrict the lower accumulation term $p_l$ to 17 bits and reserve the most significant bit $p_l[17]$ to detect overflow and recovery of correct results. As illustrated in Figure 5 (a), an overflow occurs when the two most significant bits of the current accumulation value (i.e., $p'_l[17]$ and $p'_l[16]$) are different. If an overflow is detected, we add or subtract $2^{16}$ to the accumulation term to restrict it to 17 bits. This operation can be implemented by subtracting an overflow term $r_i$ ($r_i = \pm 1$) to $p'_l[17]p'_l[16]$. When there is no overflow, $r_i = 0$. Overflow terms are accumulated. The result in the accumulation register of the DSP can be expressed as follows:

$$\begin{aligned} P_R &= \sum_{i=0}^{n-1}(f0_i + f1_i \times 2^{18}) \times w_i - \sum_{i=0}^{n-2} r_i \times 2^{16} \\ &= \left(\sum_{i=0}^{n-1} f0_i \times w_i - 2^{16}\sum_{i=0}^{n-2} r_i\right) + 2^{18}\left(\sum_{i=0}^{n-1} f1_i \times w_i\right) \\ &= p_l + 2^{18} \times p_m \end{aligned} \tag{9}$$

where $p_l$ and $p_m$ are the lower and upper accumulation terms in the register (i.e., $P_R[17:0]$ and $P_R[47:18]$). The last overflow term is retained in $p_l$. We recover the correct accumulation terms as follows:

$$\begin{aligned} \sum_{i=0}^{n-1} f0_i \times w_i &= p_l + 2^{16}\sum_{i=0}^{n-2} r_i \\ &= \left(p_l[17:16] + \sum_{i=0}^{n-2} r_i\right) \| (p_l[15:0]) \end{aligned} \tag{10}$$

$$\sum_{i=0}^{n-1} f1_i \times w_i = \frac{P_R - p_l}{2^{18}} = P_R \gg 18 + sign(p_l) \tag{11}$$

where the overflow term is aligned with the two most significant bits of $p_l$. According to [18], $-p_l/2^{18}$ can be converted to the sign of $p_l$.

The hardware architecture is shown in Figure 5 (b). PEs are organized into a cascade chain which works in weight stationary (WS) dataflow. Each PE performs two MACs and passes the intermediate results to the next PE in the chain. The final result is output at the end of the cascade chain. The Overflow Module can restrict the bitwidth of $p_l$ and prevent overflow. It consists of two logic gates and one multiplexer, and the truth table is shown in Figure 5 (c). The Recovery Module implements Equation 10 and Equation 11 using a

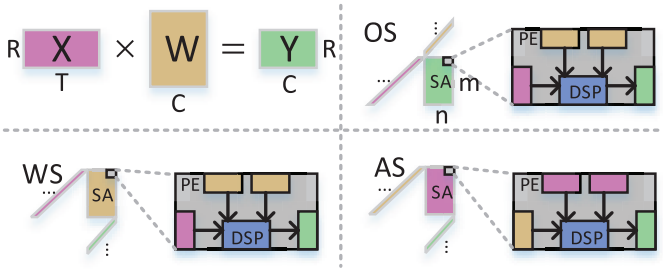



Fig. 6. Different SA dataflow for matrix multiplication.

($\lceil log_2(n+1) \rceil + 1$)-bit adder and a counter, respecitvely. When working in WS dataflow, the cascade chain requires n-1 Overflow Modules, n-2 adders and 1 Recovery Module. The advantage of WS dataflow over output stationary (OS) dataflow is that we can adjust the adder bitwidth based on the bitwidth of intermediate results. Specifically, the adder bitwidth in WS dataflow is $(\lceil log_2(i+1) \rceil + 1), 2 \le i \le n-1$ which is less than $(\lceil log_2(n+1) \rceil + 1)$ in OS stationary.

Compared with Xilinx [24] and Zhao et al. [18], our architecture has higher scalability and enables SAs with arbitrary size by modifying the width of the adder for overflow terms. Xilinx [24] supports the accumulation of up to 7 product terms. After 7 product terms, an additional DSP slice is required to extend this limitation. Zhao et al. [18] support the accumulation of 16 product terms. To scale up the SA, they propose a 3D MAC array. However, this structure can make layout difficult. Compared with Wu et al. [26] and Lee et al. [36], we implement the signed MAC by simply adding a counter. Wu et al. [26] and Lee et al. [36] only support unsigned or signed-unsigned multiplication.

### B. Design Space Exploration for GEMM-SA

FP (Equation 5) and error computing (Equation 6) are performance bottleneck. We use GEMM-SAs to compute the large-scale GEMM in these two phases. Given the specific characteristics of transformer fine-tuning, it is crucial to determine an optimal GEMM-SA configuration, including its size and dataflow.

As illustrated in Figure 6, GEMM can be expressed as $Y^{R\times C} = X^{R\times T} \times W^{T\times C}$. Suppose we use OS dataflow and the GEMM-SA size is $m \times n \quad (m \ge n, mn = S)$. $m$ and $n$ are divisible by $R$ and $C$. The latency can be expressed as follows:

$$\begin{aligned} L &= \lceil \frac{R}{m} \rceil \lceil \frac{C}{2n} \rceil (m + 2n + T - 2) \\ &= \frac{RC}{2S}\left(\frac{2S}{m} + m + T - 2\right) \end{aligned} \tag{12}$$

Due to our double MAC scheme, the output matrix of SA is $m \times 2n$. The latency can be minimized only when $m = \sqrt{2S}, n = \sqrt{2S}/2$. Therefore, the most appropriate size for GEMM-SA is $2n \times n$ (in our experiment, $n$ is 32).

