# TATAA: Programmable Mixed-Precision Transformer Acceleration with a Transformable Arithmetic Architecture

JIAJUN WU, MO SONG, JINGMIN ZHAO, YIZHAO GAO, JIA LI, and HAYDEN KWOK-HAY SO, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong

Modern transformer-based deep neural networks present unique technical challenges for effective acceleration in real-world applications. Apart from the vast amount of linear operations needed due to their sizes, modern transformer models are increasingly reliance on precise non-linear computations that make traditional low-bitwidth quantization methods and fixed-dataflow matrix accelerators ineffective for end-to-end acceleration. To address this need to accelerate both linear and non-linear operations in a unified and programmable framework, this article introduces TATAA. TATAA employs 8-bit integer (`int8`) arithmetic for quantized linear layer operations through post-training quantization, while it relies on `bfloat16` floating-point arithmetic to approximate non-linear layers of a transformer model. TATAA hardware features a transformable arithmetic architecture that supports both formats during runtime with minimal overhead, enabling it to switch between a systolic array mode for `int8` matrix multiplications and a SIMD mode for vectorized `bfloat16` operations. An end-to-end compiler is presented to enable flexible mapping from emerging transformer models to the proposed hardware. Experimental results indicate that our mixed-precision design incurs only 0.14% to 1.16% accuracy drop when compared with the pre-trained single-precision transformer models across a range of vision, language, and generative text applications. Our prototype implementation on the Alveo U280 FPGA currently achieves 2,935.2 GOPS throughput on linear layers and a maximum of 189.5 GFLOPS for non-linear operations, outperforming related works by up to 1.45× in end-to-end throughput and 2.29× in DSP efficiency, while achieving 2.19× higher power efficiency than modern NVIDIA RTX4090 GPU.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; *Single instruction, multiple data*; *Systolic arrays*; • **Hardware** → **Hardware accelerators**; **Emerging architectures**; **Operations scheduling**;

Additional Key Words and Phrases: Transformer acceleration, mixed integer-floating-point inference, transformable arithmetic architecture, non-linear arithmetic operations, systolic array, SIMD, FPGA

Jiajun Wu and Mo Song contributed equally to this research.

This work was supported in part by the Research Grants Council (RGC) of Hong Kong under the Research Impact Fund project R7003-21 and the Theme-based Research Scheme (TRS) Project T45-701-22-R, and in part by the AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR.

Authors' Contact Information: Jiajun Wu, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: jjwu@eee.hku.hk; Mo Song, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: songmo@eee.hku.hk; Jingmin Zhao, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: jmzhao@eee.hku.hk; Yizhao Gao, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: yzgao@eee.hku.hk; Jia Li, Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: lijia@eee.hku.hk; Hayden Kwok-Hay So (corresponding author), Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong, Hong Kong; e-mail: hso@eee.hku.hk.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2025 Copyright held by the owner/author(s).

ACM 1936-7414/2025/3-ART14
https://doi.org/10.1145/3714416

**ACM Reference format:**
Jiajun Wu, Mo Song, Jingmin Zhao, Yizhao Gao, Jia Li, and Hayden Kwok-Hay So. 2025. TATAA: Programmable Mixed-Precision Transformer Acceleration with a Transformable Arithmetic Architecture. *ACM Trans. Reconfig. Technol. Syst.* 18, 1, Article 14 (March 2025), 31 pages.
https://doi.org/10.1145/3714416

## 1 Introduction

Since its introduction in 2017, the transformer model [1] and its variations have rapidly risen to the forefront of modern deep learning architectures. Unlike previous-generation **convolutional neural networks (CNNs)** that were based predominantly on linear operations, modern transformer models increasingly rely on *high-precision non-linear operations* in their designs. For instance, the self-attention mechanism of a transformer model is typically based on the SoftMax function, which has been demonstrated to require high-precision computation in order to achieve a model's accuracy [2]. Normalization layers such as LayerNorm [3] or **root mean square normalization (RMSNorm)** [4] require complex non-linear operations on data that cannot easily be fused into preceding linear layers. Finally, sophisticated activation functions such as GELU [5], SiLU [6], and SwiGLU [7] are often used in transformer models which require precise computation, unlike CNNs.

To address the need to approximate these non-linear functions with high-precision and high performance, specialized hardware modules have previously been extensively explored [8–11]. Yet, customized hardware must be designed for every non-linear function that is being employed in a model, which is impractical when new non-linear operations for transformers are still actively being developed in this rapidly evolving field [12, 13]. Other researchers have focused on quantizing such non-linear functions into low-bitwidth fixed point formats in order to reduce the computation complexity [14– 17]. Due to the outliers in transformers [18, 19], retraining is generally required for such quantization to maintain good accuracy. However, the large size of modern transformer models, data availability and privacy concerns, have all made such retraining method impractical in most real-world scenarios. Besides, existing accelerators either rely on individual and specific non-linear function units [20–22], or attempt to handle non-linear functions with general arithmetic logic units [17]. Both strategies often lead to increased hardware overhead, reduced hardware efficiency, and it complicates the workload balance between linear and non-linear layers. Instead, to improve future compatibility, a general-purpose transformer accelerator that can be reprogrammed to support new non-linear functions in floating-point arithmetic with low hardware overhead is highly desirable.

In this article, we present a novel end-to-end framework for flexible and quantized **transformer acceleration using a transformable arithmetic architecture (TATAA)** that supports both floating-point and integer operations (Figure 1). In TATAA, static **post-training quantization (PTQ)** is performed on the linear layers of a transformer model to facilitate **8-bit integer (`int8`)** operations. On the other hand, non-linear layers are performed using high-precision `bfloat16` operations. In contrast to certain previous efforts that reduced data bitwidths in non-linear layers, we alleviate the need for retraining by maintaining high bitwidth data formats in these layers, all the while ensuring high efficiency during execution. To support both `int8` and `bfloat16` operations efficiently, TATAA hardware architecture consists of **dual-mode processing units (DMPUs)** that feature configurable arrays of integer **processing elements (PE)**. The proposed architecture can be transformed between a systolic array for `int8` **matrix multiplications (MatMul)** and a SIMD-like vectorized `bfloat16` computing unit. In particular, the proposed TATAA architecture

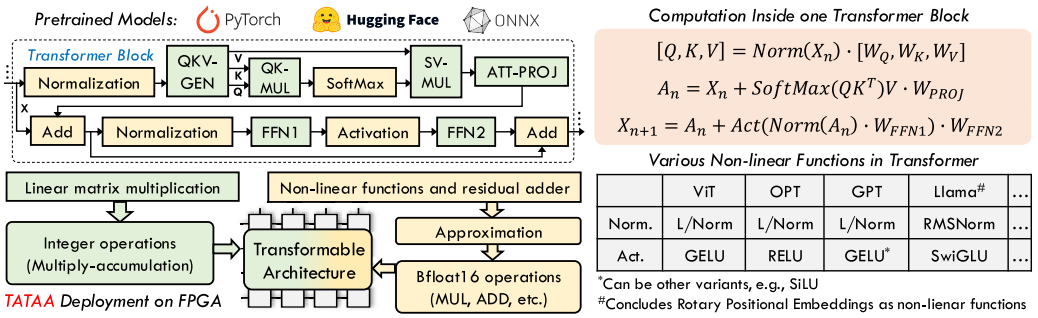


Fig. 1. Illustration of a typical transformer block, and how TATAA maps different operations in transformers including linear *MatMul* and a variety of non-linear functions into transformable architecture, based on a general end-to-end framework design.

employs a single type of processing unit, which is reused for all run-time operations, leveraging the bit field patterns of `bfloat16`. This design choice minimizes hardware overhead and maximizes flexibility compared to previous studies. By minimizing the overhead for run-time reconfiguration, the proposed transformable architecture ensures the high hardware processing density necessary to deliver the highest performance on FPGAs with limited resources. Finally, a compilation framework is developed that maps the user-provided transformer models to the custom **instruction set architecture (ISA)** of the TATAA processor cores to facilitate all operations in both linear and non-linear layers.

To the best of our knowledge, TATAA is the first FPGA-based acceleration framework for transformer inference that integrates floating-point non-linear functions into integer-based linear processing units. It is programmable and is ready to support emerging transformer models with potentially new non-linear functions. Our experimental results indicate that when simulating model performance with a hybrid data format for transformer inference, TATAA achieves only a minimal accuracy reduction, with 0.14% to 1.16% decrease across all evaluated models compared to the original pre-trained model in single-precision floating-point (`fp32`). Additionally, the FPGA accelerator reaches a peak throughput of 2,935.2 **giga-operations-per-second (GOPS)** for `int8` linear operations and a peak throughput of 169.8 **giga-floating-point-operations-per-second (GFLOPS)** when the processor is configured for `bfloat16` non-linear operations at a clock frequency of 225 MHz. Compared to related studies, TATAA achieves up to 1.45× higher throughput and 2.29× higher throughput efficiency on DSP blocks. With the transformable architecture for non-linear functions, our implementation achieves 4.25× lower latency for these complex `bfloat16` operations compared with other works, while supporting flexible and general compilation for emerging functions. Our end-to-end compilation framework also presents optimal mapping from non-linear functions to hardware ISA by appropriate approximation schemes and efficient dataflow control. Moreover, compared to state-of-the-art GPUs, our TATAA architecture outperforms a maximum 2.19× higher power efficiency over a variety of transformer models. This prototype underscores the potential of the TATAA approach for expediting transformer models and sets the stage for future optimization at the microarchitectural level, while our extensive compilation flow opens up a significant optimization space for non-linear functions and to quickly adapt to future transformer-based model as they are being developed. To inspire the community with our proposed novel angle toward transformer acceleration, we release the latest version of the open source code at https://github.com/CASR-HKU/TATAA.

## 2 Background and Related Works

### 2.1 Transformer and Quantization

The transformer architecture [1], along with its various derivatives, such as the **vision transformer (ViT)** [23–25], and language models including BERT [26], OPT [27], GPT [27], and Llama [12] have been extensively utilized in numerous applications. Regardless of the overall topology, the fundamental unit in these models, the transformer block, typically includes components such as MLP, activation. Figure 1 illustrates a block in ViT, where the green components represent linear *MatMul*, and the yellow components denote non-linear functions or residual adders. In detail, linear *MatMul* layers encompass the generation of self-attention entries (QKV-GEN), multiplication of $QK^T$ to calculate attention weights (QK-MUL), the *MatMul* operation between the SoftMax-applied attention scores and $V$ entries (SV-MUL), followed by three feed-forward networks (ATT-PROJ, FFN1, and FFN2). Accordingly, the non-linear functions incorporate normalization, SoftMax, and activation, which differ across various transformer-based models, as illustrated in Figure 1. There is also another line of works focusing on block-wise quantization, like **block floating-point (BFP)** [28, 29] which keeps mantissa computation in integer but uses more fine-grained scaling factors in customized blocks. Although we discuss integer quantization in this work, our proposed architecture can also fit these BFP-based quantization methods by adding extra shared exponent processing units.

Due to the efficiency of integer *MatMul* operations on hardware, linear quantization has been widely applied in modern deep neural networks, including transformer models, to reduce memory footprint and computational complexity. Equation (1) presents the switching between floating-point ($fp$) format and quantized integer number ($q$), in terms of a basic multiplication $z = x \cdot y$. Such a quantized basic operation can be extended to any linear operations (e.g., *MatMul*) by giving a sufficient intermediate integer bitwidth to avoid overflow.

$$\begin{aligned} q_x = \lfloor x_{fp}/S_x \rfloor, q_y = \lfloor y_{fp}/S_y \rfloor \\ z_{fp} = x_{fp} \cdot y_{fp} = q_x S_x \cdot q_y S_y \\ q_z = \lfloor q_x S_x \cdot q_y S_y / S_z \rfloor = \lfloor (q_x \cdot q_y) S_x S_y / S_z \rfloor \end{aligned} \quad . \tag{1}$$

According to Equation (1), the key components to deploy quantized operations are scaling factors. To determine the scaling factors for each layer in transformer models, the primary quantization approaches are PTQ [18, 30–32] and **quantization-aware training (QAT)** [14, 15, 33]. Since QAT requires fine-tuning and retraining with expensive overhead [34], exploring the static PTQ approach is more practical in transformer applications and is applied in our quantization framework [32, 35]. In TATAA, we develop the quantization emulator based on a hardware matching style instead of *fake quantization* to get more convincing results, following the HAWQ setups [36]. Besides, TATAA can integrate existing PTQ schemes like FQ-ViT [32] and SmoothQuant [35]. The static PTQ scheme only requires to access a relatively small part of dataset for calibration and getting all the scaling factors (i.e., $S_x, S_y, S_z$) before deploying inference. Once we have the scaling factors, our mixed-precision quantization can be done through Equation (1), switching between floating-point and integer numbers for different kinds of layers.

