# A Case for Low Bitwidth Floating Point Arithmetic on FPGA for Transformer Based DNN Inference

Jiajun Wu*, Mo Song*, Jingmin Zhao, Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering, University of Hong Kong, Hong Kong
{jjwu, songmo, jmzhao, hso}@eee.hku.hk

***Abstract*—Modern Transformer-based deep neural networks are increasingly reliance on an accelerator's ability to perform the model's non-linear operations accurately and efficiently. Unfortunately, while conventional low bitwidth fixed-point arithmetic are superior in terms of power, performance, and area consumption for low-precision linear operations, they are not well-suited to perform non-linear operations that require large dynamic range and high precision. In this work, we present a case for using a mix of low-bitwidth block floating-point and single precision floating-point operations to address the needs of both the linear and non-linear layers of Transformer-based DNNs. To support both datatypes in hardware, an 8-bit block floating point (`bfp8`) processing array that can be reconfigured to implement 32-bit floating point (`fp32`) computation during run time is proposed. With the support of both datatypes, pre-trained Transformer models in `fp32` can now be deployed without the need for quantization-aware retraining. The proposed `bfp8` systolic array has been implemented efficiently on an AMD Alveo U280 FPGA, consuming only marginally more hardware resources than an int8 equivalence. It attains 2.052 TOPS throughput for the linear operations in `bfp8` mode, which is equivalent to over 95% of the theoretical maximum 8-bit throughput of the target platform, while it achieves 33.88 GFLOPS throughput when operating in `fp32` mode. By demonstrating the hardware efficiency of low-precision floating-point operations on FPGAs, this work provides an attractive tradeoff direction in the vast design space of accuracy, speed, power, and time-to-market for full-stack Transformer models acceleration.**

## I. Introduction

The emergence of Transformer-based deep neural networks (DNNs) in recent years is introducing a new wave of challenges to conventional accelerator design that commonly employs fixed-point arithmetic for inference. To begin with, modern Transformer-based models [1], especially those colloquially referred to as large language models (LLMs), are substantially larger than even the largest vision models from just a decade ago. For instance, the largest Open Pre-trained Transformers (OPT) model contains 175 B parameters [2] while the ResNet-101 model had only 44.6 M parameters [3]. As a result, model retraining, which is an indispensable step in model quantization, is becoming either undesirable or infeasible due to the lack of training data and computing resources, as well as concerns regarding data privacy.

Perhaps more importantly, a defining feature of Transformer models is their increasing reliance on the use of non-linear operations in their designs. For example, every Transformer block contains a softmax layer for self-attention, as well as a GELU and a LayerNorm layer that are intertwined with the rest of the linear layers [1]. Unlike vision DNNs from the last decade that primarily operated with linear layers, researchers have repeatedly demonstrated that these non-linear operations in Transformers are highly susceptible to quantization error [4]–[6], [6], [7]. Although approximation schemes suitable for hardware implementations for some of these non-linear functions have been proposed [4], [8], new non-linear functions are constantly being introduced as the field progresses [9], [10]. As a result, in order to adapt to this rapidly evolving area, a run time programmable hardware solution is highly desirable.

To address these challenges, we present a case for leveraging low-bitwidth floating-point arithmetic to implement efficient hardware accelerators for Transformer-based DNNs. Specifically, we present our design that employs 8-bit block floating-point (`bfp8`) for linear operations of a Transformer model, while it relies on single-precision floating-point (`fp32`) for non-linear operations. As previously shown in [11], block-based low-bitwidth floating-point operations are adequate to preserve the accuracy of Transformer models without the need for lengthy quantization-aware re-training, which is particularly desirable with modern large models. Here, we present a systolic array design that can perform matrix multiplications in `bfp8` mode with performance comparable to an equivalent `int8` implementation, making a case that `bfp8` can indeed offer both the accuracy and the hardware efficiency needed for Transformer acceleration. In addition, the proposed architecture can be reconfigured into a `fp32` vector processing unit during run time, which can be programmed to support all non-linear functions of the model. With this mixed-precision approach, our method combines both types of operations into a single processing unit and allows Transformer models to be deployed without the need for time-consuming retraining. Our evaluation on an AMD Alveo U280 FPGA shows that our design achieves 2.052 TOPS throughput in `bfp8` matrix multiplication and theoretically up to 33.88 GFLOPS throughput in `fp32` vector processing mode. When compared to an equivalent `int8` implementation, our `bfp8` design consumes the same number of DSPs and 1.19× more flip-flops (FFs). In addition, when compared to a design with individual `bfp8` and `fp32` processing hardware, our reconfigurable design saves 20.0% DSPs, 61.2% FFs and 43.6% look-up tables (LUTs) without performance degradation. With high accuracy and low hardware overhead, the proposed system represents

*Equal contribution.

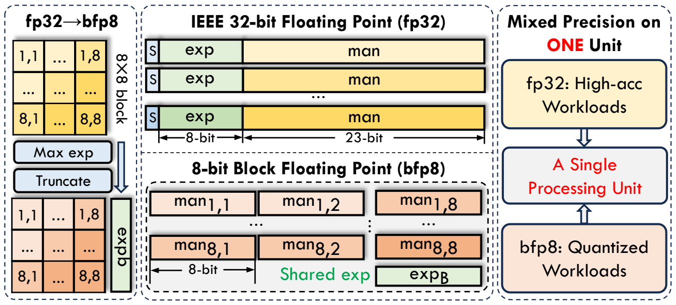


Fig. 1: Illustration of the 8-bit block floating-point (bfp) and the IEEE standard 32-bit floating-point (fp32) format. Our proposed solution aims to process the two formats in one single hardware unit.

a new trade-off direction in the vast design space among model accuracy, performance, power, and time-to-market for transformer model acceleration.

