FPGA-based Block Minifloat Training Accelerator for a Time Series Prediction Network

WENJIE ZHOU, University of Sydney, Australia
HAOYAN QI, University of Sydney, Australia
DAVID BOLAND, University of Sydney, Australia
PHILIP H.W. LEONG, University of Sydney, Australia

Time series forecasting is the problem of predicting future data samples from historical information and recent deep neural networks (DNNs) based techniques have achieved excellent results compared with conventional statistical approaches. Many applications at the edge can utilise this technology and most implementations have focused on inference, an ability to train at the edge would enable the deep neural network (DNN) to adapt to changing conditions. Unfortunately, training requires approximately three times more memory and computation than inference. Moreover, edge applications are often constrained by energy efficiency. In this work, we implement a block minifloat (BM) training accelerator for a time series prediction network, N-BEATS. Our architecture involves a mixed precision GEMM accelerator that utilizes block minifloat (BM) arithmetic. We use a 4-bit DSP packing scheme to optimize the implementation further, achieving a throughput of 779 Gops. The resulting power efficiency is 42.4 Gops/W, 3.1x better than a graphics processing unit in a similar technology.

CCS Concepts: • **Hardware** → **Reconfigurable logic applications**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: FPGA, Block Minifloat, Neural Network Training

1 INTRODUCTION

Field programmable gate arrays (FPGAs) have been widely used for deep neural networks (DNNs) inference applications due to their ability to implement domain-specific data paths to optimize latency and power consumption. To date, most research has focused on inference, with training done offline using server-based equipment such as graphics processing units (GPUs) and tensor processing units (TPUs). Edge-based training offers the potential for neural networks to adapt to local conditions but at present, the additional computational and hardware cost is an obstacle to its adoption.

Unfortunately, it requires significantly more computation than inference, as well as high power consumption, meaning there are far fewer implementations. Custom number systems can be used to help address this challenge. Both 8-bit [1][2], and 4-bit [3] quantization methods have been proposed, and some new number systems, such as LNS-madam [4], 8-bit floating-point (FP8) [5], and block floating-point (BFP) [6] have been designed specifically for low-precision implementation.

One recent approach is to use block minifloat arithmetic [7, 8] which involves sharing a common exponent with a block of low-precision floating point numbers (known as minifloats) to increase their range (see Figure 1(a)). While this technique is extremely promising, implementations have been limited to GPUs [7] and inference-only

Authors' addresses: Wenjie Zhou, wenjie.zhou@sydney.edu.au, University of Sydney, Australia; Haoyan Qi, University of Sydney, Australia, haoyan.qi@sydney.edu.au; David Boland, University of Sydney, Australia, david.boland@sydney.edu.au; Philip H.W. Leong, University of Sydney, Australia, philip.leong@sydney.edu.au.

ACM 1936-7414/2024/12-ART
https://doi.org/10.1145/3707209

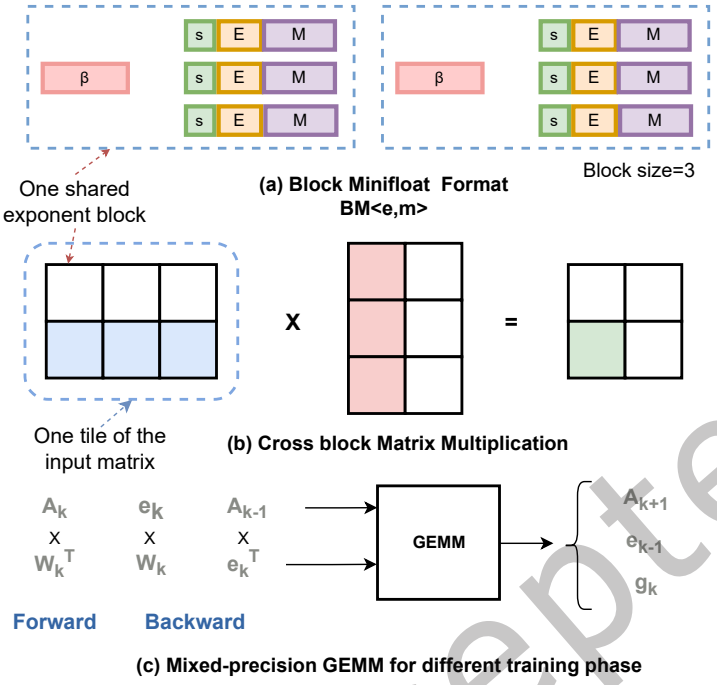


Fig. 1. Summary of the main ideas introduced in this paper. (a) Block minifloat (BM) format with block size 3. (b) cross-block matrix multiplication. The shaded parts show a single block output obtained by multiplying 1×3 blocks by 3×1 blocks. (c) general matrix multiplication (GEMM) kernel. Forward and backward passes are processed by the same GEMM accelerator at different precision.

FPGA designs [9]. In this paper, we introduce configurable block size and precision in block minifloat (BM) arithmetic to improve the training accuracy. Moreover, we propose an efficient and flexible implementation of a BM accelerator for DNN training. The design is tailored to an FPGA, and optimized to accelerate the neural basis expansion analysis for interpretable time series analysis (N-BEATS) deep neural network for time series prediction. Applications of the accelerator include anomaly detection, high-frequency trading, and sensor value prediction.

The contributions of this work can be summarised as follows:

- BM arithmetic introduces a new parameter, the block size. To achieve a balance between performance and accuracy, it should be possible to optimise block size independently of the general matrix multiplication (GEMM) tile size. We propose a novel cross-block BM GEMM unit that allows independent block and tile sizes and utilizes a higher precision buffer to improve accuracy (Figure 1(b)).
- The optimal precisions for the forward and backpropagation steps in training are different. We present a new BM GEMM kernel that supports runtime configuration of precision (Figure 1(c)).
- A novel, packed 4-bit BM MAC unit that can compute 6 independent multiplications per DSP is presented.
- We present the first FPGA implementation of 4-bit BM, mixed-precision neural network training. To the best of our knowledge, other reported FPGA implementations have used higher precision. On an Alveo U50 board, when training the N-BEATS network, we achieve a power efficiency of 42.4 Gops/W, this

being a 3.12x improvement over an Nvidia GTX 1080 GPU. To facilitate reproducible research, source code is available on Github[1].

The paper is organized as follows. Section 2 is the background; section 3 proposes the BM arithmetic and BM GEMM architecture design; in section 4, the mixed-precision BM training accelerator design for N-BEATS is proposed, section 5 is the experiment.

2 BACKGROUND

2.1 BM Number System

A BM number consists of a small floating point value (called minifloat) together with an exponent bias, β, that is shared between those BM numbers in the same block. BM supports normal and denormalized (denorm) numbers, but saturating arithmetic is employed instead of IEEE-754 overflow and underflow, Inf, and NaN. The rounding scheme used is round to nearest. Stochastic rounding is used for weight updates and contributes to the convergence of training, particularly at low precision. A minifloat representation is parameterized by the number of exponent and mantissa bits, i.e., an $\langle e, m\rangle$ minifloat format, has one sign bit, e exponent bits and m mantissa bits. Let s, E, and M be the unsigned integer representation of the values in the sign, exponent, and mantissa fields, then Equation 1 can be used to compute the real number represented.

$$f(s,E,M) = \begin{cases} (-1)^s \times g(s,M) \times 2^{1-\eta} & E = 0 \text{ (denorm)} \\ (-1)^s \times g(s,M) \times 2^{E-\eta} & E \neq 0 \text{ (normal)} \\ 0 & E = M = 0 \end{cases} \tag{1}$$

where $\eta = 2^{e-1} - 1$ is the exponent bias for the binary-offset encoded exponent, and the significand, S, is:

$$S = g(s,M) = \begin{cases} (M \times 2^{-m}) & E = 0 \text{ (denorm)} \\ (1 + M \times 2^{-m}) & E \neq 0 \text{ (normal)} \end{cases} \tag{2}$$

We also define an inverse which extracts the sign, exponent, and mantissa components from a BM representable value x:

$$(s,E,M) = f^{-1}(x) \tag{3}$$

Similar to BFP [10], the BM format [7], $BM\langle e, m\rangle$, is used to describe a submatrix (or block) X, where each element $r_i \in X$ has a shared exponent bias β:

$$r_i = f(s_i, E_i, M_i) \times 2^{\beta} \tag{4}$$

For example, the $BM\langle 2, 5\rangle$ format (sometimes written as $BM8\langle 2, 5\rangle$ to indicate the word length), has a shared bias, 2 bits to represent the exponent and 5 bits for the mantissa. The set of elements with common shared exponent bias is called a block, and the size of this set is the block size. A smaller block size has the benefit of improving training accuracy [11, 12]. The term tile size is used to describe the size of the processing element (PE) array used to form the GEMM kernel. This is denoted as $Tl \times Tl$, and should be chosen as large as possible for performance. The block size $Blk \times Blk$ is, in general, not equal to the tile size as this is chosen to control accuracy loss.

BM can be regarded as a superset of BFP, minifloat and logarithmic number systems. When $e = 0$, BM is BFP, $BM\langle 0, m\rangle$ also represents (m+1) bit BFP data with one sign bit and m bit integer; when $\beta = 0$, BM is minifloat; when $m = 0$ and $\beta = 0$, BM is a logarithmic number system; and finally when $e = 0$ and $\beta = 0$, BM is fixed point.