We discuss the performance of GEMM-SA under three different dataflow, OS, WS and activation stationary (AS). In Figure 6, each DSP receives two weights and an activation (or an error) per cycle when the GEMM-SA works in OS or WS dataflow. When the GEMM-SA works in AS dataflow, a

TABLE IV
GEMM-SA LATENCY WHEN WORKING IN THREE DIFFERENT DATAFLOW

| Condition | T=C | T=C/4 | T=4C |
|---|---|---|---|
| | $L_{OS} = L_{AS}$ | $L_{OS} > L_{AS}$ | $L_{OS} < L_{AS}$ |
| $R \geq max(T, C)$ | $L_{WS} \leq L_{AS}$ | $L_{WS} \leq L_{AS}$ | $L_{WS} \leq L_{OS}$ |
| $R < max(T, C)$ | $L_{WS} > L_{AS}$ | $L_{WS} > L_{AS}$ | $L_{WS} > L_{OS}$ |

DSP receives two activations (or two errors) and a weight per cycle. Suppose that the dimension of GEMM-SA is divisible by that of the input matrix. The GEMM-SA latency in the three dataflow can be expressed as follows:

$$L_{OS} = \frac{RC}{4n^2}(4n + T - 2) \tag{13}$$

$$L_{WS} = \frac{TC}{4n^2}(4n + R - 2) \tag{14}$$

$$L_{AS} = \frac{TR}{4n^2}(4n + C - 2) \tag{15}$$

The transformer model consists of multiple Encoder Blocks with the same structure. Each Encoder Block has a Multi-Head Attention (MHA) and a Feed Forward Network (FFN) which consists of two Multilayer Perceptrons (MLPs). Many transformer models have the following patterns: (1) The weight matrices in MHA are square matrices. (2) The weight matrices of the two MLPs in FFN are $d \times 4d$ and $4d \times d$. Based on these patterns, we summarize the three cases about $T$ and $C$ in Table IV. The three cases are valid for both FP and error computing. WS dataflow and AS dataflow are based on the same GEMM-SA architecture. They differ only in the propagation direction of the weight and activation matrices. OS dataflow achieves the best performance only if $T = 4C$ and $R < 4C$. This is not common in transformer fine-tuning, therefore, we choose WS and AS dataflow for GEMM-SAs. For example, only one MLP per Encoder Block in BERT satisfies $T = 4C$ ($T = 3072$ and $C = 768$). In transformer fine-tuning, one batch consisting of multiple samples is processed per iteration. Suppose the sequence length is 128. OS dataflow has a performance advantage in a few MLP layers only if the batch size is less than 24.

### C. High-Precision and Flexible LR-SA

LoRA fine-tuning introduces LRMMs to FP and BP (i.e., Equation 7- 8 and $B \times A$ in Equation 5). Because the low-rank matrices have small dimensions and require high precision, we design an LR-SA specifically for them. Each PE in the LR-SA can perform one BF16 MAC per cycle. In Equation 7- 8, the parameters in FFP8 format are de-quantized to BF16 format before being sent to the PE.

We express the LRMM as $Y^{R\times C} = X^{R\times T} \times W^{T\times C}$. The LRMM in transformer fine-tuning can be categorized into two cases: $R = r$ *or* $C = r$, and $T = r$, where LoRA rank $r$ is typically set to 8. To improve PE utilization, we set the shape of the LR-SA to $8 \times 64$. Since LoRA rank $r$ is much smaller than the hidden dimension, the latency of the SA varies significantly depending on the dataflow. For example, the hidden dimension of BERT is 768. The latency of $B^{768\times 8} \times A^{8\times 768}$ under WS and OS dataflow is approximately 10100 cycles and 99000 cycles, respectively. Therefore, we design an LR-SA that can work in both OS and WS dataflow (Figure 7). If $T = r$, the LR-SA works in WS dataflow. Otherwise, it operates in OS dataflow. This flexibility increases the consumption of registers and multiplexers. Due to the small size of the LR-SA, the additional resource consumption is not significant.

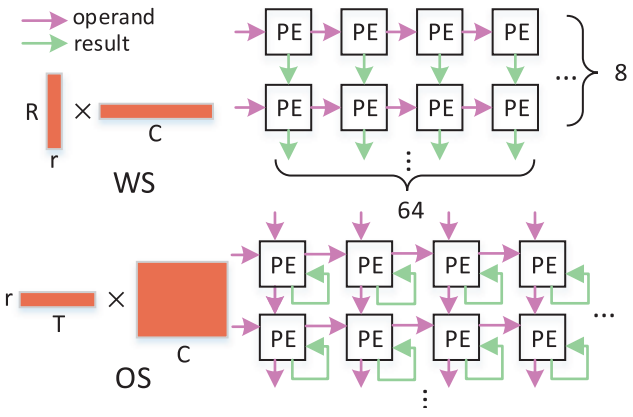



Fig. 7. Two operation modes of LR-SA.

## V. F$^3$ HARDWARE ARCHITECTURE

This section describes the architecture of F$^3$. We first introduce the architecture overview and then detail some core modules. Finally, we present the dataflow of F$^3$.

### A. Architecture Overview

Figure 8 illustrates the overall architecture of F$^3$. F$^3$ consists of off-chip memory, Memory Controller, Main Controller, on-chip buffer and processing modules. Processing modules include GEMM-SA, Adder Array (AA), Vector Processing Unit (VPU), LR-SA and Weight Update Module (WUM). GEMM-SA performs large-scale matrix multiplication and AA accumulates the output of GEMM-SA. To minimize precision loss, we choose a 32-bit adder for AA. VPU performs complex operations in FP and BP. WUM updates weights in FP32 format. We choose High Bandwidth Memory (HBM) as off-chip memory. HBM provides sufficient bandwidth and multiple access ports, and thus different processing modules can access data from different addresses of HBM simultaneously. Main Controller informs each processing module to execute its respective tasks. Data access is also controlled by Main Controller which sends control signals and addresses to Memory Controller and on-chip buffer.