## II. Hardware Design

### A. Arithmetic Analysis

As shown in Fig. 1, a block floating-point (bfp) system encodes the values of every element in a block with a shared exponent, while each element is encoded with its own mantissa. Let the shared exponent of a block be $expb$ and the mantissa of each element be $man_{ij}$, with the size of the 2-D block being $m \times n$, the value of each individual data ($val_{ij}$) within a block can be expressed as:

$$val_{ij} = 2^{expb} \times man_{ij}, \quad i \in [1,m], j \in [1,n] \tag{1}$$

In this study, 8-bit 2s complement encoding was used for both the shared exponent and each block element to form a bfp8 representation format.

To efficiently deploy mixed-precision workloads with bfp8 and fp32 formats, we first illustrate how to fuse 8-bit block floating-point (bfp8) matrix multiplication (MatMul) and 32-bit floating-point (fp32) basic operations into a single processing unit by arithmetic analysis.

When performing bfp8 MatMul between two blocks $\mathbf{X}$ ($m \times n$) and $\mathbf{Y}$ ($n \times p$), as indicated by equation Eqn. (1), the resulting matrix $\mathbf{Z}$ can be expressed as:

$$\begin{aligned} \mathbf{X} \cdot \mathbf{Y} = \mathbf{Z}, \quad Z_{ij} &= \sum_{k=1}^{n} X_{ik} Y_{kj}, i \in [1,m], j \in [1,p] \\ expb_Z &= expb_X + expb_Y, \\ man_{Z_{ij}} &= \sum_{k=1}^{n} man_{X_{ik}} \times man_{Y_{kj}} \end{aligned} \tag{2}$$

where the shared exponent $expb$ and mantissa $man$ are both set to 8-bit in bfp8 format. Therefore, the multiplication of two 2-D bfp8 blocks can be seen as an addition of int8 ($exp$) along with an int8 matrix multiplication. The use of a systolic array [12] is commonly considered appropriate for this type of multiplication. Consequently, support for bfp8 addition is necessary to combine the partial blocks since the block size is fixed in our design:

$$\begin{aligned} \mathbf{X} + \mathbf{Y} = \mathbf{Z}, \quad Z_{ij} &= X_{ik} + Y_{kj}, i \in [1,m], j \in [1,p] \\ man_{Z_{ij}} &= man_{X_{ik}} + man_{Y_{kj}} >> (expb_X - expb_Y) \\ expb_Z &= expb_X \end{aligned} \tag{3}$$

For simplified representation, we assume $expb_X \geq expb_Y$, i.e., the mantissa of $man_{Y_{kj}}$ needs to be shifted to the right for alignment. In a real hardware design, it is necessary to implement a comparator for addition operations.

The key motivation in this work is to fuse the fp32 operations with such low-bitwidth arithmetic in bfp8. Taking into account fp32 multiplication of two values $x$ and $y$, and assuming that the exponent values have already been biased, and the sign bit has been fused to the mantissa (i.e, signed magnitude), the process can be presented as follows:

$$x \times y = 2^{exp_x + exp_y} \times (man_x \times man_y) \tag{4}$$

In the fp32 data format, the exponent and the mantissa field are 8-bit and 24-bit, respectively. Many previous studies proposed mixed-precision support in fp by separating the mantissa field into low-bitwidth slices [13]. Each 8-bit slice is an individual int8 number, thus the fp32 multiplication in Eqn. (4) can be converted to:

$$\begin{aligned} x \times y &= 2^{exp_x + exp_y} \times (man_x \times man_y) \\ &= 2^{exp_x + exp_y} \times \sum_{i=0}^{2} \sum_{j=0}^{2} man_x(i) \times man_y(j) << 8(i+j), \\ man_{x,y}(i) &= man_{x,y}[8i+7:8i] \end{aligned} \tag{5}$$

where the $man_{x,y}(i)$ is the 8-bit slice of mantissa. The final mantissa result will be renormalized and truncated, which has been omitted in Eqn. (5). Based on such separation, the fp32 multiplication can be converted to one int8 addition and a sum of nine int8 multiplications with shifting. Therefore, fp32 multiplication should be able to run on the bfp8 matrix multiplication unit with different control flow.

For fp32 addition, we apply the similar modification in Eqn. (3), as shown in Eqn. (6).

$$\begin{aligned} x + y &= 2^{exp_z} \times man_z \\ man_z &= man_x + man_y >> (exp_x - exp_y) \\ exp_z &= exp_x \end{aligned} \tag{6}$$

where the sign bit is integrated in the mantissa field. Similar to Eqn. (3), we assume $exp_x \geq exp_y$ so the $man_y$ needs to be shifted.

Table I presents an overview of the fundamental functions of bfp8 MatMul and fp32 multiply & add, as determined in this section. The consistent numerical trends observed here inspire the creation of the multi-mode processing unit, as detailed in the subsequent section.