### 2.2 Non-Linear Functions in Transformer

Beyond integer-based *MatMul* layers, transformers require non-linear functions to achieve high performance. For example, SoftMax [37], Normalization (e.g., LayerNorm, RMSNorm) [3, 4], and activation functions (e.g., GELU,[1] SiLU, SwiGLU) [5–7] in Equation (2), are commonly used in

[1]We use $tanh$ approximation of GELU function in this work.

transformers to extract self-attention features, activate the feed-forward block, and normalize the output of each block, respectively. These non-linear operations and their variants are essential yet costly building basis of transformer models and can be difficult to implement directly or efficiently on hardware. The linear quantization methods described in Equation (1) no longer suit non-linear functions due to more complex operations and higher range and precision requirement during runtime.

$$\begin{aligned}
\text{SoftMax}(\mathbf{x}) &= \frac{\exp(\mathbf{x})}{\sum_i \exp(x_i)} \\
\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \text{E}[\mathbf{x}]}{\sqrt{\text{Var}[\mathbf{x}] + \epsilon}} \cdot \gamma + \beta, &\quad \text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\text{E}[\mathbf{x}^2]}} \cdot \gamma \\
\text{GELU}(\mathbf{x}) &= 0.5 \cdot \mathbf{x} \cdot \left(1 + \tanh\left(\sqrt{2/\pi}\left(\mathbf{x} + 0.044715\mathbf{x}^3\right)\right)\right) \\
\text{SiLU}(\mathbf{x}) = \mathbf{x} \cdot \sigma(\mathbf{x}) = \mathbf{x} \cdot \frac{1}{1 + \exp(-\mathbf{x})}, &\quad \text{SwiGLU}(\mathbf{x}) = (\mathbf{x} \cdot \sigma(\mathbf{x})) \cdot (\mathbf{x} \cdot \sigma(W\mathbf{x} + b))
\end{aligned} \quad . \tag{2}$$

To alleviate hardware inefficiency, several works proposed approximation techniques based on integer-only arithmetic [15] or reduced precision computation [38], and several researchers argued that **lookup table (LUT)**-based methods [39, 40] demonstrated both negligible model accuracy degradation and higher computational efficiency. Especially in [11], the miscellaneous non-linear operations are element-wise handled by a special function unit, which requires special breakpoints of vectors to perform in fine granularity to hide computation latency and wire resources. LogicNets [41] and NullaNet [42] act as general architectures that encapsulate all the operations embedded in linear or non-linear layers by enumerating the truth table values and can be further optimized following logic optimization algorithms. All these implementations of non-linear functions necessitate additional hardware units beyond the linear processing units with larger bitwidth support. In TATAA, we opt to utilize the same hardware processing units for both types of layers and comprehensive support.

### 2.3 Transformer Accelerators

Various transformer acceleration frameworks for efficient inference have been proposed based on GPU [47, 52], ASIC [16, 18, 43], and FPGA [11, 17, 20, 21, 44, 45, 48–51, 53–58]. Unlike GPU and ASIC design, FPGA has attracted much attention recently, thanks to the configurable and flexible nature of FPGA devices, which has released the low hardware utilization rate issue in fixed architecture GPU or ASIC designs [49]. In terms of hardware architecture, part of existing accelerators focus on linear *MatMul* only, without full support for transformer models [18, 44, 45, 58]. In addition, all other designs with full support for transformer implement individual float-point units [17] or specific modules for non-linear functions [16, 20, 53, 59]. Among them, spatial architecture that allows a deep pipeline between different layers has been selected in many previous works [20, 50, 51], to reduce off-chip memory I/O. The limited on-chip resources on FPGA challenge such a design choice, especially when the transformer models have a larger and larger scale. On the contrary, TATAA utilizes a transferable architecture, allowing full support for all operations in transformer models by compiling non-linear functions into basic operations. Our proposed design reuses integer PEs for all operations within transformer models, thereby avoiding additional hardware costs for the small-workload non-linear functions, as shown in Figure 1. Table 1 presents the qualitative comparison between TATAA and the relative

Table 1. Qualitative Comparison with Related Software-Hardware Co-Design Transformer Acceleration Frameworks

| Work | End-to-End Support | Retrain/ Fine-Tuning | Hardware Platform | *MatMul* Data Format | Non-Linear Data Format | Non-Linear Implementation |
|---|---|---|---|---|---|---|
| $A^3$ [43] | No | N/A | ASIC | int8 | Integer | N/A |
| Mokey [18] | No | No | ASIC | fxp | N/A | N/A |
| Auto-ViT-Acc [44] | No | Yes | FPGA | Mixed integer | fp32 | Host CPU |
| Zhang et al. [45] | No | Yes | FPGA | int8 | fp32 | N/A |
| FQ-BERT [46] | Yes | Yes | FPGA | int8 | fxp | Special units |
| I-ViT [14] | Yes | Yes | GPU | int8 | Integer | GPU vector units |
| I-BERT [15] | Yes | Yes | GPU | int8 | Integer | GPU vector units |
| Transformer Engine [47] | Yes | N/A | GPU | fp8 | fp16/fp32 | GPU vector units |
| ViA [20] | Yes | No | FPGA | fp16 | fp16 | Special units |
| SwiftTron [16] | Yes | Yes | ASIC | int8 | fxp | Special units |
| FTRANS [48] | Yes | No | FPGA | fp16 | fp32 | Special units |
| Huang et al. [21] | Yes | Yes | FPGA | int8 | int8 | Special units |
| FlexRun [49] | Yes | Yes | FPGA | int8 | fp32 | Vector units |
| SSR [50] | Yes | N/A | FPGA | int8 | fp32 | Special units |
| FlightLLM [11] | Yes | N/A | FPGA | int4 | fp16 | Special units |
| Chen et al. [51] | Yes | N/A | FPGA | int8 | fp16 | Special units (spatial pipeline) |
| TATA (ours) | Yes | No | FPGA | int8 | bfloat16 | Reuse *MatMul* hardware for vectors |

works. Note that TATAA presents a new but orthogonal angle for transformer accelerators compared to the spatial architecture, and it also has the potential to achieve an efficient pipeline in TATAA design.

## 3 Motivation

As shown earlier, while linear layers such as self-attention and MLP can be easily quantized to integers and deployed on *MatMul* processing units, quantizing other non-linear layers without sacrificing model performance is challenging unless one applies QAT. Furthermore, maintaining non-linear functions in higher precision (such as floating-point) and creating specialized processing units for these less dominating functions results in significant hardware overhead and low hardware utilization. Thus, our primary motivation is: *Can we develop a unified processing unit that efficiently supports linear layers in integer and non-linear layers in floating-point*?

Given a floating-point number $x$ with its significant bit $s_x$, exponent $e_x$ and mantissa $m_x$, the real value of $x$ can be represented as $(-1)^{s_x} \cdot 2^{e_x - e_b} \cdot m_x$ (the exponent bias is $e_b$). Then, **floating-point multiplication (fpmul)** of two numbers $x$ and $y$ is

$$x \cdot y = (-1)^{s_x \wedge s_y} \cdot 2^{e_x + e_y - e_b} \cdot (m_x \cdot m_y). \tag{3}$$

In this context, $e_x + e_y - e_b$ and $m_x \cdot m_y$ are operations on integers (specifically, unsigned integers) with a small bitwidth. Consequently, we can implement fpmul using integer processing units with minimal overhead for the significant bit $s$. Standard **floating-point addition** (***fpadd***), as another

**Algorithm 1:** Fast Inverse Square Root

**Input:** Input bfloat16 number $y$
**Output:** The inverse square root result $\frac{1}{\sqrt{y}}$

1: $y_{int} = y.view(int16)$ ▷ Does not change data bits, only changes the data format it refers to
2: $t_{int} = \texttt{0x5f37} - (y_{int} >> 1)$ ▷ `0x5f37` is the magic number in `int16` [60]
3: $t = t_{int}.view(bfloat16)$
4: $\frac{1}{\sqrt{y}} = y \cdot (1.5 - (y \cdot 0.5 \cdot t^2)$ ▷ Define $t^2$ as *fpapp* operation. In TATAA, *fpapp* is one of the basic operations

basic operation, can be represented as

$$x + y = (-1)^{s_z} \cdot 2^{e_z - e_b} \cdot m_z$$
$$e_z = \begin{cases} e_x, & e_x > e_y \\ e_y, & e_y \geq e_x \end{cases}, \quad \Delta e = \begin{cases} e_x - e_y, & e_x > e_y \\ e_y - e_x, & e_y \geq e_x \end{cases}. \tag{4}$$
$$\{s_z, m_z\} = \begin{cases} \{s_x, m_x\} + \left(\{s_y, m_y\} \gg \Delta e\right), & e_x > e_y \\ \{s_y, m_y\} + \left(\{s_x, m_x\} \gg \Delta e\right), & e_y > e_x \end{cases}$$

In Equation (4), we have already merged the significant bit and mantissa field and transformed this fixed-point number (`fxp`) into 2's complement for integer operations. It can be seen that *fpadd* is more complex than *fpmul* due to the alignment of the mantissa. However, after converting to 2's complement, this series of operations becomes integer addition and multiplication. Specifically, the right shift can be performed using *fpmul* with a small LUT. The only additional overhead is the conversion between **signed digits (SDs)** and 2's complement, as well as the small LUT for right shift.

Since floating-point division is inherently costly, we aim to speed it up using integer operations. To achieve this, we employ the fast inverse square root algorithm [60], which decomposes division into integer operations, as shown in Equation (5) and Algorithm 1. Observe that the $t^2$ computation within this algorithm is distinct from the basic *fpmul* and *fpadd* operations. Hence, we designate it as an *approximated* calculation for the square root and division, abbreviated as *fpapp*. This term will be utilized in the subsequent discussion in this article.

$$\frac{x}{y} = x \cdot \frac{1}{y} = \begin{cases} x \cdot \frac{1}{\sqrt{y}} \cdot \frac{1}{\sqrt{y}}, & y > 0 \\ x \cdot \frac{-1}{\sqrt{-y}} \cdot \frac{-1}{\sqrt{-y}}, & y < 0 \end{cases} \tag{5}$$

Based on transformable arithmetic, all basic floating-point operations can be transformed to a series of integer atom operations. As `int8` has been the most commonly used format for linear layers quantization, we choose `bfloat16` as the high-precision format for non-linear functions, featuring an 8-bit exponent and an 8-bit mantissa. The `bfloat16` format has been extensively employed in the deep learning field for many years and has developed into a well-established standard for both training and inference [61]. The bitwidth of this unique floating-point format is perfectly aligned with the widely used `int8`, for both the exponent and the mantissa. Consequently, based on the analysis aforementioned, it is feasible to repurpose standard `int8` processing units for fundamental `bfloat16` operations, such as *fpmul*, *fpadd*, and *fpdiv* discussed in this section. We also find that the most commonly used architecture for *MatMul*, systolic array, can actually match the vectorized floating-point execution in terms of computation and data layout. We will further explain the details of hardware design, ISA support, and workload mapping in the following sections.

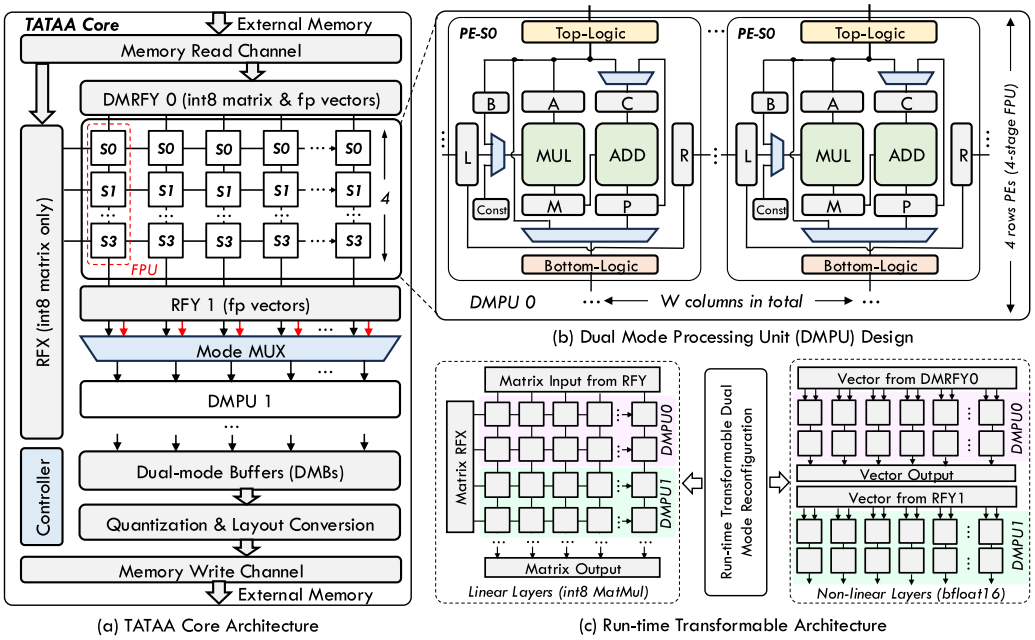


Fig. 2. TATAA hardware architecture and DMPU design. The TATAA core can be configured for two kinds of workload during runtime, to support both linear *MatMul* and non-linear functions.

## 4 Hardware Design

### 4.1 System Architecture

Figure 2 illustrates the proposed hardware architecture for the TATAA inference scheme. The on-chip accelerator comprises $K$ processing cores that function independently with their own run-time instructions, while only one TATAA core is presented for simplification. Given the prevalent use of **high-bandwidth memory (HBM)** today, we have configured the processing cores with individual memory interfaces to communicate with external memory, thereby optimizing the memory bandwidth utilization rate. To prevent data synchronization issues between cores, we have deliberately divided the workloads among different cores without data dependency, which will be further detailed in our compilation framework. As shown in Figure 2(a), each TATAA core contains $N$ DMPU with the integer PE design presented in Figure 2(b). The multi-DMPU architecture can be configured for two types of workload in transformer models, as shown in Figure 2(c). In `int8` *MatMul* mode, all $N$ DMPUs are connected to form a single systolic array. In contrast, in `bfloat16` mode, the DMPUs function independently in a SIMD-like manner to execute vectors. The Mode MUX depicted in Figure 2(a) manages the run-time configuration that controls the connections between DMPUs with a shared controller. We abstract the on-chip data memory as **register files (RFs)** for better high-level abstract and compilation. The input data RFs are separated to the X and Y directions (RFX and RFY), corresponding to the horizon and vertical directions in a common systolic array for *MatMul*. They can be configured in different modes and store different formats of data during runtime. Additionally, we incorporate a quantization unit and an on-chip layout conversion module to quantize output results across layers and handle various data layouts between different operations.