### B. Vector Processing Unit

Complex functions in transformer fine-tuning, such as softmax, layer normalization, activation functions (e.g., GELU [37] in BERT) and their derivatives, require specialized hardware support. To efficiently handle these functions, we design a flexible VPU with a dynamically adaptable dataflow for different complex operations. Figure 9 shows the structure of the ALU in VPU. The ALU processes data in BF16 format and each ALU contains a multiplier and an adder. Since dividers are resource-intensive, not all ALUs are equipped with one.

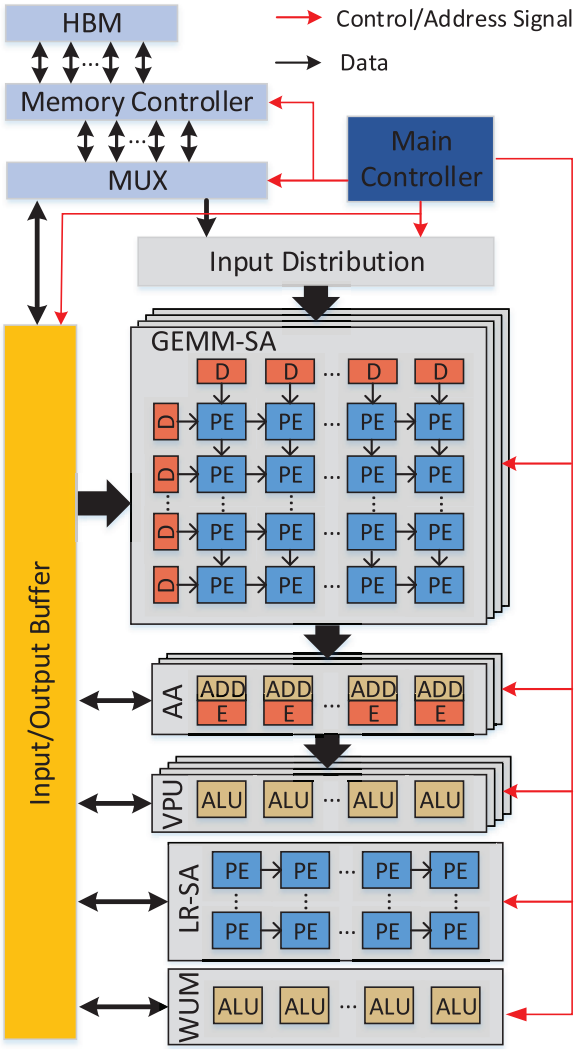



Fig. 8. Overall Architecture of F$^3$. D and E represent Decoder and Encoder, respectively.

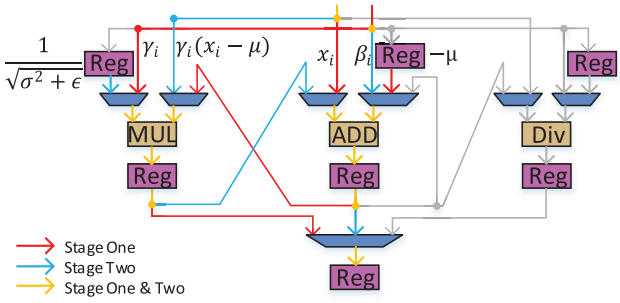



Fig. 9. ALU in Vector Processing Unit. The different colored arrows indicate the dataflow at different stages during the computation of layer normalization.

In addition, a few ALUs contain special modules designed for logarithmic function and square root function. To further save logic resources, the exponential function is implemented with lookup tables and multipliers.

The ALU achieves high hardware utilization. In the best case, the adder, multiplier and divider in the same ALU can work simultaneously. In addition, the ALU has only two input

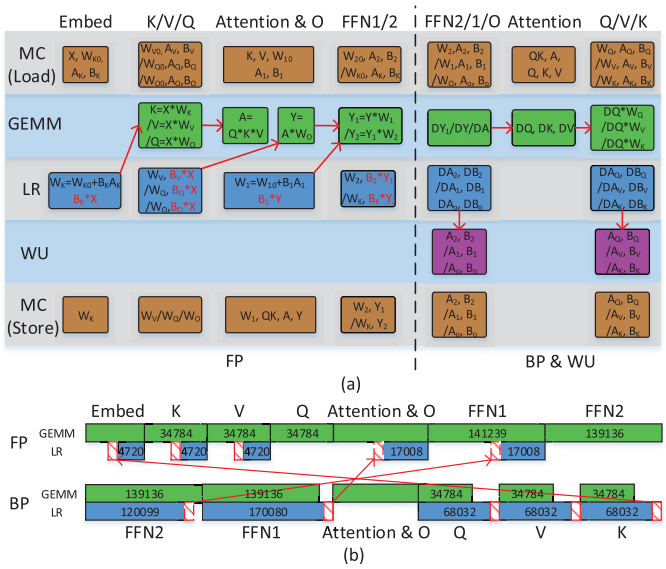



Fig. 10. (a) The dataflow of LR-FFP fine-tuning. Layer normalization is not included. Red arrows indicate on-chip data reuse. (b) Workload balance. We label the latency of GEMM-SA and LR-SA. The unit is the cycle. It is measured on BERT-small.

ports and the on-chip bandwidth requirement is not high. This low bandwidth demand is due to two factors. One is that we improve the data reuse within the ALU. The other is that some operands in complex functions are constants. We give an example in Figure 9. One ALU can complete the computation of layer normalization $\mathbf{Y} = \boldsymbol{\gamma} \odot \frac{\mathbf{X}-\mu}{\sqrt{\sigma^2+\epsilon}} + \boldsymbol{\beta}$ in two stages. Specifically, the ALU completes $\gamma_i \times (x_i - \mu)$ and $\frac{[\gamma_i \times (x_i-\mu)]}{\sqrt{\sigma^2+\epsilon}} + \beta_i$ in Stage One and Two, respectively.