TABLE I: Shared Basic Operations Between `bfp8` and `fp32`

| Arithmetic | bfp8 MatMul | fp32 mul | fp32 add |
|---|---|---|---|
| Basic Operation | * 8-bit MAC<br>* Align & shift<br>* Partial sum add<br>* Normalize | * 8-bit MAC<br>* Partial sum add<br>* Normalize | * Align & shift<br>* Mantissa add<br>* Normalize |

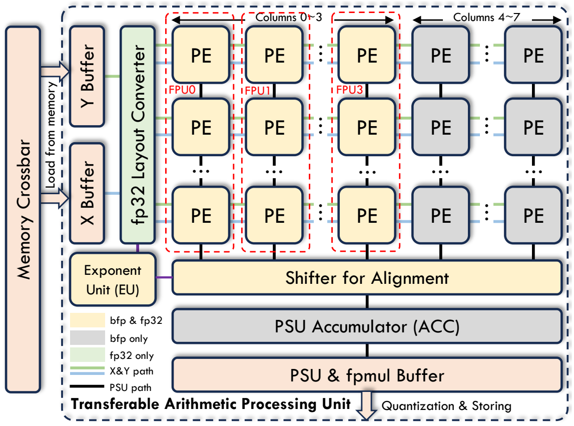


Fig. 2: The proposed processing unit architecture. We implement a DSP48E2 block per processing element (PE) to deploy 8-bit based multiplications.

## B. Hardware Design

Fig. 2 depicts the proposed design of the multi-mode processing unit. In this study, we have chosen the size of the two-dimensional block of `bfp8` to be $8 \times 8$, which contains 64 elements. Consequently, the foundational architecture is structured as a systolic array with dimensions of 8 rows and 8 columns, which corresponds to the size of a block. Each processing element (PE) within the array comprises data registers, pre-shifters, and a DSP48E2 block [14] that integrates a pre-adder, a $27 \times 18$ tiplier, and an internal partial sum (PSU) adder, as shown in Fig. 3. When matrix multiplication (MatMul) is performed, partial sums (PSUs) are accumulated through column connections. At the bottom of each column, there exists a shifter to align the mantissa according to the exponent results, along with a PSU buffer to store the previous summations. Moreover, it integrates this PE for performing `fp32` multiplication. Due to the absence of data reuse in the fpmul, systolic dataflow is not utilized in this scenario. Instead, the `fp32` layout crossbar directly broadcasts the four `fp32` data into four columns according to the slices to be processed in parallel. In addition to the systolic array, an exponent unit (EU) has been devised to handle exponent calculations for both `bfp8` and `fp32` fundamental operations. There is also a simple XOR gate to process significant bits of `fp32` data, which is not presented in the figure for simplification. Further elaboration on the dataflow in two formats that utilize the same hardware unit is provided in the following sections.

To improve computational performance, we implement combined multiply-accumulation (MAC) optimization using AMD DSP48E2 blocks [15], [16], as illustrated in Fig. 3. With

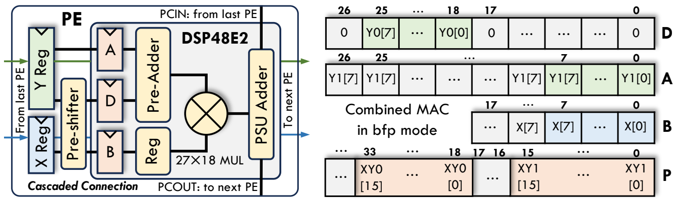


Fig. 3: Processing Element (PE) design and the optimization scheme in `bfp8` MatMul mode to process 2 MACs in one DSP block.

the mantissa bitwidth of `bfp8` set at 8-bit, each DSP48E2 block can accommodate two MACs, leading to a possible overall improvement in computational efficiency. This configuration allows for the accumulation of up to 7 product terms without overflow issues in the lower bits. By configuring the row numbers as 8, we can cleverly circumvent such overflow problems in the `bfp8` MatMul computations. When it comes to `fp32` mode, we apply a tricky optimization for better FPGA routing, by pre-shifting the input mantissa slices before multiplication, instead of shifting the results after multiplication (Eqn. (5)). With such an optimization, the DSP blocks in each column can be connected as a cascaded topology, fitting the bottom layout of the DSP [14]. It is important to note that in the `fp32` modes, DSP blocks cannot operate in the combined MAC setup because the results need to be shifted according to Eqn. (5), resulting in an overflow in the lower bits. Also, due to the data layout configuration explained in the following context, the `fp32` mode can only utilize 4 columns of PEs (i.e., 4 FPUs). In summary, for one unit, the theoretical peak throughput of `bfp8` (operations per second, OPS) can be presented as:

$$Thr_{bfp-peak} = rows \times columns \times 2 \times 2 \times Freq \tag{7}$$

where the the $Freq$ refers to the hardware frequency. The first $\times 2$ factor in $Thr_{bfp-peak}$ stands for the combined MAC optimization, and the second $\times 2$ factor is determined by the assumption that each MAC operation actually consists of two operations (multiplication and addition).

## C. Data Layout in Buffers

In correspondence with the Processing Element (PE) array, the X buffer and Y buffer are intended to store information and reorganize the data structure for further processing. Each block ram (BRAM) uses one BRAM18 in the AMD FPGA devices. The data layout of X buffers is illustrated in Fig. 4 using a single block of `bfp8` and 32 `fp32` data as an example. We have set up 16 BRAMs to store the mantissa of `bfp8` and additional one BRAM for the exponent of `bfp8`. In this figure, each column refers to one BRAM, and we index them as $0 \sim 15$. Consequently, each block occupies 8 mantissa BRAMs. When operating in `fp32` mode, the 8 mantissa BRAMs are repurposed to store `fp32` data. As illustrated before, the significant bit is integrated in the mantissa field,