[1]https://github.com/charliechou1001/BlockMinifloat_training.git

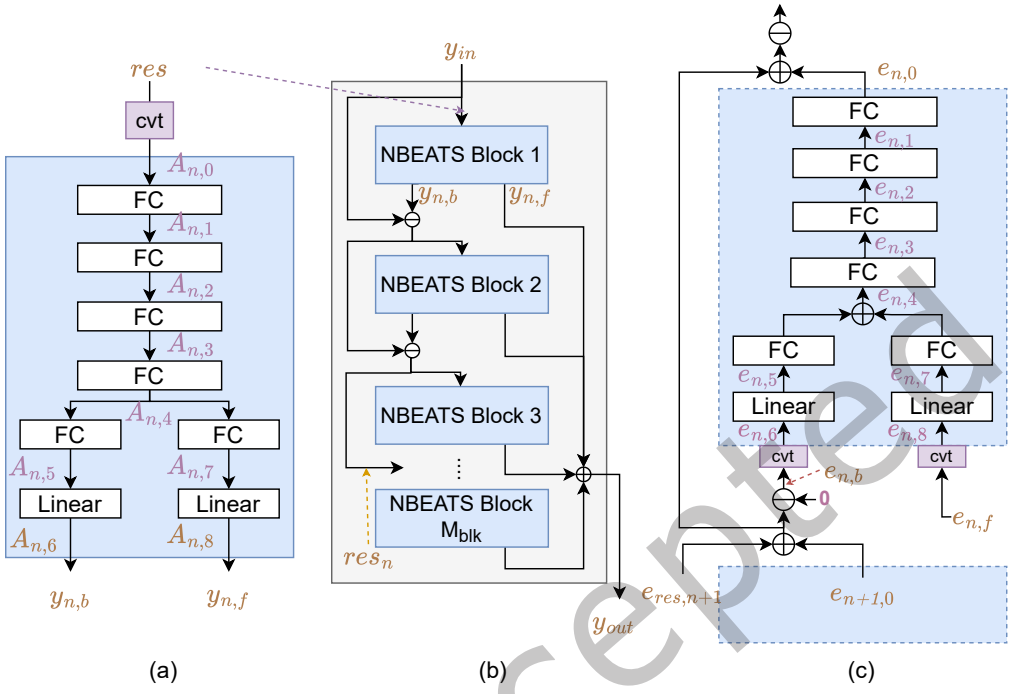


Fig. 2. N-BEATS neural network model. Variables in yellow represent high-precision BM data ($BM\langle e_h, m_h\rangle$). *Cvt* block is the precision conversion block that converts high-precision data to low-precision data. (a) shows the N-BEATS block. The input of each block is first converted to low-precision format, then goes first to the 4-layer FC stack, and then the FC stack output activation goes to the backcast branch and forecast branch. (b) shows the N-BEATS model. (c) shows the error propagation of N-BEATS block.

2.2 Forward and Backward Pass for an N-BEATS Block

N-BEATS was the first work to empirically demonstrate that pure deep learning using no time-series specific components [13]. The authors demonstrated state-of-the-art performance, improving forecast accuracy by 11% over a statistical benchmark and by 3% over the 2020 winner of the M4 competition, which used a hybrid statistical/neural residual/attention dilated long short-term memory (LSTM) stack. N-BEATS was chosen for this study due to its accuracy and its regular architecture based on fully connected layers. Our approach can be applied to recurrent neural networks (RNNs) including LSTMs or gated recurrent units (GRUs) with minor modifications.

N-BEATS is applied for discrete time series forecast tasks. Given a length observed in history $\boldsymbol{y}_{in} = [y_1, \ldots, y_{H_b}]$, the task is to forecast the future time series $\boldsymbol{y}_{out} = [y_{H_b+1}, \ldots, y_{H_b+H_f}]$.

Figure 2 illustrates the N-BEATS architecture [13], the forward pass and error propagation (backward pass) for an N-BEATS block. $\boldsymbol{y}_{in}$ is the input and $\boldsymbol{y}_{out}$ is the forecast of the network. As shown in Figure 2(b), it comprises several N-BEATS blocks, each having four fully connected (FC) layers with rectified linear unit (ReLu) activation, split into backcast and forecast branches.

The forward pass of an N-BEATS block is shown in Figure 2(a). We use n and k as block and layer indices for an N-BEATS block and the N-BEATS block indices respectively, where M_{blk} is the number of N-BEATS blocks, $n = 1, 2..., M_{blk}$, res_n is the residual vector to the nth N-BEATS block, and $A_{n,k}$ is the activation vector.

A forward pass through N-BEATS block l involves propagation through a number of linear and FC layers,

$$A_{n,k} = \begin{cases} ReLu(z_{n,k}) & k = 0, 1, ..., 5, 7 \text{ (FC)} \\ z_{n,k} & k = 6, 8 \text{ (Linear)} \end{cases} \tag{5}$$

where $z_{n,k} = A_{n,k-1} W_{n,k}^T$ is a linear projection.

$A_{n,6}$ is the backcast branch output $y_{n,b}$; $A_{n,8}$ is the forecast branch output $y_{n,f}$. $y_{n,b}$ length is the same as the block input length H_b, and $y_{n,f}$ length is the same as the model output length H_f. The intermediate result $A_{n,5}$, $A_{n,7}$ in the forecast and backcast branches has a length of $H_b + H_f$.

For the residual computation, the backcast residual res of the nth N-BEATS block and the prediction output of the model are:

$$\begin{aligned} res_n &= in - \sum_{n=1}^{n=n} y_{n,b}, \\ y_{out} &= \sum_{n=1}^{n=M_{blk}} y_{n,b} \end{aligned} \tag{6}$$

N-BEATS uses mean absolute percentage error (MAPE) loss for training and symmetric mean absolute percentage error (sMAPE) loss for validation.

$$\begin{aligned} C_{MAPE} &= \frac{1}{H} \sum_{i=1}^{H} \frac{|l_i - p_i|}{|l_i|} \\ C_{SMAPE} &= \frac{200}{H} \sum_{i=1}^{H} \frac{|l_i - p_i|}{|l_i| + |p_i|} \end{aligned} \tag{7}$$

where l_i is the actual value in time step i, and p_i is the scalar predicted value in time step i.

During training, backpropagation through each layer involves a repeated application of the chain rule [14]. The N-BEATS block error propagation is illustrated in Figure 2(c). The error of $y_{n,f}$ is defined as:

$$e_{n,f} = \frac{\partial C}{\partial y_{out}} = \begin{cases} -\frac{1}{Hl_i} & p_i \le l_i \\ \frac{1}{Hl_i} & p_i > l_i \end{cases} \tag{8}$$

the error of $y_{n,b}$ is defined as:

$$e_{n,b} = \frac{\partial C}{\partial y_{n,b}} = -(e_{\ell+1,res} + e_{\ell+1,0}) \tag{9}$$

For each FC or *Linear* layer, the propagated error is

$$e_{n,k} = \frac{\partial C}{\partial A_{n,k}} = \begin{cases} (\frac{\partial C}{\partial A_{n,k+1}} \circ \sigma'_{n,k}) \times W_{n,k} & k = 0, 1, ..., 5, 7 \\ e_{n,6} \times W_{n,k} & k = 6 \\ e_{n,8} \times W_{n,k} & k = 8 \end{cases} \tag{10}$$

where $\sigma'_{n,k}$ is the derivation of ReLu function $\frac{\partial ReLu}{\partial z_{n,k}}$. We define $(x \circ y)_{ij} = x_{ij} y_{ij}$.

Table 1. Summary of mixed-precision FC/Linear layer computation

Index	In1	In2	Eq.	Out
①	$A_{n,0}$ $BM\langle e_{in}, m_{in}\rangle$	$W^T_{n,1}$ $BM\langle e_w, m_w\rangle$	(5)	$A_{n,1}$ $BM\langle e_{act}, m_{act}\rangle$
②	$A_{n,k-1}$ $BM\langle e_{act}, m_{act}\rangle$	$W^T_{n,k}$ $BM\langle e_w, m_w\rangle$	(5)	$A_{n,k}$ $BM\langle e_{act}, m_{act}\rangle$
③	$A_{n,k-1}$ $BM\langle e_{act}, m_{act}\rangle$	$W^T_{n,k}$ $BM\langle e_w, m_w\rangle$	(5)	$y_{n,b}$ *or* $y_{n,f}$ $BM\langle e_h, m_h\rangle$
④	$e_{n,k+1} \circ \sigma'_{n,k}$ $BM\langle e_e, m_e\rangle$	$W_{n,k}$ $BM\langle e_w, m_w\rangle$	(10)	$e_{n,k}$ $BM\langle e_e, m_e\rangle$
⑤	$e_{n,1} \circ \sigma'_{n,1}$ $BM\langle e_e, m_e\rangle$	$W_{n,1}$ $BM\langle e_w, m_w\rangle$	(10)	$e_{n,0}$ $BM\langle e_h, m_h\rangle$
⑥	$(e_{n,k})^T$ $BM\langle e_e, m_e\rangle$	$A_{n,k-1}$ $BM\langle e_{act}, m_{act}\rangle$	(11)	$g_{n,k}$ $BM\langle e_g, m_g\rangle$

The weight gradient for each FC layer and *Linear* layer is computed as follows.

$$g_{n,k} = \frac{\partial C}{\partial W_{n,k}} = (e_{n,k})^T \times A_{n,k-1} \tag{11}$$

Weight updates are done using Stochastic Gradient Descent (SGD) with learning rate α:

$$W'_{n,k} = W_{n,k} - \alpha \cdot \frac{\partial C}{\partial W_{n,k}} \tag{12}$$

In the preceding description, forward and backward passes were given for a single input vector. In our actual implementation, inputs are processed in minibatches, with batch size B.

The forward and backward passes are mostly matrix multiplications as summarized in Table 1. The same table provides notation for the BM format applied to inputs, intermediate values, and outputs. Here $BM\langle e_h, m_h\rangle$ is high-precision format, others are low-precision format.