### C. The Dataflow Optimization for Fine-Tuning

LR-FFP fine-tuning strategy compresses the model and reduces the overhead of weight updates. However, this strategy introduces new operators (e.g., LRMM) and makes the dataflow of model training more complex. To enhance on-chip data reuse and achieve workload balance, we optimize the dataflow for LR-FFP fine-tuning.

In Figure 10 (a), we label the on-chip data reuse of an Encoder Block in FP and BP. First, the results of the previous layer are cached to serve as inputs for the next layer. Second, weight matrices can also be reused in FP. For example, when GEMM-SA computes the activation matrix $Y_1$ in FFN1, LR-SA computes the weight matrix of FFN2 (i.e., $W_2$) and caches it. At last, we overlap BP and WU. While LR-SA calculates the derivatives of the weight matrix, WUM uses the derivatives to update the weights. Since the number of trainable weights is much smaller than the model parameters, the latency of WUM is much smaller than that of BP. Therefore, WUM is unlikely to be a bottleneck.

There is a significant difference in the workload of LR-SA during FP and BP. In FP, LR-SA calculates $W_0 + B \times A$ in Equation 1. In BP, the workload is the gradients $\frac{\partial L}{\partial A}$ and $\frac{\partial L}{\partial B}$ in Equation 7- 8. As shown in Figure 10 (b), the workload in BP has a higher latency than in FP and thus becomes a bottleneck. To alleviate this problem, we shift part of the

TABLE V
CHARACTERISTICS OF DIFFERENT HARDWARE PLATFORMS

| Platform Frequency | Computing Resource | On-chip Memory | Off-chip Bandwidth |
|---|---|---|---|
| i9-13900K 3.0GHz | 24 cores 32 threads | L1 2.176MB L2 32 MB | 192 GB/s DDR5 |
| Tesla V100 1245MHz | 5120 CUDA cores 640 Tensor cores | Register File 20 MB L2 6 MB | 900 GB/s HBM2 |
| RTX 2080Ti 1350 MHz | 4352 CUDA cores 544 Tensor cores | L1 4.25 MB L2 6 MB | 616 GB/s GDDR6 |
| VCU128 250 MHz | 16640 PEs 256 ALUs | 7.8MB | 128 GB/s* HBM2 |

* HBM2 on VCU128 has 32 ports and a maximum frequency of 450MHz. We use 16 ports and the clock frequency is 250 MHz.

workload (i.e., $B^T \times X^T$ in Equation 7) from BP to FP. Since the latency of GEMM-SA is much higher than that of LR-SA in FP, this adjustment doesn't increase the overall latency of FP.

## VI. EVALUATION RESULTS

### A. Experimental Setup

We evaluate the effectiveness of LR-FFP fine-tuning strategy on different models and tasks. Specifically, we fine-tune BERT-small and DistilBERT on GLUE benchmark [34], CoNLL dataset [35] and Stanford Question Answering Dataset (SQuAD) [38]. These datasets cover sentence classification, named entity recognition and question answering tasks. BERT-small and DistilBERT have 29M and 68M parameters, respectively. We use the deep learning library PyTorch [39] to fine-tune these models.

We implement the F$^3$ accelerator in Verilog HDL on a Xilinx VCU128 FPGA. The VCU128 platform integrates 8GB HBM which provides a peak memory bandwidth of 460GB/s. The power consumption and hardware resource utilization are obtained after synthesis and implementation in Vivado 2021.1. We build a cycle-accurate simulator to evaluate the performance of the whole system and hardware utilization of each module.

We compare the performance and energy efficiency of F$^3$ with one of the state-of-the-art deep learning accelerating frameworks PyTorch. PyTorch can run on both the CPU and GPU. Table V summarizes the characteristics of the hardware platforms. Intel i9-13900K is a high-performance CPU and we use FP32 format for the CPU baseline. Nvidia V100 and 2080Ti are a server GPU and desktop GPU, respectively. Both of them are equipped with deep learning acceleration units (Tensor cores). Theoretically, BF16 format is more efficient than FP32 format. Without hardware support, the performance of these two GPU platforms on the BF16 model is not even as good as that of the FP32 model. In contrast, tensor cores are optimized for FP16 format. Therefore, we use FP16 format for the GPU baseline. The execution time of CPU and GPU is obtained from cProfile and NVProf, respectively. To make a fair comparison of energy efficiency, we collect the dynamic power consumption during transformer fine-tuning. The power consumption of the CPU and GPU is measured by the power meter. For the GPU baseline, we don't consider the power consumption of the host CPU.

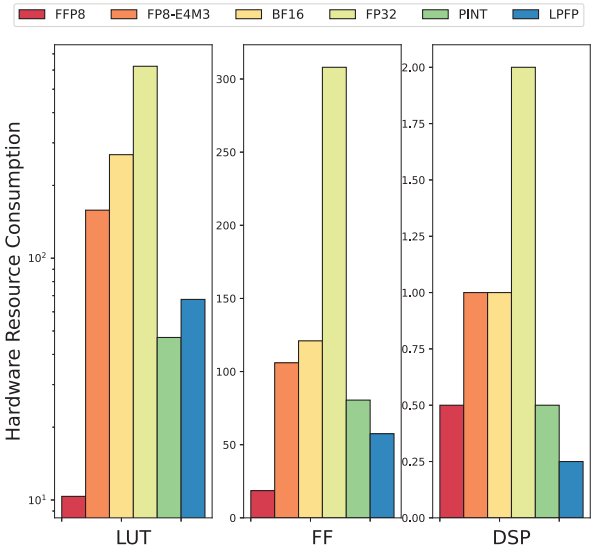



Fig. 11. Hardware resource consumption of MACs. The MAC for FFP8, PINT and LPFP includes encoding logic. We implement MACs of FP8-E4M3, BF16 and FP32 formats with Xilinx's IP and choose medium DSP slice usage.