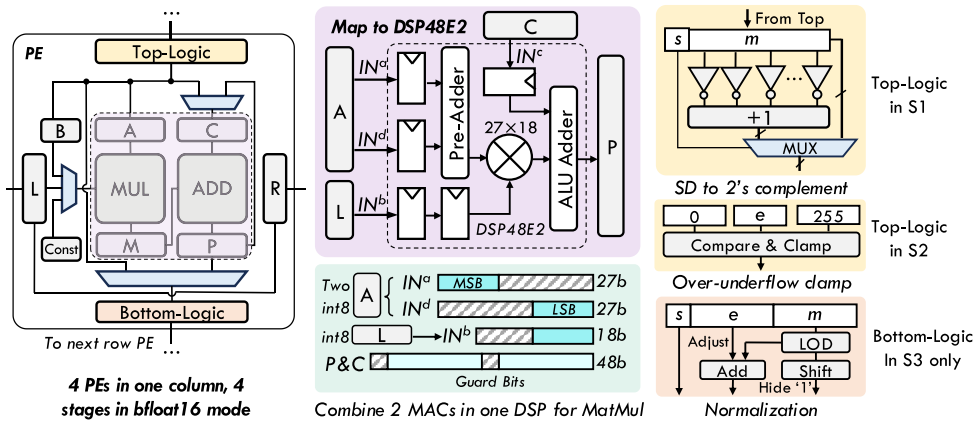


Fig. 3. PE design in the proposed DMPUs when deployed on FPGA devices. The converter from 2's complement to SD logic that locates in PE-S3's bottom logic is not depicted in this figure since it is similar to the SD to 2's complement circuit in PE-S1.

### 4.2 DMPU

The key component in the TATAA architecture is DMPU which is configurable for two data formats. As shown in Figure 2(b), each DMPU comprises $W$ columns by 4 rows of the PE array. The PE is designed for integer **multiply-accumulation (MAC)** operations, with an integer **multiplier (MUL)** and a large bitwidth **adder (ADD)**, as standard setups in `int8` *MatMul*. In the `int8` *MatMul* mode, all the $N$ arrays in the DMPUs are connected to function as a unified $W$ by $4N$ systolic array, and the results of the bottom PEs in one DMPU will be fed to the next DMPU as the top input, controlled by the mode MUX in Figure 2(a). Based on the conventional integer MAC PE design, we expand it by adding low-overhead top and bottom logic units and some extra MUX to support mapping `bfloat16` operations into it. To enhance the throughput of *MatMul* and address the memory-bound challenge in transformer models, we have implemented two loading ports per TATAA core to interact with external memory, given the presence of two matrices in *MatMul*. It's important to highlight that these memory ports are not fixed to specific RFs, RFX or RFY; instead, they are dynamically managed by a crossbar which routes the RFs to the appropriate port.

When TATAA works in the `bfloat16` mode, each DMPU functions autonomously following a SIMD-like process, receiving data from the corresponding RFY instead of the previous DMPU. Each column of the PE array is considered a **floating-point unit (FPU)**, and the 4 rows become 4 pipeline stages in a floating-point, naming PE-S0 (the first stage), PE-S1, PE-S2, and PE-S3 (the last stage). The results are buffered in **dual-mode buffers (DMB)** in both modes before being stored back to external memory. Note that the intermediate `bfloat16` results can also be written to RFY for further computation, avoiding frequent I/O access. Consequently, the core utilizes $W \cdot N$ parallel SIMD lanes in the `bfloat16` mode, allowing the software stack to specify vectorized operations in `bfloat16` with a maximum vector length of $W \cdot N$. The switch of execution modes is completely online without reconfiguring the hardware, as presented in Figure 2(c), thanks to the custom ISA design in TATAA.

### 4.3 PE Design on FPGA

Figure 3 shows the central component of DMPU in TATAA, specifically, the PE. Within this PE, the MUL and ADD perform MAC tasks for *MatMul* layers, alongside executing basic integer multiplication or addition in the `bfloat16` mode according to the breakdown described from Equations

(3)–(5). As TATAA aims to reuse the same hardware units for these diverse functions, multiple MUX are incorporated within the PE to manage data pathways in different modes. Additionally, top- and bottom-logic are executed through LUTs on FPGA for additional operations with minimal overhead, such as normalization and overflow or underflow clamping, vital for all floating-point calculations. Moreover, since DSP48E2 block in modern AMD FPGAs has large MUL and ADD bitwidth ($27 \times 18$ for MUL and 48-bit ADD), a combined MAC optimization is implemented in each PE to enhance run-time throughput, a strategy commonly adopted in various FPGA accelerators [62, 63]. When functioning in `MatMul` mode, the PEs are organized as a large systolic array, integrating multiple DMPUs to maximize throughput.

As demonstrated previously, `bfloat16` operations are perceived as a four-stage pipeline, with each stage assigned to one PE within a column. To utilize the same hardware, operations in `bfloat16` format must be transformed from the original SD format of the floating-point standard into the 2's complement format used in DSP blocks, with conversions back to SDs required before storing the results in memory. Consequently, the PE must include additional processing units specifically for `bfloat16` mode, identified as top-logic and bottom-logic in Figure 3. It is important to highlight that the top-logic architecture varies across different PE stages, and the extra circuits have minimal overhead. For example, the converter from SD to 2's complement is implemented in PE-S1 (second stage), while the logic handling overflow and underflow clamping is placed in PE-S2, and only PE-S3 contains the bottom logic needed for final normalization prior to outputting the `bfloat16` result. In detail, the SD to 2's complement converter in Figure 3 concludes with a bitwise inverter, a +1 ADD, and a MUX to select positive or negative data as the output. In addition, since the exponent in `bfloat16` is 8-bit, the corresponding PE needs to clamp exponent from 0 to 255, thus implementing such a unit in top-logic. The normalization unit is the same as the standard normalization design in common FPUs, with a leading one detector to align the mantissa and hide the hidden '1' and an ADD to adjust exponent after shifting mantissa. In addition, each PE contains several constant registers for `bfloat16` mode in each stage, as the input of MUL. The detailed dataflow of *MatMul* mode and `bfloat16` mode will be thoroughly discussed in the following sections, explaining how these top- and bottom-logic places in different stages of PEs.

### 4.4 Dataflow

The architecture proposed in TATAA can be configured for `int8` *MatMul* mode and `bfloat16` mode during runtime. Figure 4 shows the dataflow in the `int8` *MatMul* mode, where all PEs are connected as a systolic array, and intermediate results accumulate across DMPUs. We choose to deploy the output stationary dataflow for MatMul. In such an execution flow, the **X** and **Y** matrices go through the systolic array in the X (horizontal) and Y (vertical) directions, respectively. Registers L and R are responsible for horizontal and vertical data passing, while the bottom register directly accepts the data from the top and sends them to the next PE. After *MatMul* finishes, the results stored in register P will be sent to the corresponding DMB. The intermediate sums are accumulated in the `int16` format and subsequently quantized to either `int8` or `bfloat16` before being saved to external memory, depending on the format required by the subsequent layer. The static scaling factors are pre-loaded to the quantization unit in the TATAA core before *MatMul* starts.

When the TATAA architecture is set in `bfloat16` mode, it can execute three basic operations: multiplication (*fpmul*), addition (*fpadd*), and the approximation step for the inverse square root in Equation (5) and Algorithm 1 to support $(0x5f37 - (y_{int} >> 1))^2$ (*fpapp*) operation. These operations can be assigned directly to the 4 pipeline stages in the 4 rows of integer-based PE, as illustrated in Figure 5. Thanks to the arithmetic analysis from Equations (3)–(5), we can convert `bfloat16` operations into a sequence of integer operations. The integer MUL and ADD in Figure 5 are reused in the `bfloat16` mode for higher resource efficiency. The floating-point pipeline not

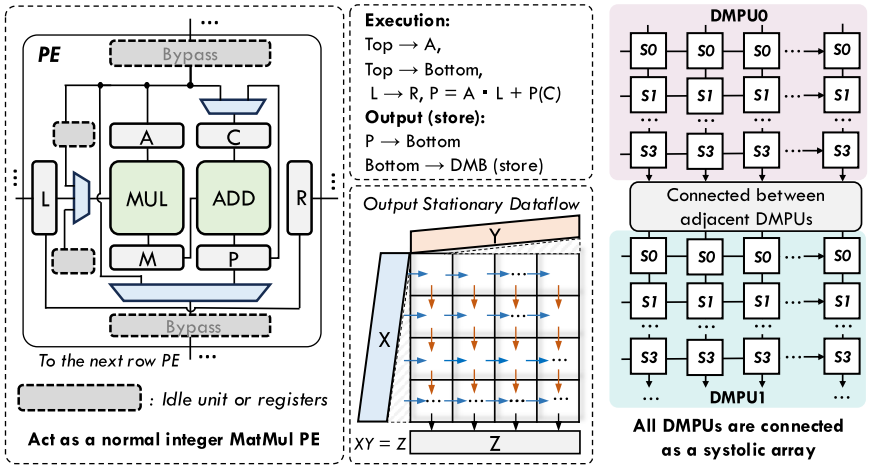


Fig. 4. Execution dataflow in `int8` *MatMul* mode, in which all the PEs are connected as a whole systolic array and deploy output stationary dataflow.

only adapts to integer arithmetic but also shares similar extra top- and bottom-logic, significantly reducing hardware overhead. The overall overhead encompasses converters for SDs and 2's complement, a compact 2's power LUT, typical overflow and underflow management, and a normalization unit, all of which are standard components in conventional FPUs. Specifically, the special *fpapp* first treats the input `bfloat16` binary number as `int16`, performs integer subtraction (addition) in the first stage, and converts the integer binary number back to `bfloat16` with the remaining three stages for square. Using this arithmetic mapping, the proposed DMPU can execute one operation per column in parallel, thus improving the SIMD execution of `bfloat16` operations from a global perspective. This type of sharing scheme between two modes significantly reduces the consumption of hardware resources so that TATAA can map more parallel cores when resources are limited.

### 4.5 RFs and Buffers

Since the proposed TATAA framework is for flexible acceleration, abstract RFs are required for efficient ISA and compiler design. As shown in Figure 6(a), the RFX is available only in the `int8` *MatMul* mode since the `bfloat16` data only go through the Y direction. Therefore, the abstract concept "*registers*" in RFX is actually matrix buffers, and we set two registers inside (RMX0 and RMX1) to apply double-buffer optimization, hiding the memory I/O latency. The RFX has $N$ ports to send data to corresponding $N$ DMPUs. In TATAA, we define that each of the X matrix buffers has $D_{mat}$ depth for MatMul, so the total depth of RFX is $2 \cdot D_{mat}$.

In terms of RFY, we set up two memory banks named RFYa and RFYb, to support the two input operators in the `bfloat16` mode, and each address stores one part of the parallel vector, as shown in Figure 6(b). The data layout becomes more complex because only the RFY for DMPU0 (Dual-mode RFY, DMRFY0) needs to store both `int8` matrices and `bfloat16` vectors. When it works in `int8` *MatMul* mode, the DMPU only needs one specific part of RFYa and RFYb. The two-bank design naturally supports double-buffer optimization (selected by the MUX), so each of them only costs $D_{mat}$ depth. Like RFX, we abstract the matrix buffers as RMY0 and RMY1 physically corresponding to RFYa and RFYb. For the other RFY, they can only be used in the `bfloat16` mode. Hence, the depth of these RFs is set to $D_{fpv}$ ($D_{fpv} \ll D_{mat}$), so the extra memory overhead of these `bfloat16` vectors is relatively small. There is a natural data layout conflict between `int8` and `bfloat16`.

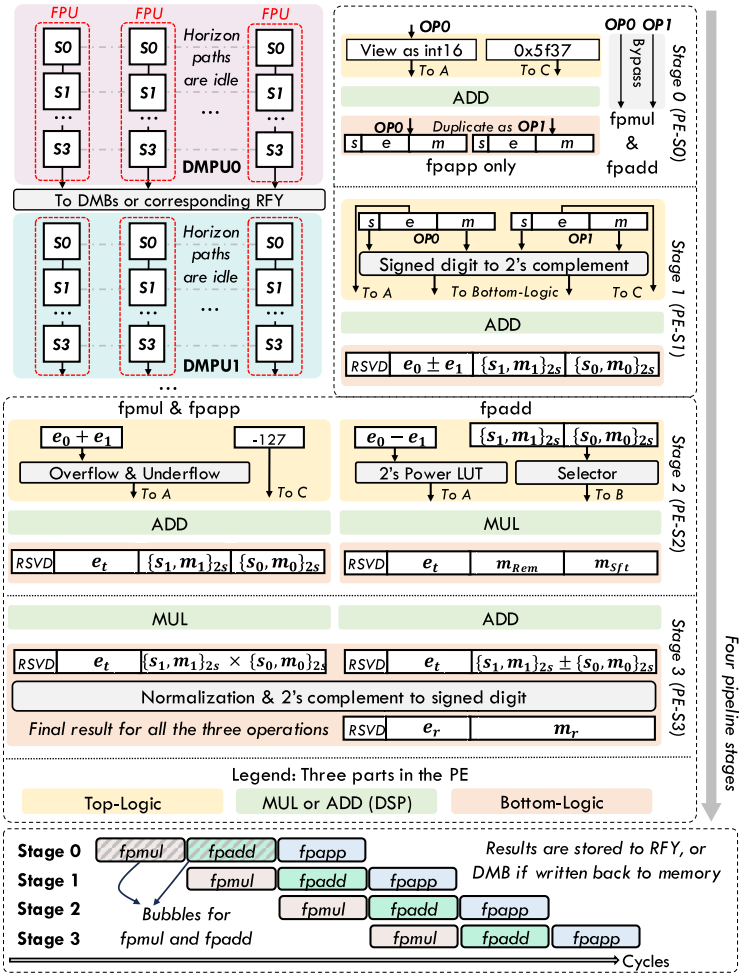


Fig. 5. bfloat16 mode dataflow and how to reuse the integer processing units. The top-logic, MUL and ADD, and bottom-logic processing are depicted in specific colors in this figure.