2.3 Stochastic Rounding

Stochastic rounding is used for weight updates [6, 15] to improve convergence in training and implemented as:

$$SRound(r) = \begin{cases} (-1)^s \times g(s, M+1) \times 2^{1-\eta} \times 2^{\beta} & 0 \le rand \le (M \times 2^{-m}) \\ (-1)^s \times g(s, M) \times 2^{1-\eta} \times 2^{\beta} & (M \times 2^{-m}) < rand \le 1 \end{cases} \tag{13}$$

Here *rand* is the random number between 0 and 1. In hardware, it's implemented with a linear feedback shift register (LFSR) generator. Thus SGD is implemented as:

$$W'_{n,k} = W_{n,k} - \alpha \cdot SRound(\frac{\partial C}{\partial W_{n,k}}) \tag{14}$$

3 IMPLEMENTATION OF BM ARITHMETIC

3.1 Cross Block BM Matrix Multiplication

Previous work [5, 16] used 32-bit floating-point (FP32) format as the high-precision accumulator and compute the inner-product in conventional floating-point (FP) process. In this work, we choose integer-based implementation, as described in Figure 3. Compared to FP based accumulation, it has fewer steps with a shorter latency than

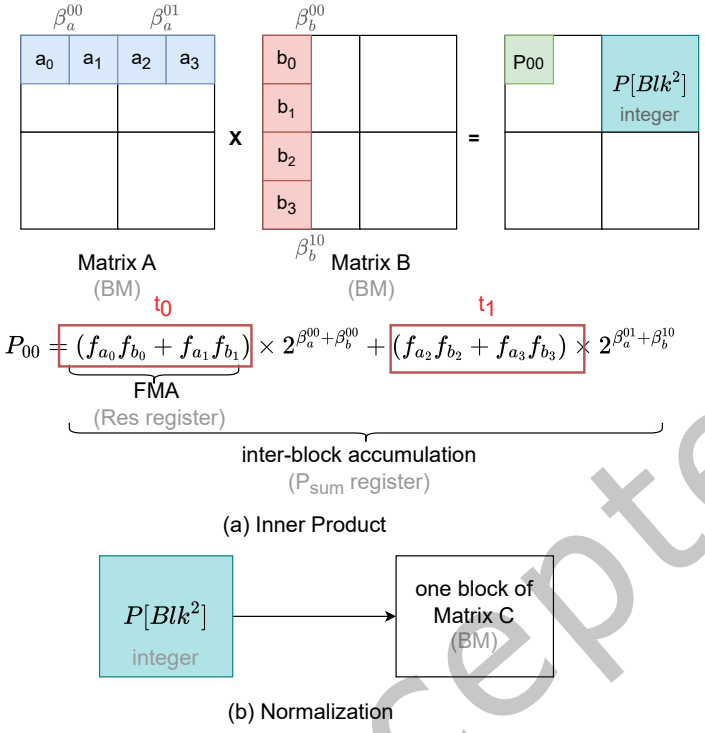


Fig. 3. Simplified example of a BM GEMM requiring inner-product and post-inner-product (normalization) computations. The matrices are 4×4 and each divided into 4, 2×2 blocks. (a) Illustrates the fused multiply accumulate (FMA), the inputs are converted to integer, multiplied and added together to form $t_0 = f_{a_0} f_{b_0} + f_{a_1} f_{b_1}$ and $t_1 = f_{a_2} f_{b_2} + f_{a_3} f_{b_3}$. In the inter-block accumulation step, the t_i values are aligned according to the shared exponent values and summed to give a high-precision inner-product P_{00}. The P_{ij} values for a block are saved in a wide integer buffer called $P[Blk^2]$. (b) The normalization process is executed after all the inner products required for a particular block have been completed. The wide integer results are converted into BM format data and form part of the output matrix C.

FP operation [8], and integer-based mixed precision implementation is less complicated than FP based mixed precision one.

Figure 3 provides a simplified example of a BM matrix multiplication. All inputs and outputs are in BM format, and it accepts two input matrices, A and B, and computes an output matrix C. P is a two-dimensional wide integer buffer of size $Blk \times Blk$ and holds an entire block of C. In cross-block matrix multiplication, matrices A and B are partitioned into multiple blocks of size $Blk \times Blk$, and each block has a shared exponent bias β_a^{uw} and β_b^{wv}, where u, v, w are block indices. The minifloat elements in A, a_i, are in $BM\langle e_a, m_a\rangle$ format, and the minifloat elements in matrix B are b_i.

3.1.1 Computation Process. A block diagram of our implementation is illustrated in Figure 4 and detailed in the following paragraphs.

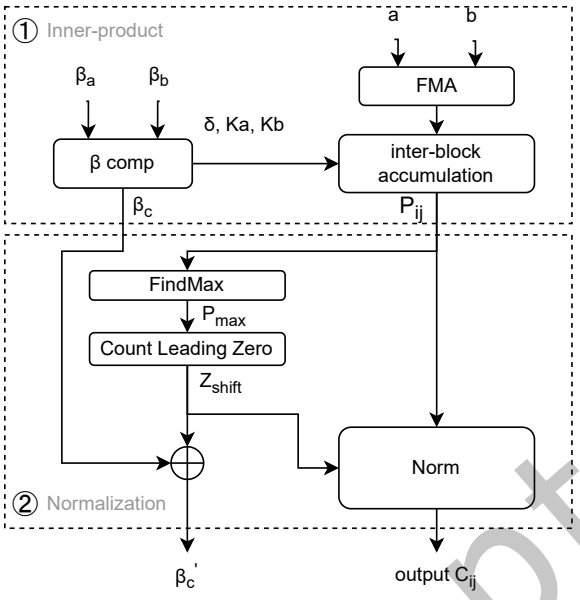


Fig. 4. Block diagram illustrating cross block BM matrix multiplication. The first step performs the minifloat fused multiply accumulate (FMA) (FMA block) and shared exponent computation (β comp block), placing the result in $P[Blk^2]$. In the second step, each P_{ij} in $P[Blk^2]$ is examined to determine the appropriate shift for normalization, which is performed in the $Norm$ block. The resulting C output is in BM format.

Inner Product. The cross block inner-product requires accumulation of fused multiply accumulate (FMA) results (see Figure 3) and can be expressed mathematically as:

$$P_{ij} = \sum_{w} \{ \sum_{t=0}^{Blk-1} \text{fma}(A, B, t) \times 2^{\beta_a^{uw} + \beta_b^{wv}} \} \tag{15}$$

where $\text{fma}(A, B, t)$ is the FMA result for block t, and P_{ij} is an element of the accumulation output block $P[Blk^2]$. During FMA computation, a_i and b_i are converted into integers first and then doing multiply accumulate (MAC) computation.

The β-comp block computes the shared exponent for the output block β_c^{uv} and the bias values Ka and Kb of the shared exponent used in inter-block accumulation.

Normalization. The normalization process converts the integer accumulation result into minifloat format data output. The accumulation results in $P[Blk^2]$ are wide integers in value and in general will exceed the dynamic range of the output minifloat representation. We find the maximum absolute value P_{max} of $P[Blk^2]$ and use it to determine a shift value, Z_{shift}, that avoids overflow.

Z_{shift} is used to adjust the output shared exponent and $Norm$ block during conversion of the P_{ij} values in $P[Blk^2]$ to BM format in C_{ij}. The detail is provided in Algorithm 1. $Cnt_{overflow}$, Cnt_{denorm} and $Cnt_{underflow}$, EXP_{dp} are constant boundary thresholds for normalization.

3.1.2 Error Analysis. The source of the inner-product computational error has two components: one is the rounding error in normalization, and the other one is the error in inter-block accumulation. We normalize the

Algorithm 1: Accumulation buffer to $BM\langle e, m\rangle$ normalization

Function *Norm*(P_{ij}, Z_{shift})
- $s \leftarrow (P_{ij} > 0)\ ?\ 0 : 1$; // Sign bit
- $P_{ij} \leftarrow |P_{ij}| \gg Z_{shift}$;
- $Z \leftarrow CLZ(P_{ij})$; // Check leading zeros
- **if** $Z < Cnt_{denorm}$ **then**
  - // denormal and underflow
  - $C_{ij} \leftarrow f(s, 0, P_{ij} \ll (Cnt_{denorm} - 1))$;
- **else**
  - // normal
  - $E \leftarrow EXP_{dp} - (Z + 1) + \eta_c$;
  - $C_{ij} \leftarrow f(s, E, P_{ij} \ll Z)$;
- return C_{ij}

P_{sum} only once after the inner product finishes. In inter-block accumulation, the aligned addition may bring truncation error in the tailing bits for the smaller addend value. P_{sum} is designed to have sufficient word length to ensure that C_{ij} does not overflow. The word length of P_{sum} is defined as follows:

$$Kadd = 1 + (2^{e_a} - 1 + m_a) + (2^{e_b} - 1 + m_b) + W_{ex} + W_{tail} \tag{16}$$

Here $Kadd$ is the word length of P_{sum}, W_{ex} is extra bits to avoid overflow, and W_{tail} is extra tailing bits to hold more bits in aligned addition.

3.1.3 Mixed Precision. The mixed precision computation process is the same as the single precision described in Figure 4, except for some changes in input and output. For input data, an extra mixed precision decoder is applied when converting a_i and b_i to integers according to different precision configurations. During normalization, the normalization process still follows Algorithm 1, but there are several selectable boundary thresholds used to convert the integer accumulation result to different target precision outputs.

3.2 BM Vector Addition

The BM vector addition is used in residual computation, weight update, and error addition. The error addition is the back propagation of the branch structure inside N-BEATS blocks. The BM addition is also conducted in integer format and computed as follows:

$$a \pm b = \begin{cases} (S_a 2^{E_a} \pm S_b 2^{E_b} * 2^{\beta_B - \beta_A}) * 2^{\beta_A}, \beta_A \geq \beta_B \\ (S_a 2^{E_a} * 2^{\beta_A - \beta_B} \pm S_b 2^{E_b}) * 2^{\beta_B}, \beta_A < \beta_B \end{cases} \tag{17}$$

where S_a, S_b are the significands of a and b (Equation 2).

The addition process is summarized in Figure 5. The input a and b are converted to integers, aligned with the bias of shared exponent $|\beta_B - \beta_A|$ according to Equation 17, and added. The output is the signed integer R_{ij} in block $R[Blk^2]$, and the output shared exponent $\beta_c = max(\beta_a, \beta_b)$. Then, the maximum absolute value R_{max} of $R[Blk^2]$ is checked for overflow and used to calculate the Z_{shift} value if overflow occurs. Finally, The *ADD Norm* converts integer value R_{ij} to BM format output c_{ij}.