### B. Accuracy

The model accuracy with different fine-tuning strategies and data formats is listed in Talbe VI. The batch size is 16 for CoNLL dataset and 8 for other datasets. The sentence length is set to 128. We train each model on the datasets for 5-10 epochs. Each model is trained multiple times and the accuracy in Table VI is the average of these results.

We apply LoRA to all linear layers of the transformer model and set the LoRA rank to 8 in the experiment. Compared with full parameter fine-tuning in FP32 formats, LR-FFP reduces the number of trainable parameters to about 1% of the original and its average accuracy degradation is within 0.6%. LR-FFP causes a more serious accuracy degradation on DistilBERT than on BERT-small. We believe this is because DistilBERT is already a compressed version of BERT and further compressing DistilBERT is more difficult than compressing BERT itself. In addition, LR-FFP can meet different accuracy requirements by changing the number of trainable parameters.

### C. Hardware Resource Evaluation

The hardware resource breakdown of F$^3$ is shown in Table VII. Most resource consumptions come from GEMM-SA, LR-SA and VPU. Although LR-SA is only 1/32 the size of GEMM-SA, it consumes more LUTs due to its use of high-precision PEs (supporting BF16 format) and the additional resources required to support OS dataflow. The Encoder and Decoder modules consume few resources.

In Figure 11, we compare the resource consumption of MACs designed for different data formats. Compared with

TABLE VI
ACCURACY OF BERT AND DISTILBERT WITH DIFFERENT FINE-TUNING STRATEGIES

| Model | Method | # Trainable Parameters | Accuracy | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | SST-2 | QNLI | MRPC | RTE | QQP | STS-B | CoNLL | SQuAD [1] | Avg [2] |
| **BERT-small (29M)** | Full Training FP32 | 29M | 89.5 | 86.7 | 83.3 | 63.4 | 88.5 | 81.1 | 88.2 | 69.5/79.1 | 81.27 |
| | Full Training BF16 | 29M | 88.9 | 86.8 | 83.3 | 63.6 | 87.9 | 80.6 | 87.9 | 68.9/78.7 | 80.98 |
| | LoRA FFP8 (LR-FFP) | 0.30M | 88.4 | 87.4 | 83.7 | 63.4 | 87.2 | 80.8 | 87.5 | 68.3/78.0 | 80.84 |
| **DistilBERT (68M)** | Full Training FP32 | 68M | 90.5 | 88.6 | 86.0 | 64.3 | 88.3 | 87.4 | 89.9 | 76.5/84.9 | 83.93 |
| | Full Training BF16 | 68M | 90.4 | 88.1 | 85.1 | 64.3 | 88.2 | 86.7 | 89.4 | 75.4/84.2 | 83.44 |
| | LoRA FFP8 (LR-FFP) | 0.66M | 90.1 | 86.6 | 86.0 | 63.4 | 87.4 | 85.6 | 89.7 | 76.3/84.8 | 83.33 |

[1] The evaluation criteria for SQuAD are accuracy/F1 score.
[2] Average of accuracy across all datasets.

TABLE VII
RESOURCE CONSUMPTION AND POWER OF DIFFERENT MODULES

| | Format | Size | LUT | FF | DSP | BRAM |
|---|---|---|---|---|---|---|
| **Decoder** | FFP8-INT9 | 256 | 7168 | 9228 | 0 | 0 |
| **Encoder** | INT32-FFP8 | 64 | 11056 | 8304 | 0 | 0 |
| **GEMM-SA** | FFP8 | 16384 | 120692 | 267136 | 8192 | 0 |
| **LR-SA** | BF16 | 512 | 138752 | 79232 | 512 | 0 |
| **AA** | INT32 | 256 | 41664 | 28608 | 0 | 0 |
| **VPU** | BF16 | 256 | 80704 | 91904 | 256 | 64 |
| **WUM** | FP32 | 64 | 24608 | 8128 | 64 | 0 |
| **Total** | - | - | 472446 | 578564 | 9024 | 1520 |

our FFP8 MACs, FP8-E4M3 MACs consume 15.3× LUTs, 5.7× FFs and 2× DSPs, respectively. After exponent alignment, the operand bitwidth of FP8-E4M3 is 20 bits, while that of FFP8 is only 9 bits. MACs of BF16 and FP32 formats consume much more resources than that of FP8-E4M3. We also reproduce two other low-precision MACs (i.e., PINT and LPFP) proposed by Shao et al. [30] and Wu et al. [26]. PINT PE is an ASIC-based architecture. In fairness, we apply the double MAC technique to it. Although the bitwidth of the three data formats is all 8 bits, the LUT consumption and FF consumption of FFP8 are much lower than that of PINT and LPFP. There are two main reasons for this. First, they don't utilize the accumulator provided by the DSP. As a result, the intermediate result of MAC has a high bitwidth and the accumulator consumes a large amount of LUTs and FFs. Second, in their architecture, all intermediate results of MAC are aligned. Due to the high bitwidth of the intermediate results, the resource consumption of the exponent alignment module is very high. They configure the exponent alignment module for each PE, which exacerbates this problem. In contrast, we eliminate this overhead by pre-aligning all operands.