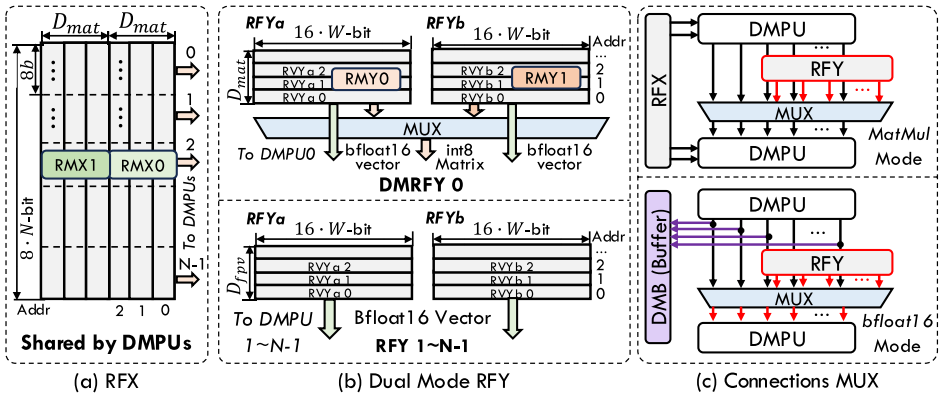


Fig. 6. RF design in TATAA, and the connections between DMPU, RF, and DMB in different modes.

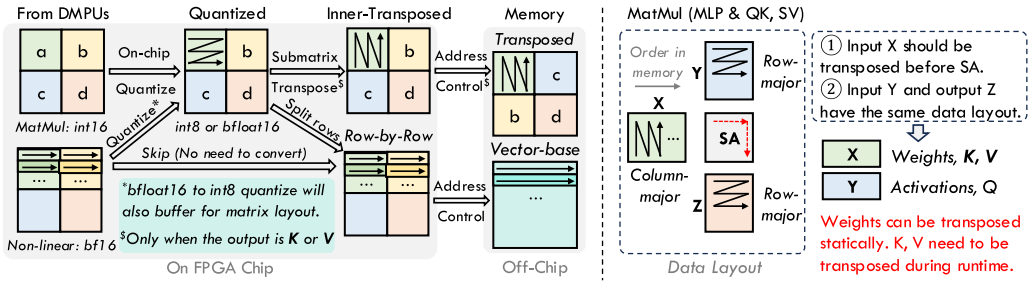


Fig. 7. Illustration of the on-chip quantization and layout conversion module. The quantization method can be found in Equation (1).

Since each `bfloat16` number takes 16 bits, the output of one DMRFY0 bank (RFYa or RFYb) should have $W \cdot 16$ bits in `bfloat16` mode. However, DMRFY0 also needs to store the `int8` matrix, and the $W$ columns of the PE array only need $W \cdot 8$ bits. Thanks to the combined MAC optimization introduced before, the bitwidth of the DMPU input in `int8` *MatMul* becomes $W \cdot 16$-bit, matching the bitwidth of one bank in DMRFY0. As for non-FPGA implementation, designers can also deploy such optimization with larger MUL and accumulator, to fit bitwidth in `bfloat16`, as well as benefits from higher throughput in *MatMul* operations.

In TATAA architecture, the DMB serves the function of storing results temporarily before they are returned to the external memory. Each DMPU has a corresponding DMB in the bottom output direction. Importantly, DMBs have varying execution procedures in *MatMul* and `bfloat16` modes. In *MatMul* mode, all DMPUs form a single systolic array, leading to inactivity in the DMBs linked to DMPU 0 through DMPU 6, with only the last DMPU 7 receiving *MatMul* intermediate results. In contrast, in `bfloat16` mode, all DMPUs with $W$ columns (essentially, $W$ FPUs) function independently following a SIMD approach, necessitating all DMBs to store `bfloat16` vector results. A MUX also dictates the data path between the two modes, as illustrated in Figure 6(c). In *MatMul* mode, both the input **Y** and output **Z** traverse all DMPUs, with the bottom DMPU receiving data transmitted from the top DMPU as determined by the MUX. Conversely, in `bfloat16` mode, each DMPU obtains its input from RFY selected by the MUX.

### 4.6 On-Chip Quantization and Layout Conversion

Before writing the calculated `int8` matrix or `bfloat16` vector results back to external memory, the quantization unit dynamically quantizes the activations according to the current configuration, as shown in Figure 7. TATAA architecture supports the switch of data formats between `int8` and `bfloat16`, with four types of configuration in the quantization unit. All conversions here can be handled on the basis of Equation (1) with pre-loaded floating-point scaling factors. Moreover, if the subsequent workload cannot be directly deployed on the current data layout, TATAA manages the on-chip layout conversion. For instance, the QK-MUL layer requires the matrix **K** from the previous layer to be transposed to match the expected layout. Additionally, the `bfloat16` mode utilizes a vector-based layout, which differs from the typical matrix-based workloads. This difference necessitates hardware support for a row-by-row storage scheme to efficiently handle both `int8` → `bfloat16` and `bfloat16` → `int8` conversions. The only hardware overhead is to transpose the submatrix from DMPU, since all other conversions can be done by delicately controlling the write-back addresses, and the transpose module can be implemented using a simple register array with dual-direction ports.

Table 2. TATAA ISA

| Instruction Type | Description |
|---|---|
| CONFIG | Set up static parameters (e.g., scaling factors, constants) |
| LOAD.M | Load a matrix from memory |
| LOAD.V | Load a vector from memory |
| MATMUL | Execute MatMul $\mathbf{Z} = \mathbf{XY}$ |
| MUL.V | Execute *fpmul* of two vectors |
| ADD.V | Execute *fpadd* of two vectors |
| APP.V | Execute *fpapp* of one vector |
| STORE.M | Store a matrix (executed results) to memory |
| STORE.V | Store a vector (executed results) to memory |

We also give more details to explain how the data layout converts between different *MatMul* kernels, according to the MatMul dataflow in the proposed systolic array, as shown in the right part of Figure 7. The horizontal input (**X** matrix) of systolic array should be transposed due to the column-major streaming flow. Therefore, we map the static weights of transformer models into the **X** side, so that all the static weights can be transposed and the final data layout matches requirement, before the inference begins. For the vertical (input **Y** and output **Z** matrix) direction, the data layout keeps the same during *MatMul* runtime, so the activations are mapped into **Y** and **Z** to reduce extra layout conversion. A special case in transformer model is self-attention, in which the two matrices are both activations (i.e., Q, K, and V). Hence, when the next computation kernel is QK-MUL or SV-MUL, the TATAA architecture needs to transpose the output matrix, as well as separates the whole matrix data into multi-head layout. All the layout conversion operations are processed on-chip.

## 5 Compilation

Before introducing the proposed TATAA compilation framework, we define the terminology related to layers, nodes, and operations. A layer is a concept at the model level, with its definition detailed in Figure 1. The nodes operate at the graph level and are derived from a specific transformer model. For example, the non-linear SoftMax function can be broken down into a sequence of sub-functions, such as exponentiation, summation, and division, that become nodes in the computational graph. These nodes can be amalgamated or subdivided into additional nodes. In *MatMul*, a node with a large *MatMul* size can be divided into smaller tiled *MatMul* to better align with hardware structures. Operations reflect a hardware-level concept derived from nodes, implemented in the TATAA architecture, as discussed in Section 4.

### 5.1 ISA

To better decouple hardware and software, we have developed a customized ISA. Our software system can map linear operations in `int8` and non-linear operations with a high-precision approximation in `bfloat16`. Table 2 presents the simple ISA design in TATAA. In an ISA-level perspective, the controller is able to detect data dependencies and exploit **instruction-level parallelism (ILP)** to improve throughput performance. As an example, the double buffer optimization allows the parallel execution of the LOAD.M and MATMUL instructions. Furthermore, the previously mentioned data layout conversion with specific write-back addresses is incorporated into the STORE.M and STORE.V instructions, offering sufficient flexibility and comprehensive support for inference runtime. After compilation, all the run-time instructions are stored in the external memory, and

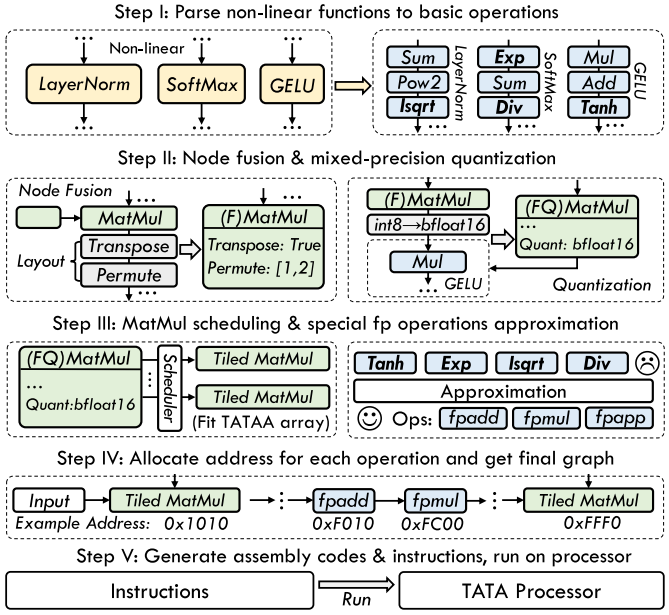


Fig. 8. Top-down workflow of TATAA compiler. Note that TATAA supports various non-linear functions. The depicted LayerNorm, SoftMax, and GELU are used as examples.

TATAA accelerator acts as a processor that fetches instructions from external memory, removing the requirement overhead of host-based scheduling.

### 5.2 End-to-End Transformer Mapping

Figure 8 illustrates the top-down compilation process from an input transformer-based model to the TATAA hardware runtime. The compilation framework first parses and converts non-linear functions into a series of basic operations (e.g., summation, squaring, multiplication) by examining the computation graph. For example, the LayerNorm function in Equation (2) is parsed to summation (sum up vectors in-between), power of 2 (calculate $\mathbf{x}^2$ for variation), division (calculate $E[\mathbf{x}]$ based on summation), and so on, as a series of operations. This parsing process has been mature in existing machine learning frameworks like ONNX [64].

Next, the compiler applies node fusion and mixed-precision quantization by integrating data layout conversion and quantization into the previous node, because the hardware supports on-chip quantization and layout conversion at runtime. With this intermediate representation, the compiler then schedules linear *MatMul* operations into a sequence of tiled *MatMul* operations, with each tile conforming to the size of the TATAA systolic array. Currently, the basic operations of non-linear functions are approximated and compiled into TATAA-supported operations (i.e., *fpmul*, *fpadd*, *fpapp*). Details of how to approximate these operations are shown in Algorithm 2. Upon completing these conversions, the compiler can analyze `bfloat16` workloads and vectorize them for SIMD-like instructions MUL.V and ADD.V as shown in Table 2. Finally, the compiler assigns addresses to each atomic operation for the hardware runtime and generates binary instructions for the TATAA processor.

**Algorithm 2:** Approximation Examples for Non-Linear Functions

**Input:** Input activation $x$
**Output:** Exponent value of $x$, $exp\_x$
1: $exp\_x = 2^{\lfloor (x/\ln 2) \rfloor}$ ▷ $2^{\lfloor \cdot \rfloor}$ is fused into output quantization process by a small LUT
**Output:** Inverse square root of $x$, $isqrt\_x$
2: $y = \texttt{0x5f37} - (short(x) >> 1)$ ▷ Similar to **Algorithm 1**
3: $isqrt\_x = 1.5y - 0.5x \cdot y^3$
**Output:** Padé approximation of tanh $(x)$
4: $tanh\_x = clamp(\frac{27x+x^3}{27+9x^2}, min = -1, max = 1)$

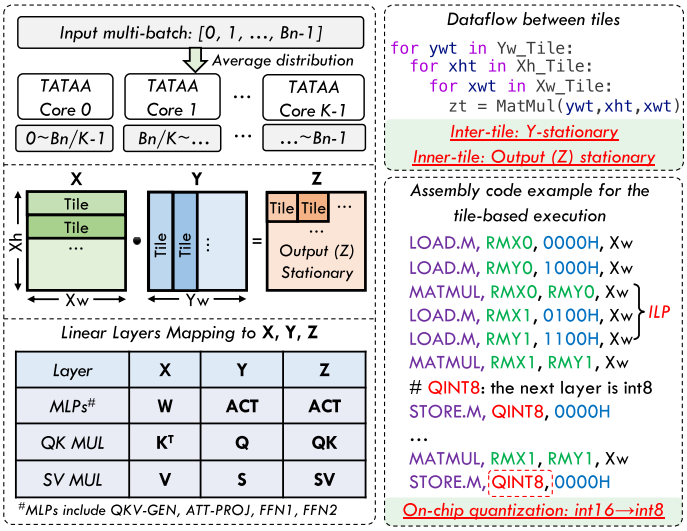


Fig. 9. Linear *MatMul* scheduling in TATAA compilation.