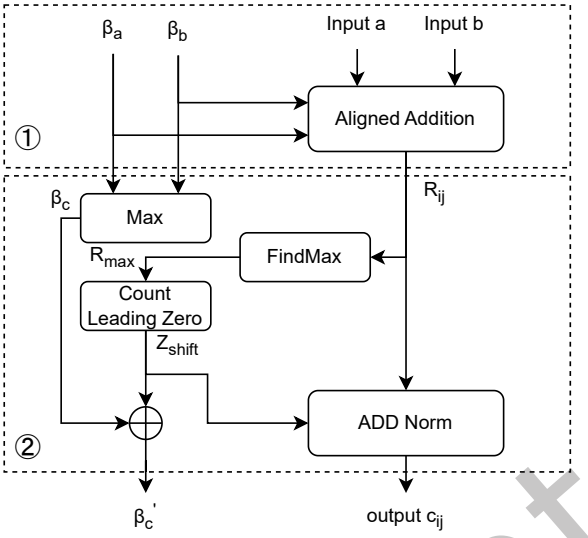


Fig. 5. BM vector addition arithmetic. The first step converts the inputs a and b into integers, aligns and sums. The output R_{ij} is placed in signed integer block $R[Blk^2]$. The second step computes the output shared exponent β_c. If there is overflow, it calculates the Z_{shift} value and adjusts β_c, then the norm block converts integer values in $R[Blk^2]$ into BM format.

3.3 BM GEMM kernel Architecture

The BM GEMM kernel, illustrated in Figure 7, is a 2D output-stationary systolic array [17]. It receives parallel input blocks A and B in minifloat format, as well as their shared exponents β_a and β_b.

3.3.1 PE Design. As shown in Figure 6, the PE performs the scalar inner product, described in Figure 4. The input formats for a and b can be different minifloat formats, configurable at run time. The first step is the mixed precision decoder. The second step is FMA computation, and step 3 is inter-block accumulation. δ is the bias of shared exponent between two blocks, which is calculated in β-comp block. After the inner product computation finishes, P_{ij} for each PE is streamed out of the PE array using the shift register.

3.3.2 BM GEMM. The GEMM has $Tl \times Tl$ PE blocks in PE array. The tile size Tl of the GEMM is sized to get the best computation performance while constrained by on-chip resources, while block size Blk is set small to get better training accuracy. The GEMM design supports configurable block size under the assumption that the block size is smaller than or equal to the tile size. The example in Figure 7 has a block size of $\frac{Tl}{2} \times \frac{Tl}{2}$. The PE array computes the inner product of a tile matrix. The β block in PE array corresponds to the β comp block in Figure 4.

After the inner-product computation finishes, the P_{sum} in each PE is pipelined out and written to the $P[Blk^2]$ buffer. The shared exponent result is also pipelined out through β blocks. $P[Blk^2]$ buffer is a ping-pong buffer. Then the FindMax computation is done in parallel by streaming P_{sum} through columnar $FindMax$ units and reduced by shifting to the leftmost column to obtain a single result P_{max}. $Calibr$ block calculates the shared exponent result $\beta_c^{'}$ and Z_{shift}. Finally, the $Norm$ block converts P_{ij} values in $P[Blk^2]$ buffer into BM output data C.

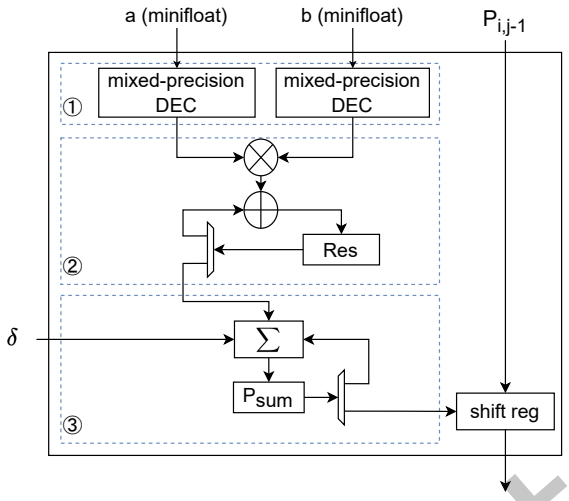


Fig. 6. PE design of GEMM kernel. The PE is responsible for the scalar inner-product computation in Figure 3.

3.3.3 GEMM Pipeline Organization. As shown in Figure 8, execution is divided into four stages. If the GEMM input matrix sizes are $Row \times K$ and $K \times Col$, the latency of naive implementation is approximately:

$$l_{naive} = \left\lceil \frac{Row \times Col}{Tl \times Tl} \right\rceil \times (K + 3 \times Tl) \tag{18}$$

The pipeline is optimized in two ways. Firstly, the inner product and normalization computations of the two tiles have no data dependency and execute in parallel. Secondly, inside the normalization block (steps 2-4 in Figure 8), there is also no data dependency between different accumulation result blocks vertically in PE array. As shown in Figure 8(b), each stage inside the normalization is pipelined, which can reduce latency to $2Blk + Tl$. Thus the inner product computation can be pipelined with normalization. The latency of the pipelined BM GEMM is:

$$l_{pipeline} = \left\lceil \frac{Row \times Col}{Tl \times Tl} \right\rceil \times K + 2Blk + Tl \tag{19}$$

3.4 DSP Packing for 4-bit BM Training

To increase the computational density of training, we optimize the arithmetic implementation with DSP packing for 4-bit BM by packing multiply operation. We propose a DSP packing scheme and DSP packed PE design. In contrast to previous works for inference [18–21], our DSP packing technique supports different precisions during training.

3.4.1 DSP Packed PE Design. The DSP packed PE design for 4-bit BM is illustrated in Figure 9. Each PE takes two input a_i values and three input b_is. There are six MAC units for one PE, and each PE uses one DSP. In the MAC unit implementation, due to its small data word length, some computations such as exponent addition, mixed precision decoder, and sign bit xor operation are implemented as bit operations using lookup table (LUT) resources.

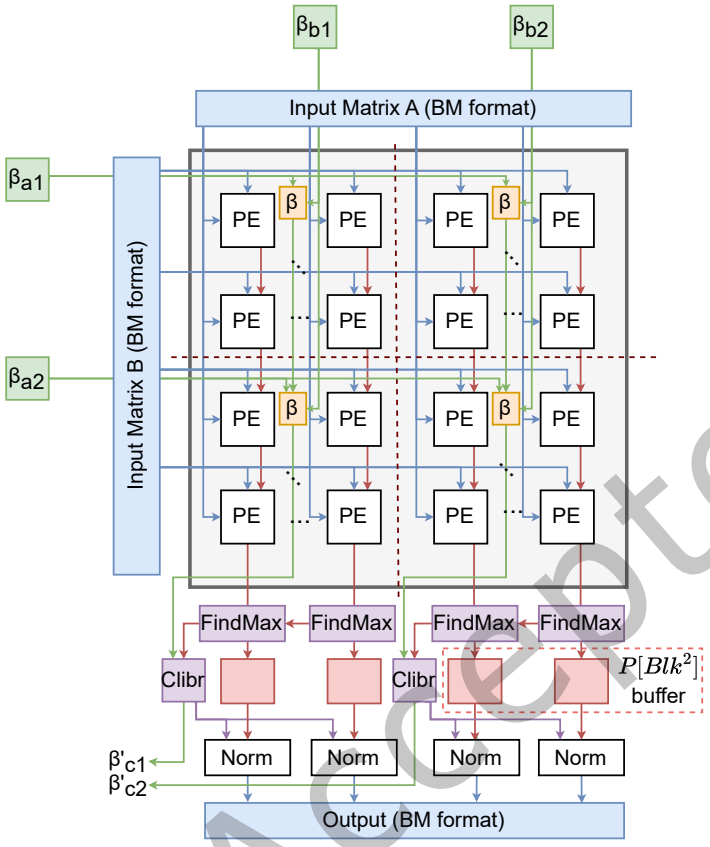


Fig. 7. Block size configurable GEMM kernel with tile size $Tl \times Tl$ ($Tl = 4$). The red dashed lines represent the boundaries of the block, and the block size in this example is $\frac{Tl}{2} \times \frac{Tl}{2}$. The GEMM kernel is the implementation of Figure 4

Our approach for the case of 4-bit BM inner product is to pack six integer operations in a single DSP as illustrated in Figure 10. Multiplication of significands is performed on the DSP, and the remaining parts, such as addition and shifting, are implemented with LUTs.

In our implementation of N-BEATS, the weight, error, and gradient use a 4-bit BM format. Figure 10(a) enumerates all possible configurations for the 4-bit BM format data. In particular, significand lengths of 1 bit to 3 bits need to be supported.

A maximum of 3 bits is needed for one of the significands of a 4-bit BM input, and the multiplier output is 6 bits. To increase accuracy, we use unsigned $BM\langle 0, 4\rangle$ for activations $A_{n,k}$ since it is the output of ReLu and non-negative; in this case, the multiplication needed is 4×3 bit, and the output is 7 bits.

As illustrated in Figure 10(b), input A of the DSP is used for two multiplier inputs and has a maximum word length of 25 bits. Input B is used for the other three inputs and has a maximum word length of 18 bits. The output, C, is populated with six 7-bit results. The total word length needed to support our packing scheme is 25×18 bits; this can fit in a Xilinx DSP48E2, which supports 27×18 bit multiplies.