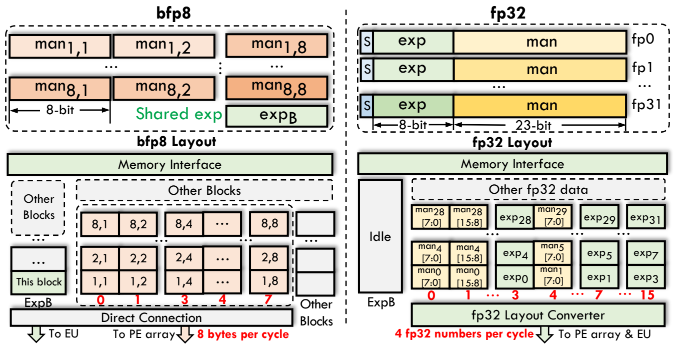


Fig. 4: Data layout in X buffers including 17 BRAMs with 8-bit (one Byte) port. For the Y buffer, data layout is similar except in `bfp8` mode, the other half of BRAMs also output data during runtime, due to the combined MAC optimization.

thus each data contains a 24-bit signed-magnitude mantissa in total. Then, all 4 BRAMs are assigned to a single `fp32` number, while the exponent BRAM remains inactive. For example, as shown in Fig. 4, BRAMs $0 \sim 2$ are utilized for the three 8-bit mantissa slices corresponding to Eqn. (5) of $fp_0$, and BRAM 3 is dedicated to its exponent ($exp_0$). BRAMs $4 \sim 15$ follow the same configuration. It is important to note that due to the implementation of a combined MAC optimization, the Y buffer requires replication of the BRAMs (i.e., 16 BRAMs are all output to PE arrays with two blocks), while maintaining the same data layout. The analysis in data layout explains why only 4 FPUs can be used in `fp32` mode, since the total bitwidth of the BRAM output per cycle is only 128-bit, thus, the bandwidth of buffers is 4 `fp32` data per cycle. Therefore, we only enable the 4 columns of PEs while keeping the remaining PEs idle to save power, as illustrated in Section II-B. Thus, the throughput of the `fp32` mode (floating-point operations per second, FLOPS) can be presented as:

$$Thr_{fp32-peak} = 4 \times Freq \tag{8}$$

## D. *Dataflow*

Fig. 5 (a) illustrates the dataflow in bfp8 MatMul mode, where a Y-stationary dataflow is employed. Initially, two $\mathbf{Y}$ blocks are loaded into the PE arrays and saved in the $\mathbf{Y}$ register within each PE. Subsequently, the $\mathbf{X}$ blocks are fed into the PE arrays using a systolic array dataflow approach, where $\mathbf{X}$ moves horizontally and the partial sums accumulate vertically. This Y-stationary dataflow necessitates keeping the $\mathbf{Y}$ matrix in the PEs for as long as possible to boost the computational throughput. Synchronously, the exponent unit computes the exponent addition of the two $\mathbf{Y}$ blocks with the streamed $\mathbf{X}$ blocks, and transmits the results to the shifters for alignment (Eqn. (3)). After shifting, the accumulator (ACC) calculates the results between different blocks by fetching the old data in the PSU buffer. Theoretically, the computational throughput (OPS) can be presented as:

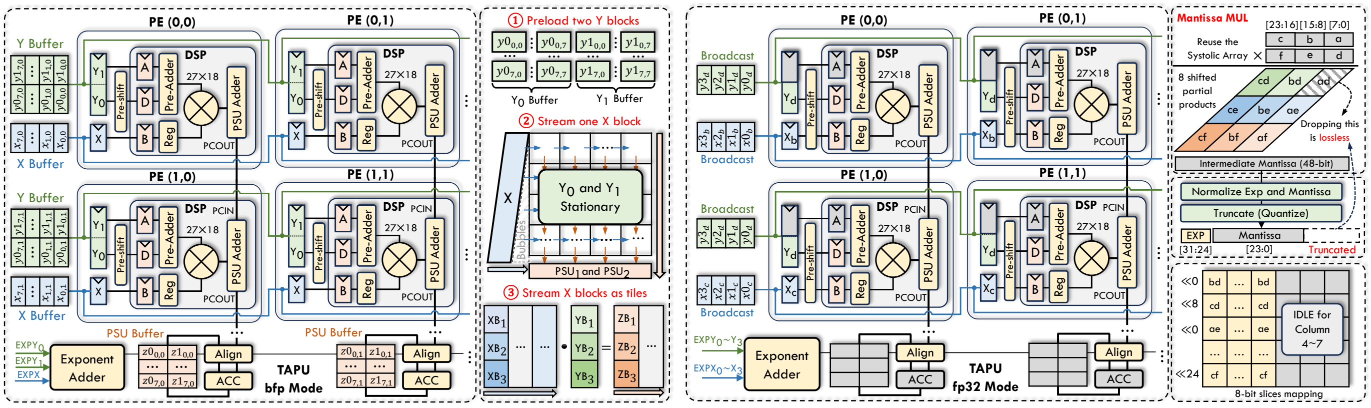
$$Thr_{bfp} = Thr_{bfp-peak} \times (8 \times N_{\mathbf{X}})/(8 \times N_{\mathbf{X}} + 15) \tag{9}$$

where the $N_{\mathbf{X}}$ refers to the number of streamed $\mathbf{X}$ blocks and $Thr_{bfp-peak}$ is the peak performance of one processing unit. The 8 and 15 factors indicate cycles for each $\mathbf{X}$ block and the pre-load $\mathbf{Y}$ & systolic array triangle cycles. In the current design, we set the maximum number of continuous $\mathbf{X}$ blocks as 64 due to the BRAM18 architecture, so the PSU buffer depth is 512, and the computational throughput can reach up to 97.15% of the peak performance.