*5.2.1 MatMul Schedule.* The scheduler first analyzes the shape of the output matrix and distributes it evenly across batches to avoid data dependency between parallel cores. As activations are independent across various batches, the *MatMul* scheduler and the non-linear functions compiler can concentrate solely on the batches of a single core, incrementally updating the starting address to determine the activation addresses for other cores. Each TATAA core employs *MatMul* based on an output-stationary dataflow with a fixed output tile size, $W$ by $4N$. Based on this tiling, each output tile corresponds to a tile of matrices **X** and **Y**. After determining the addresses for **X**, **Y**, and **Z**, the scheduler generates a series of instructions LOAD.M, MATMUL, and STORE.M as assembly codes for run-time inference and applies double buffer optimization by reordering the three types of instruction, similar to the ILP strategy. Figure 9 illustrates the scheduler in terms of dataflow design and an example of ILP optimization using assembly codes. In this example, the `MATMUL,RMX0,RMY0,Xw`, `LOAD.M,RMX1,0100H,Xw` and `LOAD.M,RMY1M,1100H,Xw` can be executed in parallel since (1) there is no RF index conflict; (2) we design two I/O ports for loading. Besides, the scheduler needs to decide how to map the two operands of *MatMul* into **X** and **Y**, since the data layout of the two input ports in systolic array are different. For normal MLP, we map the weights and activations into **X** and **Y**, respectively, while for QK-MUL and SV-MUL without weights, we use another mapping scheme, as the layout conversion between **X** and **Y** can be done on hardware.

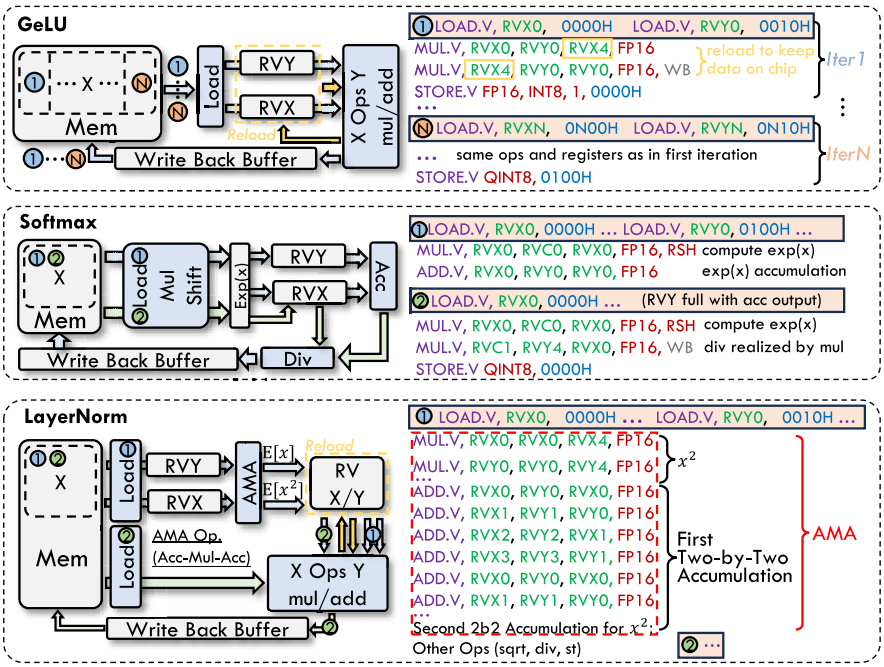


Fig. 10. Non-linear functions compilation in TATAA. We only present three examples which are commonly applied in most transformer models.

In terms of the *MatMul* tiling approach within a TATAA core (i.e., the entire workload is distributed across multiple cores in the batch dimension), we aim to reduce the schedule complexity by maintaining the $\mathbf{X_w}$ (number of columns in $\mathbf{X}$) and the matched $\mathbf{Y_h}$ (number of rows in $\mathbf{Y}$) within a single tile for as long as practical. For instance, the scheduler first determines the maximum matrix buffer depth ($D_{mat}$ as illustrated in Figure 6) and averages the tiling of $\mathbf{X_w}$ if it exceeds $D_{mat}$. Otherwise, $\mathbf{X_w}$ remains in one tile to enhance input stream length for greater throughput. Additionally, based on the systolic array dataflow, the tile size in the $\mathbf{X_h}$ and $\mathbf{Y_w}$ dimensions must align with the row and column count of the systolic array (i.e., $4N$ and $W$), with suitable padding in the input matrix. Once the tile size is set, the MatMul dataflow can be generated in a straightforward way by accessing the for-loop in all matrix dimensions. Such a fixed scheduling method may not achieve the theoretical optimized performance due to the huge design space, but it is sufficient in this work to evaluate our hardware architecture. We may explore more opportunities in the scheduling steps, by referring to the existing design space exploration frameworks like DNNExplorer [65] and AGNA [66].

*5.2.2 Non-Linear Functions Optimization.* The fundamental process of mapping non-linear functions to TATAA hardware runtime is illustrated in Figure 10. To optimize the performance of non-linear operation execution, TATAA focuses on reducing memory access and maintaining computation on the chip by consistently reloading computation results into local registers. As shown in the yellow dashed box with the reload operation in Figure 10, when two vectors perform a MUL.V or ADD.V operation, the result is stored (reloaded) in one of the X or Y registers. By consistently performing this reloading and computation process, most computational operations are executed together without requiring additional memory access. All operations that do not involve

memory access employ this on-chip computation method to maximize execution throughput. Since this on-chip computation necessitates loading as much data as possible into the RFs before executing computation instructions, load instructions are assembled at the beginning of each non-linear function. To mitigate sluggish memory access times for these load instructions, the outstanding transaction features of AXI are utilized to minimize the total load duration.

Additionally, we emphasize two types of node-level optimization for non-linear functions, as depicted in Figure 10, with the aim of significantly reducing memory I/O costs while maximizing computational efficiency. We optimize the compiler to reuse input $\mathbf{x}$ vectors in RFs by rearranging the computation nodes within the graph. This method prevents redundant loading of $\mathbf{x}$ onto the hardware. Given that the LayerNorm function involves both variance and mean calculations, as shown in Figure 10, **accumulate-multiply-accumulate (AMA)** facilitates the computation of $E[\mathbf{x}]$, $\mathbf{x}^2$, and $E[\mathbf{x}^2]$ by splitting both the X and Y registers into two groups. The input vector $\mathbf{x}$ is first loaded into the first group of the X and Y registers and then multiplied by itself to store the result $\mathbf{x}^2$ into the second group of registers. As demonstrated in the assembly code, after loading the input vectors, the AMA process initiates with a series of MUL.V instructions to compute $x^2$. The initial and subsequent two-by-two accumulation operations are executed without encountering data hazards, thus avoiding additional delays between computation instructions. In the first accumulation example, the values in the X registers 0 to 3 are added to the corresponding values in the Y registers. The sums are then stored back into registers 0 to 1 of both X and Y registers in a crosswise manner. This approach effectively reduces the total number of registers holding partial sums by half. The values in registers 0 to 1 are further consolidated into another partial sum, which is reloaded into register 0 in both X and Y registers. This partial sum is subsequently added together to produce the final accumulation result. Since the values from the first and second accumulation processes are stored in separate sections of the X and Y RFs, the second two-by-two accumulation can occur concurrently with the first accumulation process. This approach ensures that the maximal amount of input vector data that both RFs can hold is loaded only once from memory, leaving a substantial portion of the computation to be performed on-chip, thereby maximizing performance. Furthermore, to mitigate potential data hazards during this consistent computation, a two-by-two accumulation method is employed, which adds every two lines of input vectors and stores the results crosswise into the X and Y registers. Such an optimization compilation works for other normalization functions, e.g., RMSNorm as well.

Furthermore, it is important to note that the GELU function maintains a consistent tensor shape across all nodes, enabling segmentation of all nodes into uniform tile shapes and allowing sequential execution of these tiles from start to finish. As illustrated in Figure 10, each iteration corresponds to a tile, and within every iteration, a much longer sequence of computation instructions is executed between a load and a store instruction. The yellow arrows indicate this extended sequence of computations. Although the number of memory accesses increases with the number of tiles, the memory overhead remains minimal compared to the lengthy computation sequence. Upon completion of each iteration, the results are written back to the address from which the input vector was initially read. Simultaneously, all registers used in the current iteration are cleared and ready to receive the next tile for the subsequent iteration. This multi-iterative method is optimized for GELU scheduling to achieve two main objectives: (1) avoiding intermediate I/O communication with external memory and (2) accommodating the limited RF space. Given that TATAA employs a layer-by-layer execution method, these I/O optimizations are crucial for improving throughput efficiency. Note that as long as the activation functions have the same tensor shape patterns across all nodes (e.g., SiLU), this tile-based compilation can be applied.

## 6 Evaluation

### 6.1 Experimental Setup

We implemented and prototyped TATAA on Alveo U280 FPGA platform using Verilog HDL and Vitis 2021.1 tools under 225 MHz frequency, to measure resource utilization, power consumption, and end-to-end run-time throughput. The Alveo U280 concludes 1.08M LUTs, 4.5 MB of on-chip BRAM, 30 MB of on-chip URAM, and 9,024 DSP slices. Note that we implement the buffer based on BRAM only. The hyper-parameters mentioned in Figure 2 are set as $K = 8$, $N = 8$, $W = 16$, with a corresponding 32 by 32 systolic array and 128-lane SIMD FPUs in each core. In Alveo U280, we set up 16 AXI channels for the 8 TATAA cores and each AXI channel has 256-bit memory bitwidth.

We have chosen a range of transformer models to evaluate the accuracy of quantization and their run-time performance in tasks such as image classification, text classification, and text generation. The selected models are listed below and their feature dimensions are shown in Table 3.

- —For ViT, we select DeiT [24] and Swin Transformer (Swin) [25] with ImageNet-1k [67] dataset for the image classification.
- —BERT [26], as a widely used language model, is also selected for evaluation based on the GLUE benchmark including different tasks [68].
- —We also evaluate other popular language models, GPT-2 [69] and OPT [27], for the text generation task with LAMBADA [70] and WikiText-2 datasets.
- —To evaluate the general support of TATAA for non-linear functions in transformer models, we also select state-of-the-art Llama [12] and ChatGLM2 [71] where some extra functions like RMSNorm [4], SwiGLU [7] and SiLU [6] are deployed. Note that the two models are not evaluated end-to-end, as we only tested the non-linear functions part on hardware.

### 6.2 Model Accuracy

First, we evaluate the approximation techniques outlined in Section 5.2.2 to demonstrate that our `bfloat16` implementation of non-linear functions is precise and can therefore be used for complete model inference. Figure 11 shows the errors for the inversed square root, pade *tanh*, as well as the function-level GELU approximation. We did not evaluate the power of 2 approximation as it has been widely utilized in SoftMax hardware and demonstrated to produce negligible errors [38]. In the given input range, the overall RMSEs of approximations are $1.90 \times 10^{-3}$, $1.52 \times 10^{-2}$, and $1.97 \times 10^{-3}$ for the two methods and the GELU activation function, respectively, demonstrating our selected approximations for non-linear functions are sufficient.

Using the `int8` + `bfloat16` PTQ method, we evaluate several transformer models on a range of tasks, simulating model accuracy through PyTorch-based quantization codes. The calibration dataset is generated by randomly sampled a very small size of training set (16 $\sim$ 128 in our setups). Table 4 presents the inference performance for ViT,[2] BERT, and GPT-2 models, with classification and text generation tasks. The drop in accuracy among all evaluation tasks is negligible from 0.34% to 1.16%, demonstrating that the TATAA PTQ scheme is available for flexible transformer acceleration without the need for retraining overhead. As illustrated above, static PTQ is applied in TATAA, and in the current work we can deploy other existing PTQ approaches like SmoothQuant [35] and FQ-ViT [32], thanks to the general support in TATAA framework. With the on-chip quantization and layout conversion module, TATAA can efficiently deploy quantized MatMul and non-linear functions with appropriate data format and layout, as long as the framework gets static quantization scaling factors, according to Equation (1).

[2]Three scales of DeiT and Swin Transformer, -T, -S, -B refer to Tiny, Small and Base, respectively.

Table 3. Selected Transformer Models or Non-Linear Functions in the Experiments

| Model | Type | # Blocks | # Heads | Hidden Size | MLP Size | Non-Linear Functions |
|---|---|---|---|---|---|---|
| Deit-S | Encoder | 12 | 6 | 384 | 1,536 | |
| Deit-B | Encoder | 12 | 12 | 768 | 3,072 | SoftMax |
| Swin-T | Encoder | {2, 2, 6, 2}[a] | {3, 6, 12, 24}[a] | {96, 192, 384, 768}[a] | {$56^2$, $28^2$,$14^2$,$7^2$}[a] | LayerNorm |
| BERT | Encoder | 12 | 12 | 768 | 3,072 | GELU |
| GPT2 | Decoder | 24 | 16 | 1,024 | 4,096 | |
| OPT-1.3B[b] | Decoder | 24 | 16 | 2,048 | 8,192 | SoftMax, LayerNorm, ReLU |
| Llama-7B[b] | Decoder | 32 | 32 | 4,096 | 11,008 | SoftMax, RMSNorm, SwiGLU |
| ChatGLM2[b] | Decoder | 28 | 32 | 4,096 | 13,696 | SoftMax, RMSNorm, SiLU |

[a]{...} shows dimension variance of each stage in a Swin-T [25].
[b]These large language models are not evaluated on hardware runtime. We only select them for various non-linear functions test.

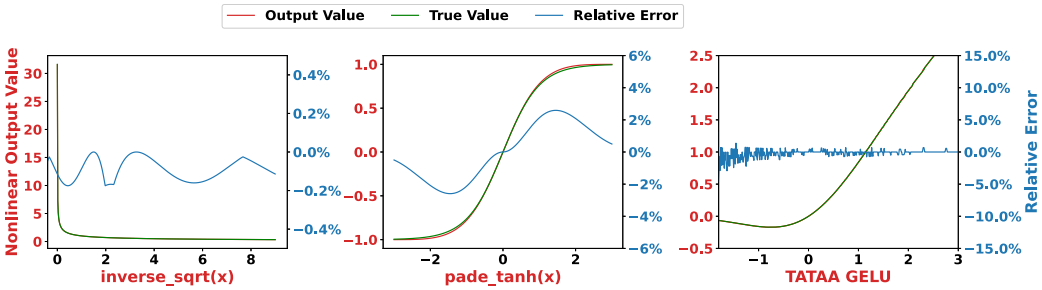


Fig. 11. The non-linear function precision measures between our approximated and PyTorch's built-in functions. We selected two approximated sub-operations (inverse square root and pade *tanh*), and a function-level GELU function for error evaluation.