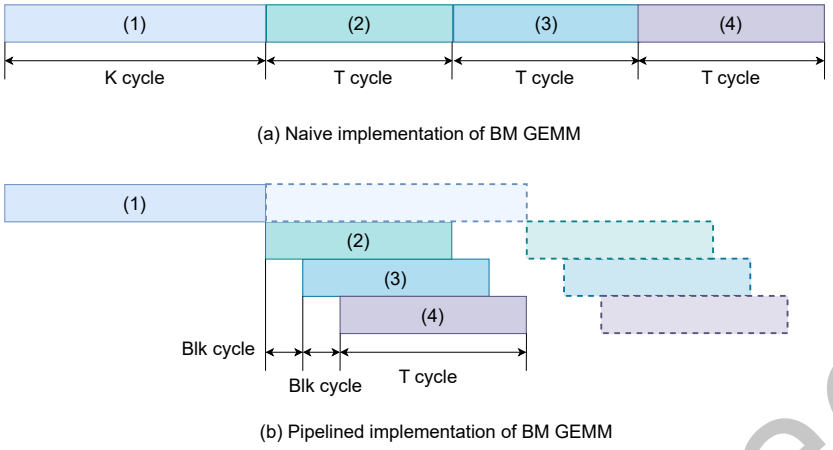


Fig. 8. Pipeline organization for BM GEMM. (a) is a naive implementation, (b) is the fine-grained pipeline for BM GEMM. (1) is inner product; (2) is Find Max in each column, and stream P_{sum} to $P[Blk^2]$ buffer; (3) is Find Max P_{max} in each block by left shifting, and calculate Z_{shift}, β_c' in $Clibr$ block; (4) is $Norm$.

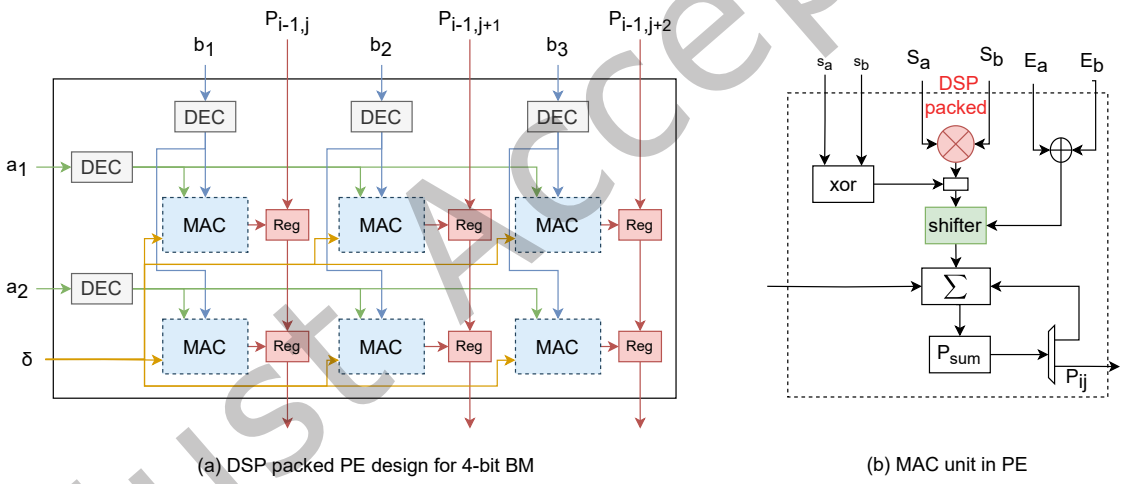


Fig. 9. DSP packed PE design for 4-bit BM MAC. (a) is the PE design, DEC block is the mixed precision decoder, and Reg block is the shift register to pipeline out the accumulation result P_{ij}. (b) is the MAC unit design.

The MAC unit, used in each PE, is illustrated in Figure 9. Figure 10(c) described how the inner product is computed. The implementation still follows step one in Figure 4, but it merges the FMA and inter-block accumulation computation.

3.4.2 Normalization and Rounding. The normalization and rounding of DSP packed GEMM is the same as the conventional BM GEMM, and the GEMM architecture also uses the architecture in Figure 7.

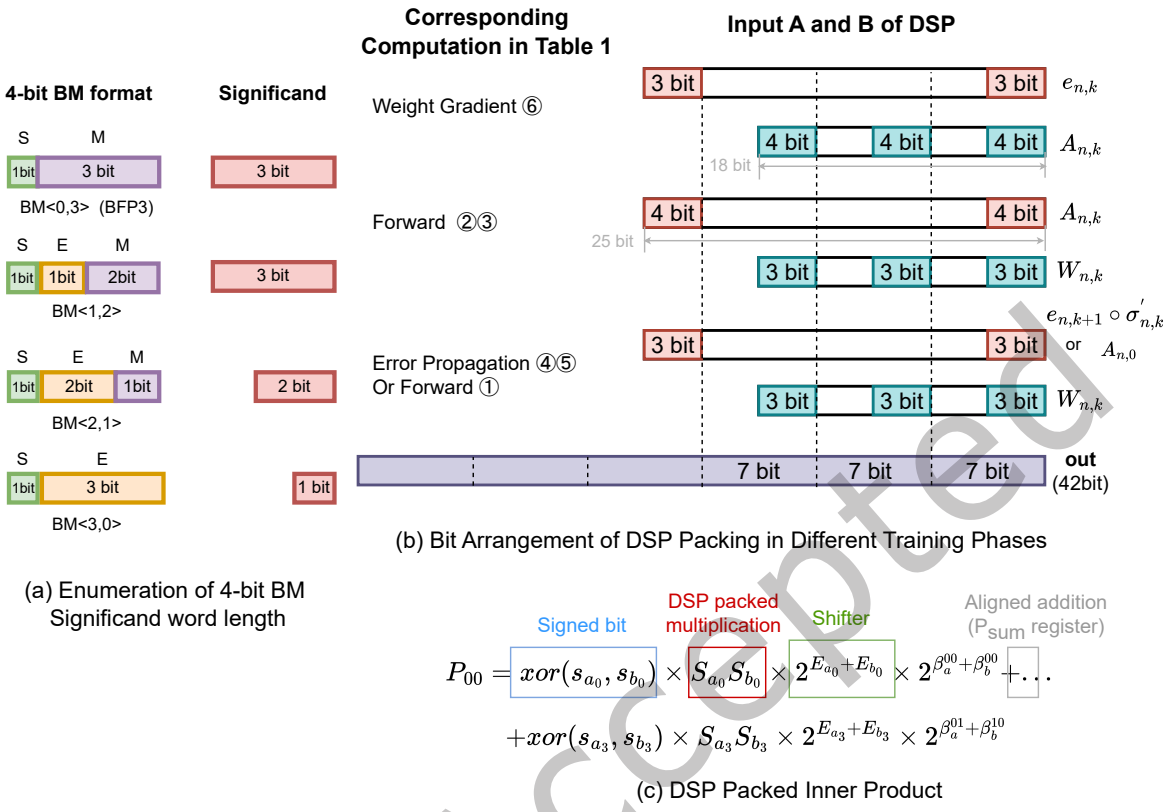


Fig. 10. DSP packing scheme for mixed precision 4-bit BM MAC. (a) is the bit arrangement for DSP packing in different training phases. The total number of multiplications is six, with two numbers in input A, three numbers in input B, and six 7-bit numbers in output C. (b) is the simplified example of DSP-packed inner product implementation. Based on Figure 3, the inner product merges the FMA and inter-block accumulation as one step.

4 NBEATS TRAINING ACCELERATOR

4.1 Training Accelerator Design

A block diagram of the N-BEATS training accelerator system is given in Figure 11. Weights, activations (used for back propagation), input data, and label values are stored in high bandwidth memory (HBM). Four separate HBM interfaces are used for independent access to weights, activations, input/label data, and the shared exponent for input/label data. Shared exponents of weight and activations are stored in FPGA BRAM. The training implementation algorithm is provided in Algorithm 2. $FC\_forward()$ computes the FC layer forward pass, $FC\_backward()$ computes the error propagation, gradient computation, and weight update of FC layer in the backward pass. FC block computes $FC\_forward()$ and Error, Gradient computation of $FC\_backward()$ mentioned in Algorithm 2. Weight update of $FC\_backward()$ is computed in SGD block.

The on-chip buffer sizes and precisions are detailed in Table 2. In this table, the largest FC layer weight size in N-BEATS is $L_k \times L_k$, where L_k is an N-BEATS size parameter specified in Section 5.4. The LP buffer stores the low-precision output of the FC block (Figure 12). HP buffer stores high-precision output ($BM\langle e_h, m_h \rangle$) from the FC block, such as residual values. In forward pass, the Residual buffer RES_{bf} is used to store the backcast and

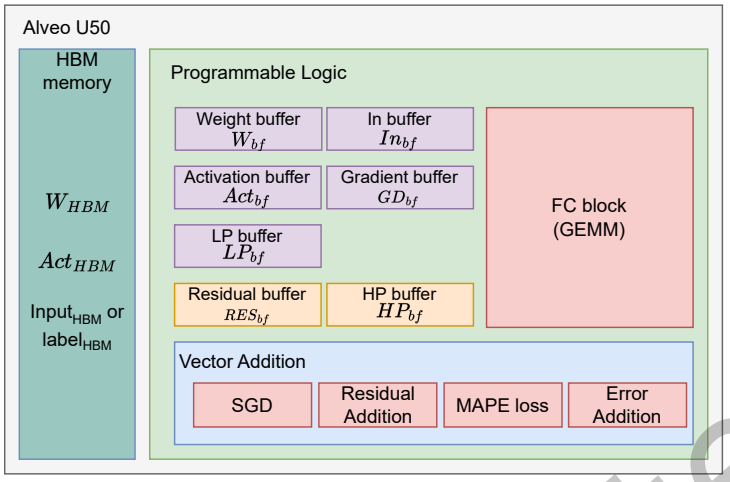


Fig. 11. N-BEATS training accelerator. The computing block is marked with red; the low-precision BM buffer is marked purple; the high-precision BM buffer is marked yellow. FC block is illustrated in Figure 12

forecast residual values, $RES_{bf}[0]$ and $RES_{bf}[1]$ in Algorithm 2 represents different area of the buffer to store backcast and forecast branch residual data. In back-propagation, the Residual buffer stores the error of two branch residuals. forward and backward passes are done layer-by-layer with weights, activations, and error values of each layer being loaded to the appropriate on-chip buffers from HBM; updated, and written back to HBM.

The *ResADD* block does the residual addition. The In_{bf} buffer stores the input values for the FC block, which are activations for forward and errors for backward. $In_{bf}[0]$ and $In_{bf}[1]$ mentioned in Algorithm 2 is the different In_{bf} area for backcast and forecast activation/error.