When operating in fpmul mode, we have detailed the adjustment of `fp32` mantissa multiplication to `bfp8` MatMul by reusing the DSPs. For simplified presentation, we do not depict the significant bit operation between two `fp32`, which only requires one XOR gate. As shown in Eqn. (4) and Fig. 5 (b), the 24-bit mantissa is divided into three slices (a, b, c and d, e, f), and the total result is obtained by summing up 8 partial products. To accommodate the 8-row PE array, the least significant partial product is omitted. These partial products require shifting before being combined with other products. Leveraging the DSP48E2 architecture, we implement a pre-shifting mechanism on the input slices, as opposed to post-multiplication shifting. For example, the partial product term *cd* necessitates an 8-bit left shift, and all PEs in row 1 (e.g., $\mathrm{PE}(1, 0)$, $(1, 1)$, etc.) left-shift the input X slice ($\mathrm{X}_c$) and Y slice ($\mathrm{Y}_c$) by 4 bits to meet the 8-bit pre-shifting requirement. In this scheme, the maximum number of left-shifted bits in `fp32` multiplication is 24, thus the 27-bit & 18-bit input widths of DSP48E2 support such pre-shifting without encountering overflow. It is important to note that due to the pre-shifting operation, the combined MAC optimization cannot be utilized in this mode. The layout converter switches and duplicates the `fp32` mantissa & exponent slices, to fit the data mapping in Fig. 5 (b). Additionally, due to the data arrangement discussed above, only the 4 columns of PEs (4 FPUs) can be used in parallel for `fp32` multiplication, with the remaining PEs idle (in sleep mode) during execution.

Regarding the fpadd mode, in the case of PEs equipped with DSP blocks, they remain inactive since only the exponent unit, shifter, and mantissa adder are necessary for this operation, as indicated by Eqn. (6). The processing unit utilizes the shifter and the PSU accumulator for fpadd tasks, ensuring minimal hardware overhead. For clarity, the fpadd dataflow is not illustrated in Fig. 5. In this mode, the mantissa is treated as a single unit rather than being divided into three slices and is sent directly to the ACC for addition. In the fpmul and fpadd modes, the X and Y slices are treated as streams, similar to the bfp mode. Currently, we set the input stream length as $L_{fp32}$ (i.e., in total $4 \times$ `fp32` numbers), which is set to a maximum of 128 due to the memory capacity of a single BRAM18 block. Similar to the analysis in `bfp8` MatMul mode (Eqn. (9)), the theoretical computational throughput (GFLOPS) can be calculated as follows. Note that since there is no preloading in `fp32` mode, so the factor 15 in Eqn. (9) becomes 8.

$$Thr_{fp32} = Thr_{fp32-peak} \times L_{fp}/(L_{fp} + 8) \tag{10}$$



(a) (b)

Fig. 5: The proposed workflow in `bfp8` MatMul mode (a) and `fp32` multiplication mode (b). Partial summations and products are accumulated through columns using the cascaded connection of AMD DSP48E2 blocks. The modules in deep gray indicate that they are idle in this mode. Due to the data layout in the bfp mode, this unit can utilize only four PE columns in the `fp32` mode, which supports four `fp32` multiplications in parallel.

TABLE II: Hardware unitilization of the proposed processing unit with other necessary modules

| | Components | LUT | FF | BRAM | DSP |
|---|---|---|---|---|---|
| PU* | PE Array | 1317 | 1536 | 0 | 64 |
| | Shifter & ACC | 768 | 644 | 0 | 8 |
| | Buffer & Layout Converter | 752 | 764 | 50.0 | 0 |
| | Exponent Unit | 269 | 195 | 0 | 0 |
| | Quantizer | 348 | 524 | 0 | 0 |
| | Misc.† | 483 | 1944 | 3.0 | 0 |
| | Memory Interface | 2963 | 4270 | 4.5 | 0 |
| | Controller | 448 | 452 | 0 | 0 |
| | Total | 7348 | 10329 | 57.5 | 72 |

*The key components of the proposed multi-mode processing unit (PU).
†Including delay chains, AXI-Stream register slices, etc.

## III. Evaluation

The proposed multi-mode processing unit is implemented on AMD Alveo U280 FPGA with high bandwidth memory (HBM), where we evaluate utilization, throughput, and energy consumption. We developed the hardware design using Verilog HDL and implemented it with the Vitis 2021.2 tools. In addition to the processing element arrays and buffers, we integrate essential components like memory interface, quantization unit, etc., for real-time computation. The hardware system runs on a 300 MHz frequency. With the multi-mode unit, the complete system can perform `bfp8` MatMul and `fp32` multiply & add operations with comprehensive support.

### *A. Hardware Utilization*

Table II shows the hardware utilization divided into different components for one processing unit with other necessary modules. Compared to the pure `bfp8` design, the overhead modules (Layout Converter and controller) only take up $10.23\%$ LUT and $11.77\%$ FF, without additional BRAM or DSP, to support the proposed hybrid data formats. The resource efficiency comes from the reuse of PE arrays and buffers in both `bfp8`

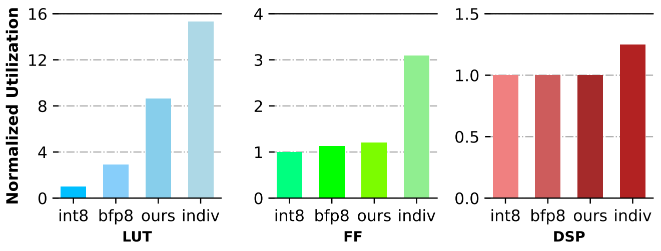


Fig. 6: Resource utilizations of different processing unit designs. Results normalized to the `int8` design.

and `fp32` modes, thus enabling our design to be expanded to multiple parallel units on FPGA, running with independent instructions.