### D. Performance and Energy Efficiency

We compares $F^3$ with prior DNN accelerators in Table VIII. The performance and energy efficiency of $F^3$ outperforms previous works. In previous research, only a limited number of works have focused on enhancing the floating-point computing power of the DSP. In addition, we fully utilize the resources within the DSP (e.g., the high bitwith accumulator), which can reduce the consumption of LUTs. Considering that many previous works don't use multi-MAC schemes, we calculate the performance efficiency based on the number of MAC units. LR-FFP strategy reduces the computation and memory access overhead, but introduces LRMM and makes the dataflow more complex. Despite this, we achieve high performance efficiency by optimizing the pipeline.

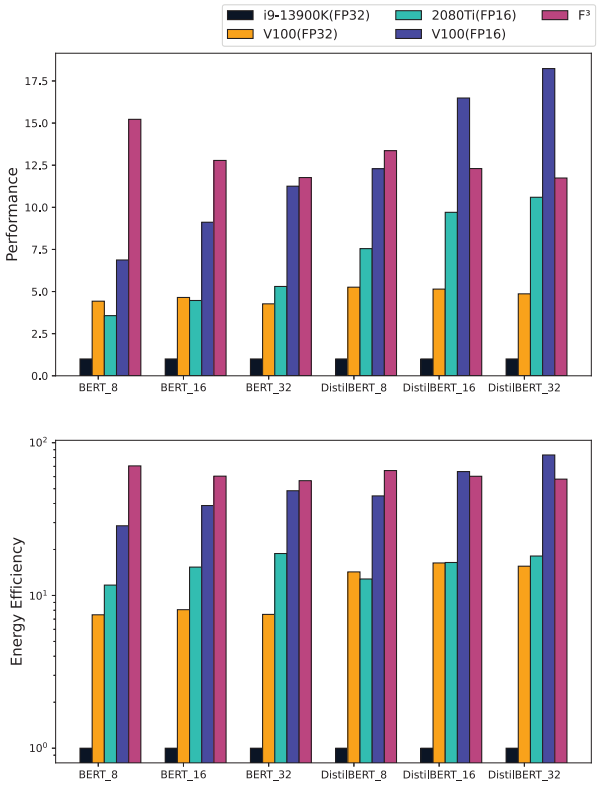



Fig. 12. Performance and energy efficiency comparison with CPU and GPU. BERT 8 indicates that the model is BERT-small and the batch size is 8.

In Figure 12, we compare the performance and energy efficiency of the CPU, GPU and $F^3$. We evaluate each hardware platform using different models and batch sizes. Although transformer fine-tuning is compute-intensive and the GPU is well suited to handle such tasks, $F^3$ still delivers performance advantage and significant energy efficiency benefits. Compared with i9-13900K (FP32) and V100 (FP32), $F^3$ achieves a performance speedup of 11.74 ~ 15.22 and 2.39 ~ 3.44, and an energy efficiency speedup of 56.43 ~ 70.52 and 3.69 ~ 9.44. $F^3$ beats 2080 Ti (FP16) with 1.11 ~ 4.26 performance speedup. As the model size and batch size become larger, the computation latency accounts for an increasing proportion

TABLE VIII
COMPARISON WITH PREVIOUS WORKS

| | [29] | [40] | FitNN [41] | [26] | [17] | [42] | [43] | F$^3$ |
|---|---|---|---|---|---|---|---|---|
| **Platform** | XCU280 | XCVU9P | XCZU3EG | XC7K325T | XC7VX690T | VX690T | XC7VX485T | XCVU37P |
| **Data Format** | INT8/INT4 | INT12/INT8 | INT8 | FP8 | BFP8 | FP8 | INT8 | FFP8 |
| **LUT** | 569K | 706K | 35K | 155K | 232K | 337K | 230.5K | 434K |
| **DSP** | 1780 (896) | 5163 | 238 | 768 | 1027 | 2877 | 2640 | 9024 |
| **MAC** | Double | Double | Single | Quad | Double | Double | Single | Double |
| **Frequency(MHz)** | 245 | 167 | 150 | 200 | 200 | 200 | 200 | 250 |
| **Performance(GOPS)** | 713 | 3375.7 | 38.4 | 1086.8 | 760.83 | 1805.83 | 767.3 | 8202.4 |
| **Power(W)** | - | 45.9 | 4.69 | 9.42 | 9.18 | - | - | 43.2 [2] |
| **Power Efficiency(GOPS/W)** | - | 73.5 | 8.18 | 115.4 | 82.88 | - | - | 197.8 |
| **Perf Density(OPS/DSP/Freq)** | 3.25 | 3.92 | 1.08 | 7.08 | 3.7 | 3.14 | 1.45 | 3.64 |
| **Perf Eff(OPS/MAC/Freq)** [1] | 1.62 | 1.96 | 1.08 | 1.77 | 1.85 | 1.57 | 1.45 | 1.82 |

[1] In fairness, we suppose one DSP can only perform one MAC per cycle.
[2] On-chip power consumption.

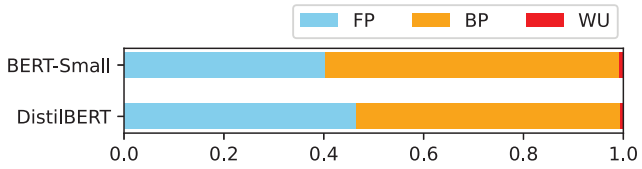



Fig. 13. Latency breakdown. We overlap WU with BP and the red bar represents only the computation latency of WU. Batch size is 16.

of the overall latency. Equipped with tensor cores, V100 and 2080 Ti have an order of magnitude higher computing power (FP16) than F$^3$. Therefore, for large models and batch sizes, the advantage of F$^3$ is not significant. Our fine-tuning strategy and architecture contribute to the improvement in two ways. First, an FFP8 MAC unit only consumes 0.5 DSP and a few LUTs. While an FP32 MAC unit consumes 4 DSPs without using a large number of LUTs. FFP8 improves computing power by 8×. Second, LoRA reduces the number of trainable parameters by two orders of magnitude. As a result, the overhead of gradient computation and WU is significantly reduced.