4.2 FC Block

The FC block is shown in Figure 12. FC block accelerates the matrix multiplication described in Table 1, and its computation is illustrated in Figure 13. This block feeds parallel data input from the *In* buffer and the parallel data from the weight or activation buffer to the GEMM kernel. $feederA$ and $feederB$ pre-process this input data by transposing it when necessary for different training phases. The transpose block is a shift register array, which can simultaneously read one tile of the matrix and feed another transposed tile data to the GEMM kernel.

The BM GEMM in FC block is the GEMM kernel illustrated in Figure 7. $feederA$ and $feederB$ are configured with different paths for different training computations. Path ① is the forward computation (Equation 5). In forward (Figure 13(a)), the activation input multiplies with transposed weight and then calculates the activation output. During forward computation, each activation input is saved to HBM for gradient computation, and the derivation of ReLu is calculated for backward computation.

Path ② is the error propagation (Equation 10). In error propagation (Figure 13(b)), the error multiplies with weight and gets the previous layer error. If it is the error of block input $e_{n,0}$, the GEMM kernel outputs the high precision BM data.

path ③ is the gradient computation (Equation 11). In weight gradient computation (Figure 13(c)), the activation is read from HBM and then transposed and multiplied with the transposed error. The output of the GEMM kernel is the transposed gradient. The output gradient is transposed so that each parallel data in the tile can compute weight update directly with weight without transposition, but the tile fetch order is different.

Algorithm 2: Forward and Back Propagation of N-BEATS block

Function *NBEATS_training*
- $RES_{bf}[0] \leftarrow Input_{HBM}$;
- **repeat** M_{blk} **times**
  - NBEATS_block_forward();
- $RES_{bf}[1] \leftarrow MAPE(label_{HBM},\ RES_{bf}[1])$; //Equation 7
- **repeat** M_{blk} **times**
  - NBEATS_block_backward();

Function *NBEATS_block_forward*
- $In_{bf}[0] \leftarrow Cvt(RES_{bf}[0])$;
- $Act_{HBM} \leftarrow write(In_{bf}[0])$;
- **repeat** *4* **times**
  - FC_forward(); //4-FC layer
- **if** *not last block* **then**
  - **repeat** *2* **times**
    - FC_forward(); //backcast branch
  - $RES_{bf}[0] \leftarrow RES_{bf}[0] - HP_{bf}$; //Equation 6
- **repeat** *2* **times**
  - FC_forward(); //forecast branch
- $RES_{bf}[1] \leftarrow RES_{bf}[1] + HP_{bf}$; //Equation 6

Function *NBEATS_block_backward*
- $In_{bf}[1] \leftarrow Cvt(RES_{bf}[1])$;
- **repeat** *2* **times**
  - FC_backward(); //forecast branch
- **if** *not last block* **then**
  - $In_{bf}[0] \leftarrow Cvt(-RES_{bf}[0])$; //Equation 9
  - **repeat** *2* **times**
    - FC_backward(); //backcast branch
- $In_{bf}[0] \leftarrow Error\ Addition(In_{bf}[0], In_{bf}[1])$;
- **repeat** *4* **times**
  - FC_backward(); //4-FC layer
- **if** *not first block* **then**
  - $RES_{bf}[0] \leftarrow RES_{bf}[0] + HP_{bf}$;

The *ReLu&Norm* block fuses the *Norm* block in Figure 7 and ReLu block. The differential value of ReLu σ' is calculated in the forward pass and used in the backward. The data output has two modes: high precision ($BM\langle e_h, m_h\rangle$) connected to the *HP* buffer, and low-precision connected to the *LP* buffer.

Table 2. On-chip buffer size and precision of the training accelerator

	Size	Precision
Weight buffer	$L_k \times L_k$	$BM\langle e_w, m_w \rangle$
Activation buffer	$B \times L_k$	$BM\langle e_{act}, m_{act} \rangle$
In buffer	$2 \times B \times K_k$	$BM\langle e_{act}, m_{act} \rangle / BM\langle e_e, m_e \rangle / BM\langle e_{in}, m_{in} \rangle$
LP buffer	$B \times L_k$	$BM\langle e_{act}, m_{act} \rangle / BM\langle e_e, m_e \rangle / BM\langle e_g, m_g \rangle$
HP buffer	$B \times L_k$	$BM\langle e_h, m_h \rangle$
Gradient buffer	$L_k \times L_k$	$BM\langle e_g, m_g \rangle$
Residual buffer	$B \times (H_b + H_f)$	$BM\langle e_h, m_h \rangle$

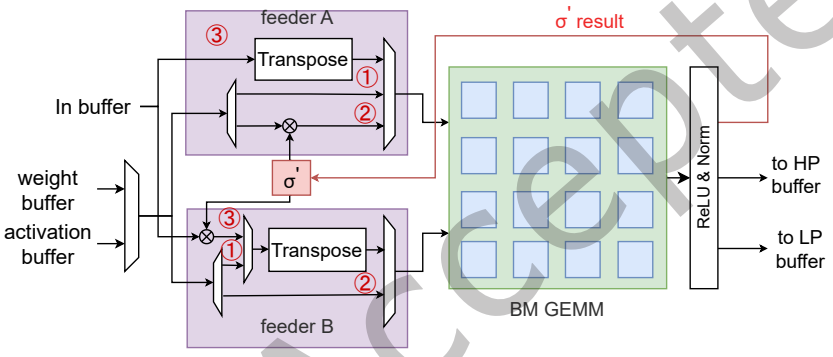


Fig. 12. FC block diagram. FeederA and FeederB have multiple paths for different training phases. ① is the forward propagation; ② is error propagation; ③ is gradient computation. The BM GEMM is illustrated in Figure 7, but the *norm* block in the GEMM is changed to *ReLU&Norm*.

5 RESULTS

5.1 Experimental setting

We evaluated the performance of our accelerator on a Xilinx Alveo U50 board. The design is written in Vitis HLS 2021.2 with a target frequency of 200 MHz. The BM computations are implemented using the HLS *ap_int* or *ap_uint* data types as primitives.

To evaluate accuracy and performance, we used the same model architecture and parameters as the original N-BEATS paper [13]. The model used M_{blk}=30 N-BEATS blocks, with the weight matrix size $L_k = 512$. Input data is arranged as a 2D array of dimension $B \times H_b$ where the batch size is $B = 1024$ and a forecast horizon H_f is 6. The backcast length is $H_b = 2H_f$, and $\theta = H_f + H_b$.

Experiments include training and validation on the M4 benchmark of the M4-Yearly dataset [22]. The M4 dataset was used to verify that the BM number system achieved minimal loss in accuracy. The N-BEATS model itself has been shown to achieve excellent accuracy over a wider set of benchmarks including the M3, M4, and TOURISM datasets [22]. Our proposed method can also be applied to other problems, and the throughput is not dependent on the dataset.

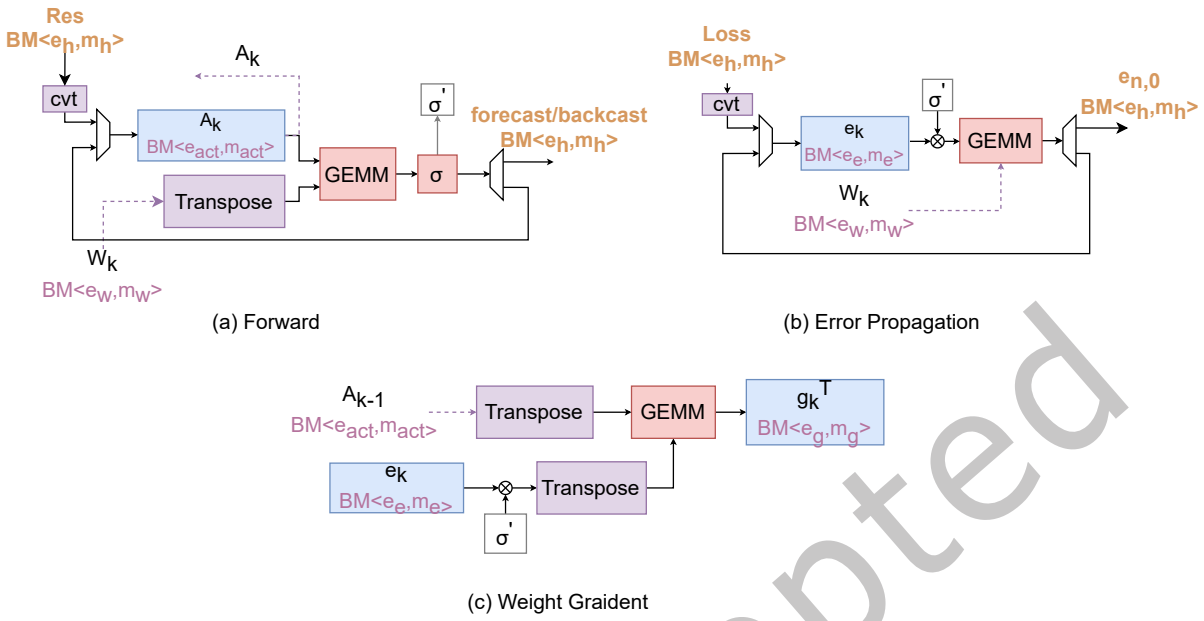


Fig. 13. Illustration of the forward and backward computation steps of an FC block. The dashed line represents the data that needs to be read/written between HBM memory and the on-chip buffer. The blue box represents the data stored in the on-chip buffer; red box is the computing blocks.