To further investigate the benefits of using a single unit for both `bfp8` and `fp32` data formats, we established four different implementations of PE arrays for matrix multiplication, illustrated in Fig. 6. These implementations comprise 1) an array for `int8` MatMul, 2) an array for exclusive `bfp8` MatMul (i.e., no `fp32` operations), 3) our proposed multi-mode design that employs a unified array for both `bfp8` and `fp32` data, and 4) separated processing units for `bfp8` MatMul and parallel `fp32` multiply & add, processing independently (briefly denoted as “indiv” in Fig. 6). For individual `fp32` units, we leverage the AMD floating-point IP core for synthesis. It should be noted that we configure the parallel level of `fp32` operations in the individual units as $4$, to align with the configuration in our design. The assessed hardware design only comprises the PE array, the exponent unit, the mantissa shifters, and the runtime controller to ensure a fair comparison. The findings indicate that `bfp` MatMul requires more LUTs than the `int8` design due to the mantissa shifter required to align different blocks. Additionally, our multi-mode processing unit only introduces LUT overhead compared to a pure `bfp8`

TABLE III: Comparison with the related mixed-precision hardware accelerators on FPGA

| Work | Data Format | Application | Need Retraining | FPGA Platform | Utilization | | | | Frequency (MHz) | Throughput (GOPS) | Efficiency (GOPS/DSP) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | LUT (k) | FF (k) | BRAM | DSP | | | |
| Lian et. al. [17] | `bfp8` | CNN | No | VX690T | 231.8 | 141.0 | 913 | 1027 | 200 | 760.83 | 0.74 |
| Wu et. al. [18] | `fp8` | CNN | No | XC7K325T | 154.6 | 180.6 | 234.5 | 768 | 200 | 1086.8 | 1.42 |
| Fan et. al. [19] | `bfp8` | CNN | No | Intel GX1150 | 437.2 | 170.9 | 2713 | 1518 | 220 | 1667 | 1.24 |
| Wong et. al. [20] | `bfp10` | CNN | No | KU115 | 386.3 | 425.6 | 1426 | 4492 | 125 | 794 | 0.18 |
| Auto-ViT-Acc [21] | `int4` & `int8` | Transformer | Yes | ZCU102 | 185.0 | - | - | 1152 | 150 | 907.8 | 0.59 |
| ViA [22] | `fp16` | Transformer | No | Alveo U50 | 258.0 | 257.0 | 1002 | 2420 | 300 | 309.6 | 0.13 |
| Ye et. al. [23] | `int8` & `int16` | Transformer | Yes | Alveo U250 | 736.0 | - | 1781 | 4189 | 300 | 1800 | 0.43 |
| Ours | `bfp8` & `fp32` | Transformer | No | Alveo U280 | 410.6 | 602.7 | 1353 | 2163 | 300 | 2052.06$^{\dagger}$ | 0.95 |

$^{\dagger}$Only list `bfp8` throughput here. The theoretical `fp32` throughput is 33.88 GFLOPS.

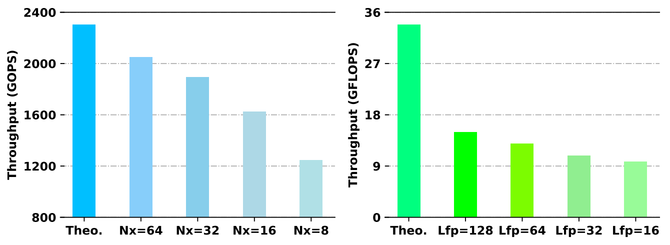


Fig. 7: Measured throughput of one unit on Alveo U280 FPGA under different workloads v.s. the theoretical maximum.

array (approximately 2.94×) due to the pre-shifting in each PE, while the utilization of FF and DSP remains nearly identical to that of the `bfp8` array. Opting for the individual processing approach with segregated units results in substantial hardware overhead compared to our design (2.58× FF and 1.25× DSP). As mentioned earlier, contemporary Transformer models demand either significant retraining overhead (through `int8` quantization) or mixed-precision inference where the non-linear functions retain a higher precision akin to `fp32`. Hence, our approach offers a novel alternative within the extensive design space that includes accuracy, speed, power efficiency, and time-to-market considerations for accelerating full-stack Transformer models.

## B. Throughput Analysis

The proposed multi-mode unit provides high flexibility for top-level compilers. Users can map different types of workload to the hardware with mixed-precision during run-time. Therefore, to evaluate the different performance, we implemented 15 processing units in the Alveo U280 FPGA to fully utilize the HBM channels[1], and several workloads have been tested in this design, including `bfp8` MatMul with various $N_{\mathbf{X}}$, and some `fp32` multiplication operations with various $L_{fp}$. All throughput results are measured by latency based on going through the whole FPGA system in 300 MHz, i.e., including the memory I/O latency, while the theoretical results are calculated by Eqn. (9) and Eqn. (10). Fig. 7 left part shows the results of linear layers with different tile lengths of $\mathbf{X}$ ($N_{\mathbf{X}} = 64, 32, 16, 8$). The theoretical throughput of `bfp8` MatMul is calculated by Eqn. (9). It can be seen that as the length of the stream increases, the throughput increases closer to the theoretical maximum. For the `fp32` operations, the performance results are displayed in the right section of Fig. 7. The maximum `fp32` throughput can be determined by referring to Eqn. (8). The observed pattern is similar to the `bfp8` mode, where a longer input stream length leads to increased throughput.