Table 4. Quantization Evaluation for Various Transformer Models Based on TATAA Setups with `int8` + `bfloat16`

| Method | PTQ. Format | ViT Classification Accuracy (%) | | | | | | BERT on GLUE (%) | | | GPT-2 Medium | OPT-1.3B[b] | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | DeiT-T | DeiT-S | DeiT-B | Swin-T | Swin-S | Swin-B | QQP | SST-2 | MRPC | WikiText2 PPL | WikiText2 PPL | Lambada Acc(%) |
| Baseline | `fp32` | 72.14 | 79.83 | 81.79 | 80.99 | 83.21 | 83.60 | 90.98 | 92.90 | 86.03 | 15.94 | 14.62 | 75.41 |
| TATAA | `int8+ bfloat16` | 70.98 (−1.16) | 79.35 (−0.48) | 81.65 (−0.34) | 79.98 (−1.01) | 82.44 (−0.77) | 82.70 (−0.90) | 90.15 (−0.83) | 92.32 (−0.58) | 85.54 (−0.49) | 16.41 (+0.47) | 15.18 (+0.56) | 74.96 (−0.45) |

[a]Baseline models with pre-trained `fp32` parameters are loaded from PyTorch or Hugging Face model hub.
[b]SmoothQuant [35] is applied for OPT-1.3B quantization.

### 6.3 Hardware Utilization

Table 5 presents the hardware utilization and the corresponding breakdown in FPGA based on the selected configuration. It can be concluded that the DMPUs dominate the resource cost, and other overhead units such as quantization and transpose are relatively small. In detail, the proposed DMPUs cost 86.8% LUTs, 85.4% FFs, and 94.1% DSPs for FPGA resources throughout the design. Due to the SIMD approach in the `bfloat16` mode, the controller overhead is minimal because the dataflow is shared between all FPUs.

We provide a detailed comparison of different schemes, separating integer linear units from the overhead needed to support non-linear operations, and the experimental setups are illustrated in Figure 12, named as *SA+FPU Indiv.* design. For TATAA and *SA+FPU Indiv.* setup, Figure 13 presents

Table 5. Hardware Utilization of the Proposed TATAA Processing Core and the Breakdown of Resources

| Components | FPGA Utilization (Breakdown %) | | | |
|---|---|---|---|---|
| | LUT | FF | BRAM | DSP |
| DMPUs | 60,117 (86.8%) | 85,035 (85.4%) | 0 (0.0%) | 512 (94.1%) |
| RFs | 2,240 (3.2%) | 4,333 (4.4%) | 78.5 (54.0%) | 0 (0.0%) |
| DMB | 280 (0.4%) | 224 (0.2%) | 60 (41.2%) | 0 (0.0%) |
| Quantization layout convert | 6,558 (9.5%) | 9,899 (9.9%) | 7 (4.8%) | 32 (5.9%) |
| Controller misc | 87 (0.1%) | 80 (0.1%) | 0 (0.0%) | 0 (0.0%) |
| One TATAA core total | 69,282 | 99,571 | 145.5 | 544 |

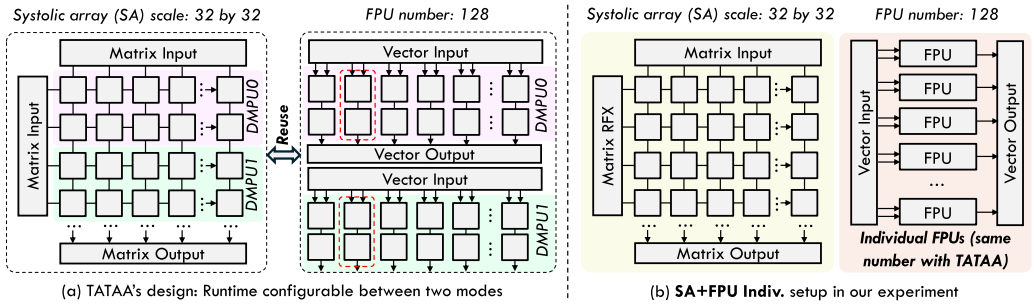


Fig. 12. Experimental setup of comparing TATAA (reusing same hardware for *MatMul* and `bfloat16` operations) and traditional implementation (individual systolic array and FPU, *SA+FPU Indiv.* in abbreviation). The systolic array scale and the number of FPUs are the same for fair comparison. Note that the systolic array scale and FPU number in this figure are based on one single TATAA core.

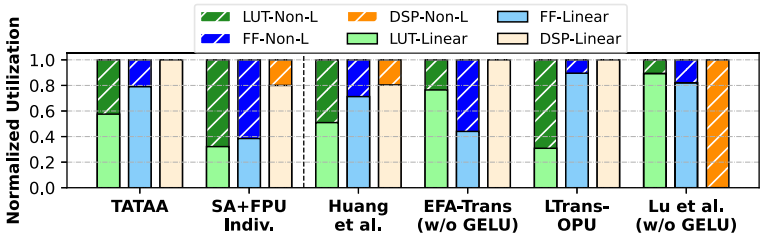


Fig. 13. Normalized hardware utilization on FPGA for the proposed TATAA and other related works, in terms of linear *MatMul* units (-L) and non-linear functions overhead (-NL) based on three types of resources (LUT, FF, DSP).

the normalized utilization in terms of hardware units for both linear layers (*MatMul*) and non-linear functions (indicated with shadows). We compare our TATAA architecture with a design that utilizes individual integer systolic arrays and FPUs without reuse, on the same scales (32 by 32 array and 128 lane SIMD FPUs in one TATAA core), as illustrated on the left side of Figure 13 (*SA+FPU Indiv.*). Note that all non-linear functions can be compiled into separated FPUs, and the SIMD mode is similar to the strategy commonly used in previous studies [17]. Such a comparison indicates that the reuse scheme drastically reduces hardware costs for non-linear functions. Additionally, we present other related FPGA-based accelerators utilization of linear versus non-linear functions based on their reported results [21, 53, 54, 57, 72]. Our TATAA architecture exhibits comparable overhead across three resource types, with only 10.5% FFs, and no DSPs overhead especially. As

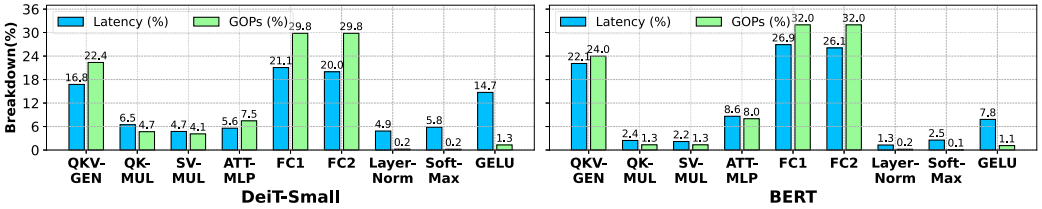


Fig. 14. Layerwise latency and computation workload size breakdown in DeiT-S and BERT, during TATAA runtime.

exceptions, EFA-Trans [53] reports linear operation units including SoftMax, and in the work by Lu et al. [72], where LUTs are employed for linear computations and the DSP overhead for non-linear operations reaches 100%.

### 6.4 TATAA Run-Time Analysis

We choose **DeiT-Small (DeiT-S)** with batch size 16 and BERT with sequence length 128 and batch size 32 to measure layer latency with its workload size Figure 14. We emphasize that for the end-to-end evaluation of models (including vision models and BERT as shown in Table 3), latency is measured from the start of the first transformer block to the conclusion of the final MLP layer, excluding the initial embedding layer of the BERT model. The linear *MatMul* layers (such as QKV-GEN, QK-MUL, SV-MUL) are the primary contributors to the transformer workload when measured in terms of GOP (giga operations) or GFLOP (giga floating-point operations), and therefore heavily influence total latency. Among these linear layers, QK-MUL and SV-MUL are slightly less efficient, exhibiting a smaller workload-latency ratio compared to other MLP layers such as QKV-GEN. This inefficiency is attributed to the fact that the multi-head attention mechanism reduces the matrix sizes in each *MatMul*, which in turn leads to less data reuse in the output-stationary dataflow. Such an analysis shows the optimization space in the future to dynamically optimize the different size of workloads. In addition, despite the non-linear functions having significantly smaller workloads compared to the linear *MatMul* layers, they nevertheless contribute significantly to latency, accounting for approximately 25% of the total end-to-end inference time. Hence, the implementation of non-linear functions is crucial for both model performance and hardware efficiency, as is also proved in some previous studies [2]. Our flexible and adaptable framework for compiling these functions offers an optimization potential for new transformer models that utilize various non-linear functions.

Figure 15 presents the non-linear functions throughput based on `bfloat16` basic operations (GFLOPS), over several selected transformer models. Since the proposed TATAA architecture supports full pipeline between basic operations (*fpmul*, *fpadd*, and *fpapp*), the theoretical maximum throughput considering the computation resources (i.e., FPU number in TATAA architecture) can be calculated by Equation (6):

$$GFLOPS_{theo} = K \cdot N \cdot W \cdot freq, \tag{6}$$

where the $K \cdot N \cdot W$ refer to the number of FPUs (128 in one core and $128 \times 8 = 1024$ in the whole multi-core design). In our evaluation setup, the throughput $GFLOPS_{theo}$ is thus $1024 \times 225$ MHz = 230.40 GFLOPS. Among all the test functions, our TATAA can reach to maximum 189.45 GFLOPS in GELU function of BERT model. This is because the memory-bound nature of these `bfloat16`-based non-linear functions. Still, our compilation framework can reach 82.2% maximum throughput and leave an optimization space for compiler in the future. For instance, the SoftMax function requires accessing external memory several times due to the data dependency as illustrated

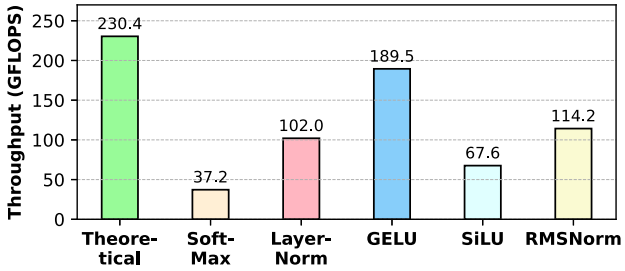


Fig. 15. Evaluation of the selected non-linear functions in various transformer models. The throughput (GFLOPS) is measured on hardware runtime. The SoftMax, LayerNorm, and GELU functions are based on BERT model, while the SiLU is used in ChatGLM and RMSNorm is based on Llama-7B.

in the compilation steps Figure 10, thus causing lower throughput as the compiled operations are highly memory-bound. The users can deploy more efficient SoftMax schemes to improve it, like Flash-Attention [73] which significantly reduces the memory I/O.

We further compare the latency of non-linear functions in TATAA with various related studies that have documented their evaluations of non-linear functions, as shown in Table 6. Since the token lengths and model scales in these acceleration works are different, we normalize the latency to cycles per element, as the same setup in [17]. The proposed TATAA achieves significantly lower latency in terms of SoftMax and LayerNorm function, because the transformable architecture is able to utilize all the processing units for non-linear functions, boosting the theoretical floating-point operations throughput. In terms of GELU, since TATAA only applies naive approximation to demonstrate our flexibility, the final latency is not good as Chen et al. [51]. But in general, our total latency for non-linear functions still outperforms Chen et al. by 4.25×, without any computational resources overhead. In addition, we compare the proposed implementation with previous works in terms of throughput and area efficiency (throughput/DSP blocks), as shown in Figure 16. The results show that TATAA also achieves higher throughput and comparable area efficiency in DSPs (reach 19.6× and 9.1× higher than the baseline NPE-1024 design [17]), while other works may still cost more LUTs or FFs for non-linear functions. Compared to our previous study [74] (Wu et al.) which fuses `fp32` and **8-bit BFP (`bfp8`)** with similar motivations, TATAA achieves higher throughput improvement since we are targeting a cheaper data format (`int8` and `bfloat16`) and a more efficient hardware reuse scheme. All in all, the key benefit of TATAA is the potential for further optimization of efficiency through compilation of emerging non-linear functions, a feature that is absent in those accelerators with fixed and specific units.

### 6.5 Resource Efficiency

As one of the key contributions in TATAA, we reuse the same integer hardware units for non-linear functions deployment, saving the hardware overhead and thus improving the resource/area efficiency. First, we evaluate end-to-end throughput (GOPS) of TATAA and normalize the throughput into resource to obtain resource efficiency, in terms of both LUT and DSP. For comparison, we selected related FPGA-based accelerators for transformers, calculating their efficiency based on the utilization results. The selected models have various data format setups, e.g., DFX [56] implements `fp16` for all operations while Auto-ViT-Acc [44] only targets linear *MatMul* in integer. To benchmark our transformable architecture, we also implement the *SA+FPU Indiv.* design, the opposite setup compared to TATAA, as shown in Figure 12.