Table 3. Precision configurations used in this work

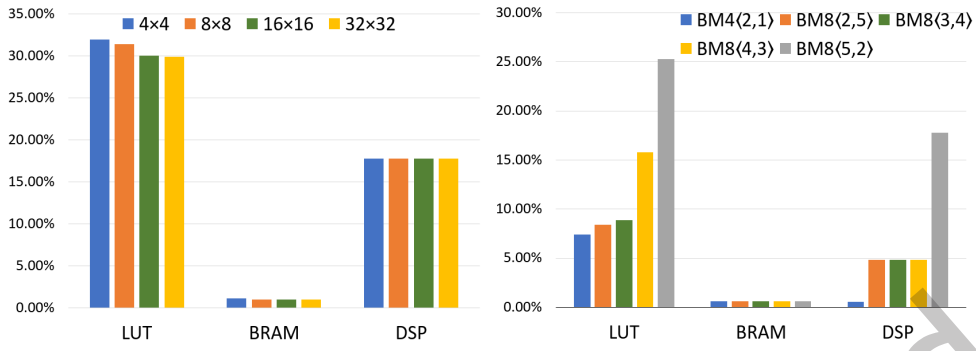
	$BM\langle e_{in}, m_{in}\rangle$	$BM\langle e_w, m_w\rangle$	$BM\langle e_{act}, m_{act}\rangle$	$BM\langle e_e, m_e\rangle$	$BM\langle e_g, m_g\rangle$	$BM\langle e_h, m_h\rangle$
BM8-uniform	$BM\langle 0, 7\rangle$	$BM\langle 0, 7\rangle$	$BM\langle 0, 7\rangle$	$BM\langle 0, 7\rangle$	$BM\langle 0, 7\rangle$	$BM\langle 0, 15\rangle$
BM4-mixed	$BM\langle 0, 3\rangle$	$BM\langle 2, 1\rangle$	unsigned $BM\langle 0, 4\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 15\rangle$
BM4-uniform-1	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 15\rangle$
BM4-uniform-2	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$	$BM\langle 0, 3\rangle$

The GPU selected for comparison is an Nvidia GTX 1080. This was chosen because both this and the Alveo U50 were fabricated in a Taiwan Semiconductor Manufacturing Company (TSMC) 16 nm FinFET process. On the GPU, FP32 precision was used without quantization, and power was measured using the nvidia-smi tool. Throughput was calculated from the time taken to train the model from scratch using Pytorch.

5.2 BM GEMM Area Exploration

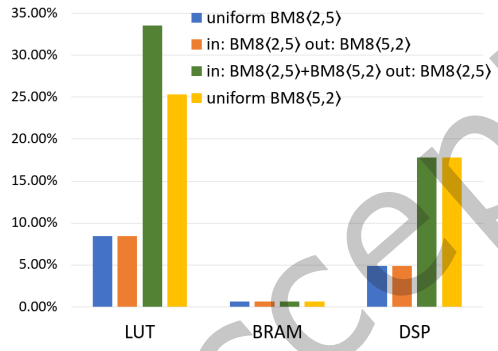
Figure 14 shows the area comparison of BM GEMM under different configurations; no DSP packing is applied in the implementation. Figure 14(a) illustrates block size configuration impact. In this experiment, the GEMM has 32×32 PEs, and data precision is set to uniform $BM8\langle 2, 5\rangle$. It shows that the impact of block size on area is very small, with a small increase in LUT consumption as the block size becomes small.

Figure 14(b) shows the impact of the BM parameters on resource utilization for a GEMM with 16×16 PEs, and block size 8×8; the input and output data precision in each $\langle e, m\rangle$ configuration are the same. In $BM8$ number



(a) Block size configuration

(b) $\langle e, m \rangle$ configuration



(c) Mixed precision configuration

Fig. 14. BM GEMM area comparison under different configuration

system, the hardware resources required for $BM8\langle 2,5\rangle$ and $BM8\langle 3,4\rangle$ are similar. $BM8\langle 4,3\rangle$ has similar DSP and BRAM usage but LUT consumption is doubled, and $BM8\langle 5,2\rangle$ has approximately triple the LUT and quadruple the DSP consumption than $BM8\langle 2,5\rangle$. Thus $BM8\langle 2,5\rangle$ and $BM8\langle 3,4\rangle$ are preferred from an area perspective over $BM8\langle 4,3\rangle$ and $BM8\langle 5,2\rangle$ configurations. Compared to $BM8$, the MAC units in $BM4\langle 2,1\rangle$ configuration do not require DSPs, and the LUT usage of the GEMM is smaller than $BM8\langle 2,5\rangle$.

Figure 14(c) shows the resource impact of the mixed precision configuration. In this design, GEMM kernel has 16×16 PEs, and the size is 8×8. The GEMM kernel for a uniform $BM8\langle 2,5\rangle$ configuration has the same area as the $BM8\langle 2,5\rangle$ input and $BM8\langle 5,2\rangle$ output. The mixed precision design, supporting $BM8\langle 2,5\rangle + BM8\langle 5,2\rangle$ inputs has approximately a 30% LUT overhead compared with uniform $BM8\langle 5,2\rangle$ GEMM. The resource utilization of a configurable GEMM is determined by the largest minifloat exponent word length.

5.3 Effect of Rounding Scheme

Figure 15 shows a tiny N-BEATS training example written in HLS C to demonstrate the difference in the rounding scheme. The model used M_{blk}=2 N-BEATS blocks, with the weight matrix size $L_k = 8$. The x-axis is the number of the epochs and the y-axis is the MAPE loss. The HLS C-stochastic curve is for stochastic rounding of weight update, while HLS C-nearest curve uses round to nearest. The figure shows that the HLS C-stochastic curve can

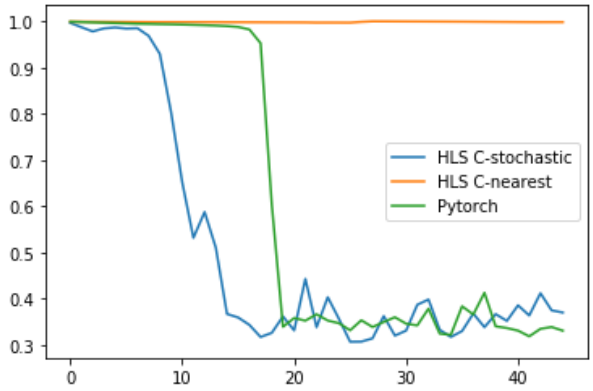


Fig. 15. Effect of different rounding schemes with tiny N-BEATS training example

Table 4. sMAPE loss for N-BEATS training

Block size	BM8-uniform	BM4-mixed	BM4-uniform-1
16×16	12.95	14.47	17.82
64×64	12.98	14.79	18.45
256×256	12.97	15.00	18.88
whole matrix	12.97	15.69	23.21
FP32		12.93	
BM4-uniform-2		32.01	

converge to the same loss value as PyTorch, while the HLS C-nearest curve does not converge. Here, the PyTorch curve also uses stochastic rounding for weight update computation.

5.4 Accuracy and Performance of N-BEATS Training

Three different precision configurations for software and hardware evaluation were tested, as shown in Table 3. Table 4 shows the N-BEATS training accuracy with different BM configurations, evaluated using sMAPE loss. Firstly, The impact of block size is less significant in BM8-uniform configuration; its training result is close to FP32. For 4-bit, as the block size is reduced, both BM4-mixed and BM4-uniform-1 configurations achieve smaller sMAPE loss. Secondly, we note that a high precision residual during training is also important, sMAPE loss decreases from 32.01 to 23.21 when a $BM\langle 0, 15\rangle$ format residual is applied. Thus, mixed precision significantly improves accuracy.

Table 5 shows an area comparison of the N-BEATS training accelerator for different precision configurations. The training accelerator uses a GEMM kernel with 32×32 PEs without DSP packing. The model training parameters are the same as Table 4. The same design with different number systems was used to obtain Table 5. For all designs, the target frequency was set to 200 MHz, and the frequency reported in Table 5 and Table 6 is from the Vivado post-routing report.

The BM4-uniform-1 configuration accelerator has 17.5% lower LUT consumption than BM4-mixed, and uses 31.1% less BRAM and 20.3% LUT than the BM8-uniform configuration. An advantage of smaller word length is that on-chip memory size requirements are reduced. Our hardware resource constraint is on-chip memory, and

Table 5. area comparison of N-BEATS training accelerator with different precision configuration

	LUT	BRAM	DSP	Freq. (MHz)
BM8-uniform	273K	942	1111	157
	36.23%	80.93%	18.72%	
BM4-mixed	218K	649	1111	183
	28.85%	55.76%	18.72%	
BM4-uniform-1	180K	649	1111	179
	23.80%	55.76%	18.72%	

Table 6. Performance comparison between various training accelerators

	[23]	[24]	[25]	[9]	Ours (no packing)	Ours (packing)	GPU
Device	Stratix 10 MX	VC709	MAX5	ZCU102	Alveo U50	Alveo U50	GTX 1080
number system	FP16	INT16	INT8	$BM\langle 2,5\rangle$	BM4-mixed	BM4-mixed	FP32
Freq(MHz)	185	-	200	225	183	159	1733
Network	ResNet 20	AlexNet	VGG-like	VGG-like	N-BEATS	N-BEATS	N-BEATS
DSP	1040(26%)	≈ 2880(80%)	6241(91%)	373(15%)	1111(19%)	1103(19%)	-
LUT/ALM	239K(34%)	-	679K	147K(54%)	218K(29%)	498K(66%)	-
BRAM/M20K	2558(37%)	-	1232(29%)	1255(69%)	649(56%)	1003(86%)	-
Batch size	1	-	128	1	1024	1080	1024
Tput. (Gops)	180	1022 (per FPGA)	1417	209	299.03	779.12	**2991**
Power(W)	20	32	13.5	7.7	20.86	18.38	220
Gops/W	9	31.97	**105**	27.1	14.34	42.4	13.6
Gops/DSP	0.17	0.36	0.34	0.56	0.27	**0.71**	-

DSP packing significantly reduces LUT utilization. In the present design, the performance is limited by HBM bandwidth, so unfortunately, additional computational capability will not improve throughput.

The mixed precision 4-bit accelerator performance with and without packing is given in Table 6. It uses a GEMM kernel with 36×24 PEs, with each PE employing a single DSP to implement six, 2×3 significand multipliers. Overall, the GEMM kernel has 72×72 MAC units.

For the packing configuration in Table 6, the batch size used was 1080 to fit the tile size, and a block size of 12×12 was chosen. This resulted in an sMAPE loss of 14.5. The accelerator achieves a throughput of 779.12 Gops and system power of 18.38 W from the Vitis analyzer. The implementation with packing has lower power consumption due to the lower clock frequency.