[1]Each multi-mode unit has 2 256-bit AXI channels connected to HBM

Based on this calculation, our FPGA prototype can achieve 33.88 GFLOPS performance on the Alveo U280 in `fp32` operations. However, as shown in Fig. 7, the real systematic throughput in FPGA is still far from the theoretical value. This is because in our current design, we have not optimized the compilation level which enables larger burst lengths for `fp32`, as the `fp32` operations have more random memory access compared to `bfp8` MatMul. Nevertheless, the trend of performance change is similar to the `bfp8` mode, indicating that users should try to boost the processing stream length in computation. The difference between different workloads reflects the optimization space to boost the both `bfp8` and `fp32` performance, and thus we will explore further compilation-level optimization in the future. It is important to mention that in our current configuration, the division operations in `fp32`, which constitute a minor portion, are executed on the host CPU due to lack of support.

## C. Comparison with Previous Work

To compare the proposed processing unit with previous mixed-precision acceleration systems, we adhere to the same settings in the throughput evaluation. We present hardware utilization, throughput, and efficiency with a qualitative configuration in Table III, compared to previous studies. The proposed processing unit can attain a throughput that is on par with other designs utilizing `bfp8`, low-bitwidth `fp` or integers (`int8`). Although some previous works have exceeded our throughput efficiency, we are able to integrate `fp32` operations into the proposed processing unit, thereby achieving greater area efficiency. From a systematic perspective, our approach offers increased flexibility to the compilation stage, enabled by the mixed-precision runtime without separate units. It is worth noting that some of these works leverage LUTs for computation, leading to higher throughput, whereas our evaluation strictly adheres to DSP computation based on the equations outlined in Section II-D. Looking ahead, we plan to delve deeper into high-precision floating-point optimization

TABLE IV: Estimated proportion of linear and non-linear operations of a DeiT-Small model

| Workloads Partitions | OPs or FLOPs[†] | Operations Proportion | Latency (ms) | Latency Proportion |
|---|---|---|---|---|
| `bfp8` MatMul | 2465M | 98.649% | 1.201 | 8.170% |
| `fp32` LayerNorm | 6.383M | 0.043% | 0.425 | 2.891% |
| `fp32` SoftMax | 145.3M | 0.969% | 9.686 | 65.887% |
| `fp32` GELU | 50.84M | 0.339% | 3.389 | 23.053% |

[†] Counted from all 12 blocks in a DeiT-Small model.

within the mixed-precision unit, as the `fp32` format is often overly precise for many machine learning systems.

### *D. Case Study: Vision Transformer*

As previously stated, there is a significant increase in the demand for high-precision floating-point accuracy, particularly in Transformer models that involve non-linear computations. To evaluate the overall latency performance, we utilized the DeiT-Small model. In this scenario, we employed `bfp8` for matrix operations such as MLP and self-attention, while `fp32` was used for non-linear functions. The SoftMax, GELU, and LayerNorm functions were integrated into basic arithmetic operations. Table IV illustrates the distribution of `bfp8` and `fp32` workloads, along with the calculated end-to-end latency. The analysis reveals that although `fp32` computation only takes $1.35\%$ workloads, it still contributes to $92.45\%$ of the total latency, underscoring the importance of optimizing mixed-precision techniques. It can be concluded that while our proposed processing unit can dynamically support both formats with minimal overhead, the processing efficiency for high precision (`fp32`) is still lacking. From Fig. 5 (b), the primary inefficiency in `fp32` can be attributed to the idle state of the PEs during execution due to the data layout constraint. To mitigate latency issues, we will explore the optimization of more suitable data formats to support the non-linear function unit within our hardware architecture.

## IV. Related Work

### *A. Mixed-Precision Quantization*

Researchers have found that different parts of DNN models show varying levels of vulnerability to quantization errors [24]. For example, the linear layer of a Transformer model is resilient to quantization errors even at very low bitwidths [5], [11], [19]. However, the accuracy of non-linear operations such as SoftMax, GELU, and LayerNorm significantly influences the overall accuracy of a Transformer-based model [6]. Consequently, recent quantization frameworks either attempt to quantize the non-linear functions through retraining, incurring substantial overhead [4], [5], or maintain these operations at higher precision [25], [26]. Although quantization frameworks can balance model performance and hardware efficiency without the need for retraining overhead [11], [22], the hardware expenses of high-precision computation are significant. For instance, a recent study has shown that as the embedding dimension increases, the latency of non-linear functions in Transformers increases significantly [8]. In this study, we consider this hardware efficiency for high-precision non-linear operations as a complementary investigation to understand mixed-precision quantization frameworks.

### *B. Mixed-Precision Hardware Architecture*

Previous research has explored various architectural designs to effectively support mixed-precision quantization [27]. For example, integer-based mixed-precision architectures [16], [28], [29], such as BitFusion [28] and bit-serial architecture [16], [30], have been explored. In contrast, since integer quantization suffers from accuracy loss or retraining overhead in emerging Transformer models, there is a growing interest among researchers in floating-point-based mixed-precision hardware [13], [25], [31], [32]. A common strategy in this domain involves partitioning the mantissa field of floating-point numbers into slices and utilizing shift-add operations to obtain the final outcomes, enabling different formats to leverage the same fundamental unit for each slice and thus reducing overhead [13]. We have integrated this concept into our design by segmenting the mantissa of `fp32` into several low-bitwidth slices to align with `bfp8` processing requirements. Despite the numerous efficient mixed-precision architectures proposed, existing quantization framework systems do not yet widely adopt such an approach. Instead, they often employ distinct hardware units for different formats at run-time [22], [33]. In this work, we present a novel alternative that combines all of these operations into a single hardware unit.