Figure 17 presents the resource efficiency results based on the selected implementations, in terms of end-to-end GOPS per DSP (GOPS/DSP) and GOPS per kilo-LUT (GOPS/kLUT). Compared with

Table 6. Normalized Non-Linear Functions Latency for TATAA and Related Works for Non-Linear Functions Implementation

| Non-Linear Implementation | Latency (Cycles per Element) | | | | DSP Overhead |
|---|---|---|---|---|---|
| | SoftMax | LayerNorm | GELU | Total | |
| NPE-1024 [17] (fxp vector unit) | 2.53 | 9.94 | 0.75 | 13.22 | 1.58% |
| Huang et al. [21] (fxp special unit) | 1.59 | 2.40 | - | - | 18.5% |
| Chen et al. [51] (fp special unit) | 5.14 | 0.66 | 0.13 | 5.92 | 21.0% |
| TATAA (bfloat16 transformable architecture) | 0.50 | 0.51 | 0.39 | 1.39 (4.25×) | 0% |

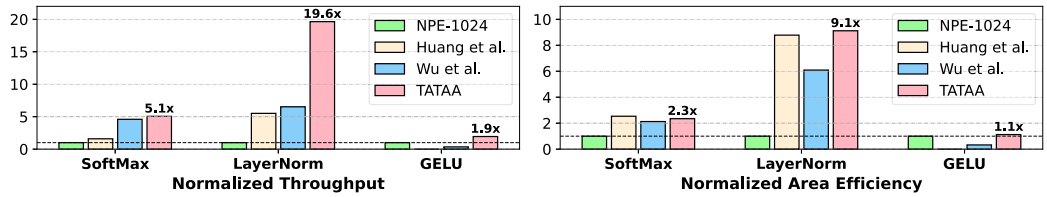


Fig. 16. Comparison of normalized throughput and normalized area efficiency between TATAA and related works. The area efficiency is measured by DSP utilization.

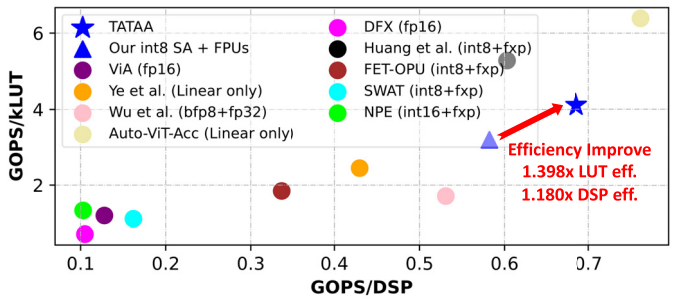


Fig. 17. Resource efficiency in terms of GOPS/DSP and GOPS/kLUT, comparing TATAA with several FPGA-based acceleration frameworks.

our own baseline (*SA+FPU Indiv.*), TATAA achieves 1.28× LUT efficiency and 1.18× DSP efficiency, due to the hardware reusing scheme in the proposed transformable architecture. Compared to other related works, our `int8` + `bfloat16` approach is still compatible with smaller workload in linear layers. Although some previous works demonstrate superior resource efficiency in terms of LUT or DSP, it is important to mention that they either do not support full inference on hardware (Auto-ViT-Acc [44]) or employ more aggressive approximations of non-linear functions using fixed-point format (Huang et al. [21]), lacking the PTQ retrain-free benefit that is implemented in our TATAA. Besides, Wu et. al [74] proposed to fusing `bfp8` and `fp32` formats in the same architecture which is similar to TATAA. However, due to the higher hardware cost for block-wise operations and `fp32` complexity, our TATAA achieves around 2.25× and 1.31× higher efficiency for GOPS/kLUT and GOPS/DSP with similar model accuracy.

Table 7. Hardware Performance Comparison with Relative FPGA-Based Accelerators for Transformer Models

| Work | Data Formats[a] | End2end Support | FPGA Platform | FPGA Utilization | | | | Freq. (MHz) | Power (W) | Eval. Models | Throughput Inf./sec[b] | Throughput (GOPS) | DSP Efficiency |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | LUT(k) | FF(k) | BRAM | DSP | | | | | | |
| Auto-ViT-Acc [44] | fxp, fp32 | No | ZCU102 | 185.0 | - | - | 1,552 | 150 | 9.6 | DeiT-S<br>DeiT-B | 99.7<br>34.0 | 907.8<br>1,181.5 | 0.585<br>0.761 |
| Huang et al. [21] | int8, int8 | Yes | ZCU102 | 144.5 | 168.0 | 648 | 1,268 | 300 | 29.6 | ViT-S<br>ViT-T | 89.7<br>245.3 | 762.7<br>616.1 | 0.601<br>0.486 |
| HPTA[75] | int8, int8 | Yes | ZCU102 | 209.9 | 368.4 | 345 | 2,307 | 200 | 20.0 | BERT<br>Swin-T | 81.9<br>148.8 | -<br>- | 0.035[c]<br>0.065[c] |
| NPE [17] | int16, fxp | Yes | VCU118 | 192.4 | 351.1 | 369 | 2,020 | 200 | 20.0 | BERT | 36.8 | - | 0.018[c] |
| FTRANS [48] | fp16, fp32 | Yes | VCU118 | 451.1 | 506.6 | - | 6,531 | - | - | RoBERTa | 94.25 | - | 0.014 |
| ViA [20] | fp16, fp16 | Yes | Alveo U50 | 258.0 | 257.0 | 1,022 | 2,420 | 300 | 39.0 | Swin-T | - | 309.6 | 0.128 |
| SWAT [54] | int8, fxp | Yes | Alveo U50 | 271.0 | - | 609.5 | 1,863 | 200 | 14.4 | Swin-T | - | 301.9 | 0.162 |
| ME-ViT[55] | int8, fxp | Yes | Alveo U200 | 192.0 | 132.0 | 288 | 1,024 | 300 | 9.3 | DeiT-B<br>DeiT-S | 23.9<br>41.7 | -<br>- | 0.0233[c]<br>0.0407[c] |
| DFX [56] | fp16, fp16 | Yes | Alveo U280 | 520.0 | 1,107.0 | 1,192 | 3,533 | 200 | - | GPT-2 | 0.361 | 185.6 | 0.0001[c] |
| Ye et al.[58] | int8, fxp | No | Alveo U250 | 736.0 | - | 1,781 | 4,189 | 300 | - | - | - | 1,800.0 | 0.430 |
| FET-OPU[22] | int8, fxp | Yes | Alveo U280 | 886.8 | 716.6 | 1,357 | 4,864 | 200 | 7.4 | DeiT-B<br>BERT<br>Swin-T | 71.8<br>146.6<br>124.1 | 1,264.6<br>1,635.8<br>1,070.1 | 0.0148[c]<br>0.0301[c]<br>0.0256[c] |
| TATAA | int8, bfloat16 | Yes | Alveo U280 | 724.9 | 1,154.9 | 1,472 | 4,352 | 225 | 10.8 | DeiT-S | 218.6 | 2,836.2[d] | 0.626<br>0.0502[c] |
| | | | | | | | | | | DeiT-B | 67.6 | 2,796.5 | 0.643<br>0.0156[c] |
| | | | | | | | | | | BERT | 116.8 | 2,935.2 | 0.674<br>0.0269[c] |
| | | | | | | | | | | Swin-T | 179.7 | 2,512.3 | 0.685<br>0.0587[c] |
| | | | | | | | | | | GPT-2 | 7.9 | 2,579.4 | 0.593<br>0.0018[c] |

[a]Data formats for linear *MatMul* (the former one) and non-linear functions (the latter one). `fxp` refers to fixed-point numbers.
[b]Inference per second (Inf./sec) measures how many end-to-end images or sequences can be processed through hardware in one second.
[c]Results with inference/sec/DSP are marked with symbols †, while those based on GOPS/DSP are indicated separately. We provide both results for TATAA.
[d]We clarify that in our work, the total operation is obtained by doubling the MAC operation. Some previous work may directly report MACs as throughput.

### 6.6 Systematic Comparison with Related Studies

We summarize and compare other related FPGA-based transformer accelerators with our TATAA FPGA prototype with respect to throughput and resource efficiency, as illustrated in Table 7. In our setup, the ViT models (DeiT, Swin) are evaluated on ImageNet with a batch size of 16, the BERT sequence length is fixed at 128, and the GPT-2 with 345M parameters is under 512 sequence length. We only measure the pre-fill stage for the GPT-2 inference. For other related works, we scaled their results to match ours if they set different sequence lengths or batch sizes, for a fair comparison. Our TATAA achieves up to 2,836.2 GOPS with an end-to-end acceleration rate of maximum 218.6 **inference per second (Inf./sec)** for vision models, while reaching 2,579.4–2,935.2 GOPS in language models. We also report throughput efficiency by normalizing the total throughput with computational resources (DSPs in FPGA), offering results in terms of GOPS/DSP or Inf./sec/DSP to facilitate comparisons across all related works. Compared to other accelerators, TATAA obtains higher resource efficiency by factors ranging from 1.13× to 2.29× in terms of Inf./sec/DSP among all small-scale models, except in certain cases where end-to-end support is lacking (e.g., Auto-ViT-Acc [44]),

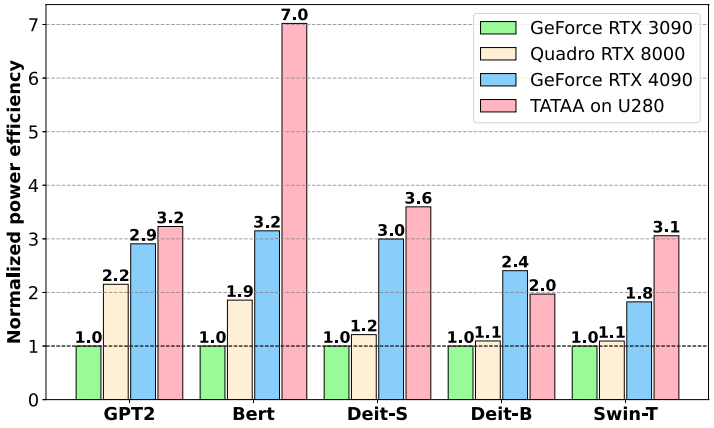


Fig. 18. Normalized power efficiency (Inf./sec/W) comparison between TATAA and GPU implementations.

or in some dedicated optimization work (e.g., FET-OPU [22] achieves ~ 0.003 higher efficiency than TATAA). Nevertheless, the most important benefit of TATAA is to support general and flexible non-linear functions, unlike other works deploying specific support for these functions.

We acknowledge that there are other leading acceleration frameworks that surpass TATAA in terms of throughput and area efficiency. For instance, the work by Chen et al. [51] introduced a complete pipeline model using a spatial accelerator on FPGAs, with each block layer allocated to a separate hardware module, maintaining adequate buffer capacities for the pipeline. This configuration significantly enhances throughput by minimizing memory I/O operations, but as transformer models expand, mapping and compiling such a comprehensive pipeline approach becomes increasingly complex. Our strategy, in contrast, focuses on developing a general-purpose and broadly applicable accelerator. SSR's proposal by Zhuang et al. [50] involves implementing a spatial and temporal hybrid design on Versal ACAP devices, where `MatMul` operations utilize the AI Engine of AMD FPGAs. Directly comparing their framework's efficiency with ours wouldn't be equitable. Nonetheless, the primary innovation in TATAA is presenting a novel methodology for reusing the same hardware across varied operations within transformer models. Thus, TATAA can operate independently of spatial or temporal architectures in transformer accelerators. There is significant potential to further refine pipeline strategies among different TATAA cores to facilitate the `MatMul` and `bfloat16` pipeline, and we intend to explore this potential in the future, inspired by these related works.

### 6.7 Power Efficiency versus GPUs

Power efficiency, as an important highlight for the TATAA to be deployed on U280, shows prominent features of FPGA over modern GPUs. We used Xilinx RunTime for hardware execution, and Vitis embedded power profile with Xilinx Board Utility for computing power measurements. NVIDIA system management interface (nvidia-smi) is used for measuring GPU power on Quadro RTX8000, GeForce RTX3090, and GeForce RTX4090.

We evaluate the internal power consumption of the TATAA framework, assessing the power efficiency (inferences per watt) for each model. Subsequently, we measured the model execution latency on various GPUs using consistent batch sizes and sequence lengths as in the TATAA framework, thereby computing the power efficiency (Inf./sec/W). As depicted in Figure 18, our TATAA FPGA surpasses the performance of the RTX3090 and RTX8000 across all models, demonstrating a 1.10× improvement over the RTX4090 in normalized power efficiency for GPT2. For smaller models, such as the small DeiT transformer, TATAA remains competitive, achieving comparable

power efficiency in end-to-end throughput. While TATAA shows a slight degradation in efficiency on Deit-B compared to the RTX4090, it significantly outperforms other devices on the BERT model (2.19× more than 4,090) when the sequence length of inputs is properly adapted to the dataflow requirements of the TATAA hardware. This emphasizes one of the critical advantages of deploying TATAA on FPGAs compared to GPUs, highlighting its potential for processing large models with superior power efficiency.

### 6.8 Potential Limitations of TATAA and Optimization Opportunities

We have thoroughly evaluated the proposed architecture based on hardware overhead, area/resource efficiency, run-time throughput analysis and optimization, as well as power efficiency which is significantly important for FPGA accelerators. Even though the results show a highly promising trend that reusing hardware resources can improve hardware efficiency, we have to admit that such a scheme in hardware architecture for MatMul and non-linear functions would have potentially worse absolute performance in some cases. For example, the SIMD mode which is also used in some previous works like NPE [17] may have lower efficiency than specialized units with dedicated pipeline, and the potential PE under-utilization due to the different arithmetic intensity between MatMul and floating-point non-linear operations, as well as the tensor shape and data layout mismatch between layers and kernels. Therefore, the responsibility of compilation shows up, and it should be optimized further to exploit such a general-purpose hardware architecture. Compared with specific optimized hardware units for non-linear functions which may have higher throughput, lower latency, or even better utilization rate, our TATAA design instead aim to support a wide range of non-linear functions since the transformer models are growing at a rapid speed. Our design choice of reusing the *MatMul* resources for non-linear functions in a SIMD way, not only saves hardware overhead but also leaves sufficient design space for compiler design. Again, the hardware accelerators design is always a tradeoff between generalization and performance. We will definitely explore more compilation optimization opportunities based on such a novel angle proposed in TATAA, to solve the challenges and limitations in current TATAA architecture.