The GPU we choose for comparison is an Nvidia GTX 1080 as both it and the AMD Alveo U50 were fabricated using a TSMC 16 nm FinFET process. The GTX 1080 GPU implementation has a throughput of 2991 Gops with FP32 precision for training, and power consumption of 220 W measured using the nvidia-smi tool. The FPGA implementation demonstrated 3.12x better Gops/W than the GPU, and better Gops/DSP performance than other FPGA training implementations, albeit for different sizes and types of neural networks.

We note that a direct comparison of the different implementations in Table 6 is not particularly meaningful as the underlying neural network architectures are different, and computational demands are dependent on the network topology. The main purpose is to demonstrate that the proposed architecture achieves good utilization of the FPGA resources and can sustain high throughput for training. Regarding the batch size, a small value optimizes latency, whereas a large one is best for throughput. For example, [23] and [9] in Table 6 are optimized

for latency. This is because "Low-batch training greatly reduces memory requirement and unlocks opportunities for FPGAs" [23]. The N-BEATS paper uses batch size 1024 for training, and we used the same number in our implementation. Since matrix multiplication is the dominant computation in N-BEATS training, a larger batch size can increase the utilization of the GEMM kernel and thus improve the system's throughput.

5.5 Related Work

5.5.1 Number Systems. Conventional training accelerators typically use 16 or 32-bit FP encoding (such as IEEE-754 single-precision, half-precision [26], or Google Brain Floating-Point 16 bits (BFLOAT16) [27]). Different low-precision number systems have been applied to low-precision training, significantly improving the performance of hardware implementations. Seungkyu et al. [28] proposed a heterogeneous data type implementation for neural network training, using low-precision fixed-point activation/weight and half-precision error/gradient. A novel MAC architecture is designed to facilitate the heterogeneous data-type inputs configured with different precision levels of unbalanced bit-widths. The FP8 data type [1, 2] is another popular number system for low-precision training and uses 5-bit exponent and 2-bit mantissa, or 4-bit exponent and 3-bit mantissa data representations for training. Notably, FP8 was highlighted in a paper from NVIDIA, ARM, and Intel [5] in which they demonstrated the efficacy of the format on various training tasks, matching the accuracy of 16-bit approaches. However, FP8 was only applied to GEMM operations for fully connected and convolutional layers, with non-GEMM operations remaining in 16-bit floating-point (FP16) or BFLOAT16.

Examples include BM [7] and FP8 [5, 29], in which the latter utilizes a scaling factor for each layer of a Transformer neural network [30]. Previous work [8] has shown that BM can achieve similar accuracy to FP16 and with area close to 8-bit signed integer (INT8) for inference. Microscaling [31] adopted a similar numerical format, but included special values such as NaN and Inf not present in block minifloat. The techniques in this paper could be directly used to implement Microscaling.

5.5.2 DSP Packing. DSP packing has been frequently used in neural network inference implementation to improve computational density in FPGAs, which have a fixed number of high-performance blocks. Xilinx proposed 8-bit integer [18] and 4-bit integer [19] for convolutional neural network (CNN) inference. A recent paper [20] supports six 4-bit signed multiplications packed on one DSP for CNN inference. [21] enables four 8-bit FP multiplications on one DSP. The present work extends this idea by supporting different precision formats, as illustrated in Figure 10, and using it for training rather than inference.

5.5.3 FPGA Training Accelerators. Guo et al. [32] proposed a CNN training accelerator using sparsity and pruning that achieved 641GOP/s equivalent performance. Geng et al. [24] proposed the ALexNet training accelerator using the FPGA cluster. The power efficiency per FPGA is up to 3.4 times higher than the Tesla K80 GPU. Luo et al. [25] implemented 8-bit CNN training accelerator by utilizing batch-level parallelism. Venkataramanaiah et al. [23] implemented a CNN accelerator for low-batch training. Finally, Guo et al. [9] described a $BM\langle 2, 5\rangle$ data type accelerator for single-batch CNN transfer training. This work differs from [9] in the use of mixed precision arithmetic to reduce precision loss from cross-block operations and utilizes DSP packing. To the best of our knowledge, is the first reported FPGA-accelerated time-series forecasting network.

6 CONCLUSION

In this work, we demonstrated the feasibility of training time series forecasting networks using BM arithmetic. Our mixed precision scheme combined with a high-precision residual was used to implement NBEATS at primarily 4-bit precision (Mix BM4) and achieve a sMAPE loss of 14.47, which is similar to an FP32 result of 12.93 on the M4-Yearly dataset. We proposed a BM arithmetic implementation scheme that incorporates a novel BM GEMM architecture and 4-bit DSP packed PE design. Our Mix BM4 implementation uses 31.1% less BRAM and 20.3%

less LUTs compared to BM8, and with DSP packing, the accelerator achieves a throughput of 779 Gops, DSP utilization 0.7 Gops/DSP, and power efficiency 42.4 Gops/W.

ACKNOWLEDGMENT

The authors would like to thank the AMD/Xilinx University Program and the National University of Singapore for access to their Heterogeneous Accelerated Compute Cluster (HACC).

REFERENCES

[1] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[2] Léopold Cambier, Anahita Bhiwandiwalla, Ting Gong, Mehran Nekuii, Oguz H Elibol, and Hanlin Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. *arXiv preprint arXiv:2001.05674*, 2020.

[3] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *Advances in Neural Information Processing Systems*, 33:1796–1807, 2020.

[4] Jiawei Zhao, Steve Dai, Rangharajan Venkatesan, Brian Zimmer, Mustafa Ali, Ming-Yu Liu, Brucek Khailany, William J Dally, and Anima Anandkumar. Lns-madam: Low-precision training in logarithmic number system using multiplicative weight update. *IEEE Transactions on Computers*, 71(12):3179–3190, 2022.

[5] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[6] Sai Qian Zhang, Bradley McDanel, and HT Kung. Fast: Dnn training under variable precision block floating point with stochastic rounding. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 846–860. IEEE, 2022.

[7] Sean Fox, Seyedramin Rasoulinezhad, Julian Faraone, Philip Leong, et al. A block minifloat representation for training deep neural networks. In *International Conference on Learning Representations*, 2020.

[8] Wenjie Zhou, Haoyan Qi, David Boland, Philip Leong, et al. Fpga implementation of n-beats for time series forecasting using block minifloat arithmetic. In *the 18th Asia Pacific Conference on Circuits and Systems*, 2022.

[9] Chuliang Guo, Binglei Lou, Xueyuan Liu, David Boland, Philip HW Leong, and Cheng Zhuo. Boost: Block minifloat-based on-device cnn training accelerator with transfer learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023.

[10] David Elam and Cesar Lovescu. A block floating point implementation for an n-point fft on the tms320c55x dsp. *Texas Instruments Application Report*, 2003.

[11] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. *Advances in Neural Information Processing Systems*, 31, 2018.

[12] Simla Burcu Harma, Ayan Chakraborty, Babak Falsafi, Martin Jaggi, and Yunho Oh. Accuracy boosters: Epoch-driven mixed-mantissa block floating-point for dnn training. *arXiv preprint arXiv:2211.10737*, 2022.

[13] Boris N Oreshkin, Dmitri Carpov, Nicolas Chapados, and Yoshua Bengio. N-beats: Neural basis expansion analysis for interpretable time series forecasting. In *International Conference on Learning Representations*, 2019.

[14] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *Siam review*, 61(4):860–891, 2019.

[15] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.

[16] Bita Darvish Rouhani, Ritchie Zhao, Venmugil Elango, Rasoul Shafipour, Mathew Hall, Maral Mesmakhosroshahi, Ankit More, Levi Melnick, Maximilian Golub, Girish Varatkar, et al. With shared microexponents, a little shifting goes a long way. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[17] AMD Xilinx. Systolic array. https://xilinx.github.io/Vitis_Accel_Examples/2019.2/html/systolic_array.html.

[18] Xilinx. Deep learning with int8 optimization on xilinx devices white paper (wp486). https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8.

[19] Xilinx. Convolutional neural network with int4 optimization on xilinx devices (wp521). https://docs.xilinx.com/v/u/en-US/wp521-4bit-optimization.

[20] Liu Qi, Sun Mo, Sun Jie, Lu Liqiang, Zhao Jieru, and Wang Zeke. Ssimd: Supporting six signed multiplications in a dsp block for low-precision cnn on fpgas. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 161–169. IEEE, 2023.

[21] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. Low-precision floating-point arithmetic for high-performance fpga-based cnn acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(1):1–21, 2021.
[22] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The M4 Competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*, 34(4):802–808, 2018.
[23] Shreyas K Venkataramanaiah, Han-Sok Suh, Shihui Yin, Eriko Nurvitadhi, Aravind Dasu, Yu Cao, and Jae-sun Seo. Fpga-based low-batch training accelerator for modern cnns featuring high bandwidth memory. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–8, 2020.
[24] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. Fpdeep: Acceleration and load balancing of cnn training on fpga clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 81–84. IEEE, 2018.
[25] Cheng Luo, Man-Kit Sit, Hongxiang Fan, Shuanglong Liu, Wayne Luk, and Ce Guo. Towards efficient deep neural network training by fpga-based batch-level parallelism. *Journal of Semiconductors*, 41(2):022403, 2020.
[26] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
[27] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.
[28] Seungkyu Choi, Jaekang Shin, and Lee-Sup Kim. A deep neural network training architecture with inference-aware heterogeneous data-type. *IEEE Transactions on Computers*, 71(5):1216–1229, 2021.
[29] Greg Palmer Michael Andersch. Inside the nvidia hooper architecture. https://rd.yyrcd.com/CUDA/2022-GTC/S42663-Inside_the_NVIDIA_Hopper_Architecture.pdf.
[30] Using fp8 with transformer engine. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html.
[31] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.
[32] Kaiyuan Guo, Shuang Liang, Jincheng Yu, Xuefei Ning, Wenshuo Li, Yu Wang, and Huazhong Yang. Compressed cnn training with fpga-based accelerator. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 189–189, 2019.