## V. Conclusions

In this paper, we have presented a case for using low-bitwidth floating-point arithmetic for Transformer-based DNNs inference. We demonstrate that low-bitwidth floating-point (`bfp8`) matrix multiplication can be implemented effectively in hardware with a marginal increase over an 8-bit integer equivalence while attaining processing throughput close to the platform maximum. In addition, we show that such an array can be effectively reconfigured during run-time into a programmable `fp32` vector processing unit that can be programmed to implement any non-linear functions required by future Transformer-based DNN models. With efficient support of both datatypes, the proposed design eliminates the need to quantize and retrain Transformer models, which is increasingly challenging due to its size. We argue that mixed-precision floating-point appears to be a promising datatype that provides a favorable balance between model accuracy and hardware performance for Transformer-based DNN acceleration. Currently, an automatic compilation framework that provides full stack acceleration of Transformer models is underway. The vector processing unit is also being optimized to improve non-linear function performance.

## VI. Acknowledgements

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. This work was supported by

AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR.

## References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "OPT: Open Pre-trained Transformer Language Models," *arXiv preprint arXiv:2205.01068*, 2022.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-Bert: Integer-only Bert Quantization," in *International conference on machine learning*. PMLR, 2021, pp. 5506–5518.

[5] Z. Li and Q. Gu, "I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 17 065–17 075.

[6] Y. Bondarenko, M. Nagel, and T. Blankevoort, ""Understanding and Overcoming the Challenges of Efficient Transformer Quantization"," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 7947–7969. [Online]. Available: https://aclanthology.org/2021.emnlp-main.627

[7] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore, "Efficient 8-bit Quantization of Transformer Neural Machine Language Translation Model," *arXiv preprint arXiv:1906.00532*, 2019.

[8] J. R. Stevens, R. Venkatesan, S. Dai, B. Khailany, and A. Raghunathan, "Softermax: Hardware/Software Co-Design of An Efficient Softmax for Transformers," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 469–474.

[9] N. Shazeer, "GLU Variants Improve Transformer," *arXiv preprint arXiv:2002.05202*, 2020.

[10] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "LLaMA-2: Open Foundation and Fine-tuned Chat Models," *arXiv preprint arXiv:2307.09288*, 2023.

[11] M. Song, J. Wu, Y. Ding, and H. K.-H. So, "SqueezeBlock: A Transparent Weight Compression Scheme for Deep Neural Networks," in *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023, pp. 238–243.

[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[13] W. Mao, K. Li, Q. Cheng, L. Dai, B. Li, X. Xie, H. Li, L. Lin, and H. Yu, "A Configurable Floating-Point Multiple-Precision Processing Element for HPC and AI Converged Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 213–226, 2021.

[14] AMD, "UltraScale Architecture DSP Slice User Guide (UG579)," 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp

[15] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices," *Xilinx White Paper*, 2016.

[16] J. Wu, J. Zhou, Y. Gao, Y. Ding, N. Wong, and H. K.-H. So, "MSD: Mixing Signed Digit Representations for Hardware-efficient DNN Acceleration on FPGA with Heterogeneous Resources," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2023, pp. 94–104.

[17] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance FPGA-based CNN Accelerator with Block-Floating-point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[18] C. Wu, M. Wang, X. Chu, K. Wang, and L. He, "Low-Precision Floating-Point Arithmetic for High-Performance FPGA-based CNN Acceleration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–21, 2021.

[19] H. Fan, S. Liu, Z. Que, X. Niu, and W. Luk, "High-Performance Acceleration of 2-D and 3-D CNNs on FPGAs Using Static Block Floating Point," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[20] Y. Wong, Z. Dong, and W. Zhang, "Low Bitwidth CNN Accelerator on FPGA using Winograd and Block Floating Point Arithmetic," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2021, pp. 218–223.

[21] Z. Lit, M. Sun, A. Lu, H. Ma, G. Yuan, Y. Xie, H. Tang, Y. Li, M. Leeser, Z. Wang *et al.*, "Auto-ViT-Acc: An Fpga-Aware Automatic Acceleration Framework for Vision Transformer with Mixed-Scheme Quantization," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 109–116.

[22] T. Wang, L. Gong, C. Wang, Y. Yang, Y. Gao, X. Zhou, and H. Chen, "ViA: A Novel Vision-Transformer Accelerator Based on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4088–4099, 2022.

[23] W. Ye, X. Zhou, J. Zhou, C. Chen, and K. Li, "Accelerating Attention Mechanism on FPGAs Based on Efficient Reconfigurable Systolic Array," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 6, nov 2023.

[24] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[25] J. Zhou, J. Wu, Y. Gao, Y. Ding, C. Tao, B. Li, F. Tu, K.-T. Cheng, H. K.-H. So, and N. Wong, "DyBit: Dynamic Bit-Precision Numbers for Efficient Quantized Neural Network Inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[26] A. H. Zadeh, M. Mahmoud, A. Abdelhadi, and A. Moshovos, "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*, 2022, pp. 888–901.

[27] V. Camus, L. Mei, C. Enz, and M. Verhelst, "Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 697–711, 2019.

[28] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.

[29] L. Mei, M. Dandekar, D. Rodopoulos, J. Constantin, P. Debacker, R. Lauwereins, and M. Verhelst, "Sub-word Parallel Precision-scalable Mac Engines for Efficient Embedded Dnn Inference," in *2019 IEEE international conference on artificial intelligence circuits and systems (AICAS)*. IEEE, 2019, pp. 6–10.

[30] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[31] M. K. Jaiswal and H. K.-H. So, "Area-Efficient Architecture for Dual-Mode Double Precision Floating Point Division," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 2, pp. 386–398, 2017.

[32] H. Zhang, D. Chen, and S.-B. Ko, "New Flexible Multiple-Precision Multiply-accumulate Unit for Deep Neural Network Training and Inference," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 26–38, 2019.

[33] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia Tensor Core Programmability, Performance & Precision," in *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.