## 7 Conclusions

In this work, we have presented TATAA, a programmable accelerators on FPGA for transformer models by using a novel transformable arithmetic architecture. Using TATAA, we demonstrate that both low-bitwidth integer (`int8`) and floating-point (`bfloat16`) operations can be implemented efficiently using the same underlying processing array hardware. By transforming the array from systolic mode for `int8` MatMul to SIMD-mode for vectorized `bfloat16` operations, we show that end-to-end acceleration of modern transformer models including both linear and non-linear functions can be achieved with state-of-the-art performance and efficiency. In the future, we plan to explore more general FPGA implementations of TATAA with more devices support (i.e., with or without HBM) and to enhance the flexibility of our compilation framework to accelerate future transformer models as they are being rapidly developed.

## Acknowledgments

We thank Mr. Haiqiao Hong and Dr. Ngai Wong who helped set up and evaluate the GPUs results in our work.

## References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need. In *Advances in Neural Information Processing Systems* . Retrieved from https://arxiv.org/abs/1706.03762

[2] Jacob R. Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. 2021. Softermax: Hardware/software co-design of an efficient softmax for transformers. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC '21)*. IEEE Press, 469–474. DOI: https://doi.org/10.1109/DAC18074.2021.9586134

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. arXiv:1607.06450. Retrieved from https://arxiv.org/abs/1607.06450

[4] Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, Vol. 32. Retrieved from https://openreview.net/references/pdf?id=S1qBAf6rr

[5] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (GELUs). arXiv:1606.08415. Retrieved from https://arxiv.org/abs/1606.08415

[6] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for activation functions. arXiv:1710.05941. Retrieved from https://arxiv.org/abs/1710.05941

[7] Noam Shazeer. 2020. GLU variants improve transformer. arXiv:2002.05202. Retrieved from https://arxiv.org/abs/2002.05202

[8] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE* 105, 12 (2017), 2295–2329.

[9] Dexu Lin, Liao Edward, Somdeb Majumdar, Aaron Lamb, and Karamvir Chatha. 2018. Approximation of non-linear functions in fixed point using look-up tables. US Patent No. 10037306.

[10] Xue Geng, Jie Lin, Bin Zhao, Anmin Kong, Mohamed M. Sabry Aly, and Vijay Chandrasekhar. 2019. Hardware-aware softmax approximation for deep neural networks. In *Proceedings of the 14th Asian Conference on Computer Vision (ACCV '18)*. Springer, 107–122.

[11] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, et al. 2024. FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Retrieved from https://api.semanticscholar.org/CorpusID:266844224

[12] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971

[13] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[14] Zhikai Li and Qingyi Gu. 2023. I-VIT: Integer-only quantization for efficient vision transformer inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 17065–17075.

[15] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. I-BERT: Integer-only BERT quantization. In *Proceedings of the 38th International Conference on Machine Learning* 139 (2021), 5506–5518. Retrieved from https://proceedings.mlr.press/v139/kim21d.html

[16] Marchisio Alberto, Dura Davide, Capra Maurizio, Martina Maurizio, Masera Guido, and Shafique Muhammad. 2023. SwiftTron: An efficient hardware accelerator for quantized transformers. In *Proceedings of the 2023 International Joint Conference on Neural Networks (IJCNN '23)*, 1–9.

[17] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. 2021. NPE: An FPGA-based overlay processor for natural language processing. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '21)*. ACM, New York, NY, 227.

[18] Ali Hadi Zadeh, Mostafa Mahmoud, Ameer Abdelhadi, and Andreas Moshovos. 2022. Mokey: Enabling narrow fixed-point inference for out-of-the-box floating-point transformer models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 888–901.

[19] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. 2022. Outlier suppression: Pushing the limit of low-bit transformer language models. In *Advances in Neural Information Processing Systems*, Vol. 35, 17402–17414.

[20] Teng Wang, Lei Gong, Chao Wang, Yang Yang, Yingxue Gao, Xuehai Zhou, and Huaping Chen. 2022. VIA: A novel vision-transformer accelerator based on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4088–4099.

[21] Mingqiang Huang, Junyi Luo, Chenchen Ding, Zikun Wei, Sixiao Huang, and Hao Yu. 2023. An integer-only and group-vector systolic accelerator for efficiently mapping vision transformer on Edge. *IEEE Transactions on Circuits and Systems I: Regular Papers* 70, 12 (2023), 5289-5301.

[22] Yueyin Bai, Hao Zhou, Keqing Zhao, Hongji Wang, Jianli Chen, Jun Yu, and Kun Wang. 2023. FET-OPU: A flexible and efficient FPGA-based overlay processor for transformer networks. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv:2010.11929. Retrieved from https://arxiv.org/abs/2010.11929

[24] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. In *Proceedings of the International Conference on Machine Learning*. PMLR, 10347–10357.

[25] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 10012–10022.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805

[27] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open pre-trained transformer language models. arXiv:2205.01068. Retrieved from https://arxiv.org/abs/2205.01068

[28] Mario Drumond, Tao LIN, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with hybrid block floating point. In *Advances in Neural Information Processing Systems*. S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31, Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2018/file/6a9aeddfc689c1d0e3b9ccc3ab651bc5-Paper.pdf

[29] Mo Song, Jiajun Wu, Yuhao Ding, and Hayden Kwok-Hay So. 2023. SqueezeBlock: A transparent weight compression scheme for deep neural networks. In *Proceedings of the 2023 International Conference on Field Programmable Technology (ICFPT '23)*, 238–243. DOI: https://doi.org/10.1109/ICFPT59805.2023.00032

[30] Haoli Bai, Lu Hou, Lifeng Shang, Xin Jiang, Irwin King, and Michael R. Lyu. 2022. Towards efficient post-training quantization of pre-trained language models. In *Advances in Neural Information Processing Systems*, Vol. 35, 1405–1418.

[31] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. 2021. Post-training quantization for vision transformer. In *Advances in Neural Information Processing Systems*, Vol. 34, 28092–28103.

[32] Yang Lin, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. 2022. FQ-ViT: Post-training quantization for fully quantized vision transformer. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI '22)*, 1173–1179.

[33] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. arXiv:2305.14314. Retrieved from https://arxiv.org/abs/2305.14314

[34] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. arXiv:2106.08295. Retrieved from https://arxiv.org/abs/2106.08295

[35] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning (2023)*, Article 1585, 13 pages.

[36] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. 2021. HAWQ-V3: Dyadic neural network quantization. In *Proceedings of the International Conference on Machine Learning*. PMLR, 11875–11886.

[37] John Bridle. 1989. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Advances in Neural Information Processing Systems*, Vol. 2.

[38] Jacob R. Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. 2021. Softermax: Hardware/software co-design of an efficient softmax for transformers. arXiv:2103.09301. Retrieved from https://arxiv.org/abs/2103.09301

[39] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 75–84.

[40] Xiao Dong, Xiaolei Zhu, and De Ma. 2019. Hardware implementation of softmax function based on piecewise LUT. In *Proceedings of the 2019 IEEE International Workshop on Future Computing (IWOFC '19)*. IEEE, 1–3.

[41] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. 2020. LogicNets: Co-designed neural networks and circuits for extreme-throughput applications. In *Proceedings of 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), IEEE*, 291–297. DOI: http://dx.doi.org/10.1109/FPL50879.2020.00055

[42] Mahdi Nazemi, Ghasem Pasandi, and Massoud Pedram. 2018. NullaNet: Training deep neural networks for reduced-memory-access inference, 1807.08716. arXiv:1807.08716. Retrieved from https://arxiv.org/abs/1807.08716

[43] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, et al. 2020. A^ 3: Accelerating attention mechanisms in neural networks with approximation. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*. IEEE, 328–341.

[44] Zhengang Lit, Mengshu Sun, Alec Lu, Haoyu Ma, Geng Yuan, Yanyue Xie, Hao Tang, Yanyu Li, Miriam Leeser, Zhangyang Wang, et al. 2022. Auto-ViT-Acc: An FPGA-aware automatic acceleration framework for vision transformer

with mixed-scheme quantization. In *Proceedings of the 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL '22)*. IEEE, 109–116.

[45] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. 2021. Algorithm-hardware co-design of attention mechanism on FPGA devices. *ACM Transactions on Embedded Computing Systems* 20, 5s (2021), 1–24.

[46] Zejian Liu, Gang Li, and Jian Cheng. 2021. Hardware acceleration of fully quantized BERT for efficient natural language processing. In *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE '21)*. IEEE, 513–516.

[47] NVIDIA. 2024. Transformer engine documentation. Retrieved from https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/

[48] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. FTRANS: Energy-efficient acceleration of transformers using FPGA. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '20)*. ACM, New York, NY, 175–180. DOI: https://doi.org/10.1145/3370748.3406567

[49] Suyeon Hur, Seongmin Na, Dongup Kwon, Joonsung Kim, Andrew Boutros, Eriko Nurvitadhi, and Jangwoo Kim. 2023. A fast and flexible FPGA-based accelerator for natural language processing neural networks. *ACM Transactions on Architecture and Code Optimization* 20, 1, Article 11 (Feb. 2023), 24 pages. DOI: https://doi.org/10.1145/3564606

[50] Jinming Zhuang, Zhuoping Yang, Shixin Ji, Heng Huang, Alex K. Jones, Jingtong Hu, Yiyu Shi, and Peipei Zhou. 2024. SSR: Spatial sequential hybrid architecture for latency throughput tradeoff in transformer acceleration. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. ACM, New York, NY, 55–66. DOI: https://doi.org/10.1145/3626202.3637569

[51] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. 2024. Understanding the potential of FPGA-based spatial acceleration for large language model inference. *ACM Transactions on Reconfigurable Technology and Systems* (Apr. 2024). 1936–7406. DOI: https://doi.org/10.1145/3656177

[52] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. VAttention: Dynamic memory management for serving LLMs without paged attention. Retrieved from https://arxiv.org/abs/2405.04437

[53] Xin Yang and Tao Su. 2022. EFA-trans: An efficient and flexible acceleration architecture for transformers. *Electronics* 11, 21 (2022), 3550.

[54] Qiwei Dong, Xiaoru Xie, and Zhongfeng Wang. 2024. SWAT: An efficient Swin transformer accelerator based on FPGA. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASPDAC '24)*. IEEE Press, 515–520. DOI: https://doi.org/10.1109/ASP-DAC58780.2024.10473931

[55] Kyle Marino, Pengmiao Zhang, and Viktor K. Prasanna. 2023. ME-ViT: A single-load memory-efficient FPGA accelerator for vision transformers. In *Proceedings of the 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC '23)*. IEEE, 213–223.

[56] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. 2022. DFX: A Low-latency multi-FPGA appliance for accelerating transformer-based text generation. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*. IEEE, 616–630.

[57] Yueyin Bai, Hao Zhou, Keqing Zhao, Manting Zhang, Jianli Chen, Jun Yu, and Kun Wang. 2023. LTrans-OPU: A low-latency FPGA-based overlay processor for transformer networks. In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL '23)*, 283–287. DOI: https://doi.org/10.1109/FPL60245.2023.00048

[58] Wenhua Ye, Xu Zhou, Joey Zhou, Cen Chen, and Kenli Li. 2023. Accelerating attention mechanism on FPGAs based on efficient reconfigurable systolic array. *ACM Transactions on Embedded Computing Systems* 22, 6, Article 93 (Nov. 2023), 22 pages. DOI: https://doi.org/10.1145/3549937

[59] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '18)*. IEEE, 522–531.

[60] Chris Lomont. 2003. *Fast Inverse Square Root*. Technical Report 32.

[61] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. 2019. A study of BFLOAT16 for deep learning training. arXiv:1905.12322. Retrieved from https://arxiv.org/abs/1905.12322

[62] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2016. Deep learning with INT8 optimization on Xilinx devices. Xilinx White Paper.

[63] Xinheng Liu, Yao Chen, Prakhar Ganesh, Junhao Pan, Jinjun Xiong, and Deming Chen. 2022. HiKonv: High throughput quantized convolution with novel bit-wise management and computation. In *Proceedings of the 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC '22)*, 140–146. DOI: https://doi.org/10.1109/ASP-DAC52403.2022.9712553

[64] Junjie Bai, Fang Lu, and Ke Zhang. 2019. ONNX: Open neural network exchange. Retrieved from https://github.com/onnx/onnx

[65] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. ACM, New York, NY, Article 61, 9 pages. DOI: https://doi.org/10.1145/3400302.3415609

[66] Yuhao Ding, Jiajun Wu, Yizhao Gao, Maolin Wang, and Hayden Kwok-Hay So. 2023. Model-platform optimized deep neural network accelerator generation through mixed-integer geometric programming. In *Proceedings of the 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '23)*, 83–93. DOI: https://doi.org/10.1109/FCCM57271.2023.00018

[67] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255.

[68] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. arXiv:1804.07461. Retrieved from https://arxiv.org/abs/1804.07461

[69] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[70] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. arXiv:1606.06031. Retrieved from https://arxiv.org/abs/1606.06031

[71] Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools. arXiv:2406.12793. Retrieved from https://arxiv.org/abs/2406.12793

[72] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. 2020. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. In *Proceedings of the 2020 IEEE 33rd International System-on-Chip Conference (SOCC '20)*. IEEE, 84–89.

[73] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. arXiv:2205.14135. Retrieved from https://arxiv.org/abs/2205.14135

[74] Jiajun Wu, Mo Song, Jingmin Zhao, and Hayden Kwok-Hay So. 2024. A case for low bitwidth floating point arithmetic on FPGA for transformer based DNN inference. In *Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '24)*. IEEE, 178–185.

[75] Yuntao Han and Qiang Liu. 2023. HPTA: A high performance transformer accelerator based on FPGA. In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL '23)*, 27–33. DOI: https://doi.org/10.1109/FPL60245.2023.00012

Received 14 October 2024; revised 14 October 2024; accepted 19 December 2024