Figure 13 shows the latency breakdown of the entire fine-tuning process. WU's latency is much less than that of FP and BP. The latency of BP is 1.2 ∼ 1.5× that of FP. As the model size increases, the latency gap between FP and BP decreases. In transformer training, FP, error computation and gradient computation have similar time complexity. Theoretically, BP should have twice the latency of FP. Since LR-FFP greatly reduces the overhead of gradient computation, the latency gap is reduced.

### E. Evaluation of PE Utilization

We evaluate the hardware utilization of BERT-small and DistilBERT during fine-tuning. As illustrated in Figure 14, in FP, GEMM-SA can almost reach peak performance for both BERT-small and DistilBERT. In BP, the GEMM-SA utilization for these models differs significantly. This phenomenon can be explained by Figure 10 (b). Despite the differences in latency of BERT-small and DistilBERT, the latency of LR-SA in FP is always lower than that of GEMM-SA, so LR-SA cannot be

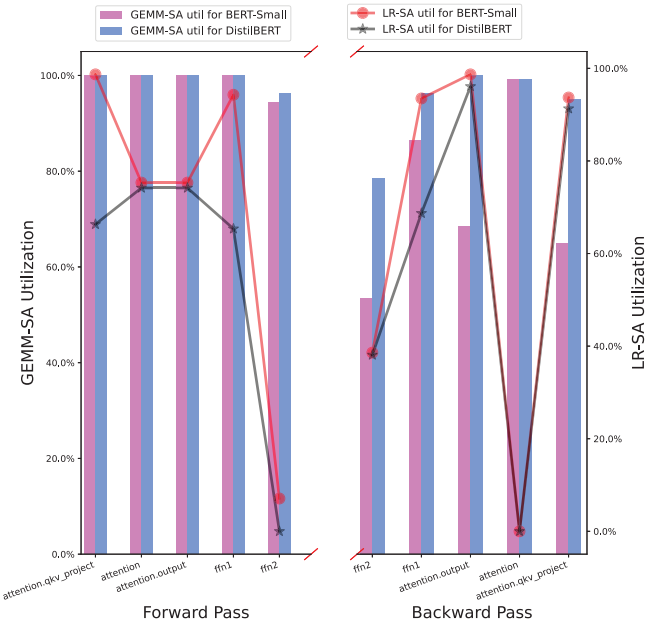



Fig. 14. GEMM-SA and LR-SA utilization in FP and BP for BERT-small and DistilBERT.

the bottleneck. In BP, GEMM-SA and LR-SA calculate the error (Equation 6) and gradient (Equation 7- 8), respectively. LR-SA often becomes a bottleneck thereby reducing GEMM-SA utilization. The time complexity of the error and gradient is $O(bnd^2)$ and $O(bndr)$, where $b$, $n$, $d$ and $r$ are batch size, sequence length, hidden dimension and Lora rank. As the hidden dimension $d$ increases, the time complexity of the error grows faster than that of the gradient. In other words, as the model size grows, the latency of GEMM-SA increases more than that of LR-SA. Therefore, in larger models like DistilBERT, LR-SA may no longer be a bottleneck in BP.

## VII. CONCLUSION

In this article, we propose an FPGA-friendly floating-point format, FFP8. Compared with FP8 and BFP8, FFP8 not only saves hardware resources but is also suitable for both inference and training tasks. We adapt LoRA to FFP8 format

and propose LR-FFP strategy to fine-tune transformer models. Results on different models and a variety of tasks show an accuracy degradation of less than 1% compared with the full-precision model. Based on FFP8 format, we design a PE that enables one DSP to perform two signed MAC in one cycle. Our approach further extends the number of cascaded DSPs and has better scalability than the previous double MAC schemes. Finally, we present our architecture $F^3$ which integrates two specialized SAs with different precision and sizes. These two SAs (GEMM-SA and LR-SA) are suitable for processing GEMM and LRMM. We determine the optimal GEMM-SA size by theoretical analysis. The experimental results demonstrate that $F^3$ achieves a throughput of 8.2 TOPS and has higher performance and energy efficiency than the CPU and GPU baseline. In addition to transformer fine-tuning, FFP8 format can also be applied to training other neural networks. The floating-point double MAC scheme and the design of SA are also applicable to other DNN accelerators.

## REFERENCES


[1] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, Jun. 2017, pp. 5998–6008.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Language Technol.*, vol. 1, J. Burstein, C. Doran, and T. Solorio, Eds., Jun. 2019, pp. 4171–4186.

[3] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[4] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *Proc. Eur. Conf. Comput. Vis.*, Cham, Switzerland: Springer, 2020, pp. 213–229.

[5] A. Dosovitskiy et al., "An image is worth 16×16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Represent.*, Jan. 2020.

[6] Z. Liu et al., "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2021, pp. 10012–10022.

[7] T. B. Brown et al., "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, Jan. 2020, pp. 1877–1901.

[8] H. Touvron et al., "LLaMA: Open and efficient foundation language models," 2023, *arXiv:2302.13971*.

[9] S. Tuli and N. K. Jha, "TransCODE: Co-design of transformers and accelerators for efficient training and inference," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 12, pp. 4817–4830, Dec. 2023.

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," in *Proc. Adv. Neural Inf. Process. Syst.*, Jan. 2023, pp. 10088–10115.

[11] Y. Xu et al., "QA-LoRA: Quantization-aware low-rank adaptation of large language models," 2023, *arXiv:2309.14717*.

[12] Y. Bondarenko, R. Del Chiaro, and M. Nagel, "Low-rank quantization-aware training for LLMs," 2024, *arXiv:2406.06385*.

[13] J. E. Hu et al., "LoRA: Low-rank adaptation of large language models," in *Proc. Int. Conf. Learn. Represent.*, Jan. 2021.

[14] D. Kalamkar et al., "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*.

[15] X. Sun et al., "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, Jan. 2019, pp. 4900–4909.

[16] P. Micikevicius et al., "FP8 formats for deep learning," 2022, *arXiv:2209.05433*.

[17] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance FPGA-based CNN accelerator with block-floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

[18] W. Zhao, Q. Dang, T. Xia, J. Zhang, N. Zheng, and P. Ren, "Optimizing FPGA-based DNN accelerator with shared exponential floating-point format," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 11, pp. 4478–4491, Nov. 2023.

[19] N. Houlsby et al., "Parameter-efficient transfer learning for NLP," in *Proc. Int. Conf. Mach. Learn.*, K. Chaudhuri and R. Salakhutdinov, Eds., Jun. 2019, pp. 2790–2799.

[20] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics, 11th Int. Joint Conf. Natural Language Process.*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., Aug. 2021, pp. 4582–4597.

[21] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proc. Conf. Empirical Methods Natural Language Process.*, M.-F. Moens, X. Huang, L. Specia, and S. W.-T. Yih, Eds., Nov. 2021, pp. 3045–3059.

[22] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. Int. Conf. Learn. Represent.*, Jan. 2017. [Online]. Available: https://openreview.net/forum?id=Bkg6RiCqY7

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent.*, San Diego, CA, USA, Y. Bengio, and Y. LeCun, Eds., May 2015.

[24] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on Xilinx devices," Xilinx, San Jose, CA, USA, White Paper WP486 (v1.0.1), Apr. 2017.

[25] P. Metzgen, "Higher performance neural networks with small floating point," AMD, Santa Clara, CA, USA, Rep. WP530, Jun. 2021.

[26] C. Wu, M. Wang, X. Chu, K. Wang, and L. He, "Low-precision floating-point arithmetic for high-performance FPGA-based CNN acceleration," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, pp. 1–21, Mar. 2022.

[27] J. Zhuang et al., "SSR: Spatial sequential hybrid architecture for latency throughput tradeoff in transformer acceleration," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Apr. 2024, pp. 55–66.

[28] S. Zeng et al., "FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Apr. 2024, pp. 223–234.

[29] H. Chen et al., "Understanding the potential of FPGA-based spatial acceleration for large language model inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 1, pp. 1–29, Mar. 2025.

[30] H. Shao, J. Lu, M. Wang, and Z. Wang, "An efficient training accelerator for transformers with hardware-algorithm co-optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 31, no. 11, pp. 1788–1801, Nov. 2023.

[31] J. Lu, C. Ni, and Z. Wang, "ETA: An efficient training accelerator for DNNs based on hardware-algorithm co-optimization," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 10, pp. 7660–7674, Oct. 2022.

[32] J. Yu, K. Prabhu, Y. Urman, R. M. Radway, E. Han, and P. Raina, "8-bit transformer inference and fine-tuning for edge accelerators," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2024, pp. 5–21.

[33] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[34] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proc. EMNLP Workshop BlackboxNLP, Analyzing Interpreting Neural Netw. NLP*, T. Linzen, G. Chrupała, and A. Alishahi, Eds., 2018, pp. 353–355. [Online]. Available: https://aclanthology.org/W18-5446

[35] E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition," in *Proc. 7th Conf. Natural Lang. Learn. (HLT-NAACL)*, 2003, pp. 142–147.

[36] S. Lee, D. Kim, D. Nguyen, and J. Lee, "Double MAC on a DSP: Boosting the performance of convolutional neural networks on FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 888–897, May 2019.

[37] D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," 2016, *arXiv:1606.08415*.

[38] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000 questions for machine comprehension of text," in *Proc. Conf. Empirical Methods Natural Language Process.*, J. Su, K. Duh, and X. Carreras, Eds., 2016, pp. 2383–2392. [Online]. Available: https://aclanthology.org/D16-1264

[39] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, Jan. 2019, pp. 8024–8035.
[40] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[41] Z. Zhang, M. A. P. Mahmud, and A. Z. Kouzani, "FitNN: A low-resource FPGA-based CNN accelerator for drones," *IEEE Internet Things J.*, vol. 9, no. 21, pp. 21357–21369, Nov. 2022.
[42] S. Yin et al., "A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 678–691, Apr. 2019.
[43] D. T. Nguyen, H. Kim, and H.-J. Lee, "Layer-specific optimization for mixed data flow with mixed precision in FPGA design for CNN-based object detectors," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 31, no. 6, pp. 2450–2464, Jun. 2021.

**Zerong He** received the bachelor's degree in applied physics from the University of Science and Technology of China in 2022, where he is currently pursuing the Ph.D. degree in electronic science and technology. His research interests include computer architecture, reconfigurable computing, machine learning, and in memory computing.

**Xi Jin** is currently an Associate Professor with the Department of Physics, University of Science and Technology of China, where he directs the SoC Laboratory. His group has worked in accelerating scientific computing applications with FPGAs and using heterogeneous platforms to accelerate data processing.

**Zhongguang Xu** is currently a Professor with the Department of Microelectronics, University of Science and Technology of China, where he directs the Intelligent Storage Laboratory. His group has worked in the field of storage class memory computing in memory and smart memory system for AI acceleration.