FPGA Implementation of N-BEATS for Time Series Forecasting using Block Minifloat Arithmetic

Wenjie Zhou, Haoyan Qi, David Boland and Philip H.W. Leong

Computer Engineering Laboratory
School of Electrical and Information Engineering
The University of Sydney, Australia 2006
Email: {wenjie.zhou, haoyan.qi, david.boland, philip.leong}@sydney.edu.au

***Abstract*—The block minifloat (BM) number format uses an 8-bit floating point format with additional shared exponent bias to enable low-precision representation with large dynamic range. While it has been shown that the BM format can support low-precision training of convolutional neural networks such as ResNet on ImageNet at precisions down to 6 bits, its applicability to inference-only applications has not been studied. We present a BM implementation of N-BEATS, a deep neural architecture for univariate time series forecasting. N-BEATS utilises residual and fully connected (FC) blocks to achieve high accuracy. It was found that 8-bit BM had similar area and speed as 8-bit integer arithmetic with NBEATS accuracy similar to 16-bit floating point.**

***Index Terms*—Block minifloat, Minifloat, Neural network inference**

I. Introduction

Machine learning inference using low-precision arithmetic has been a heavily researched topic as applications of inference abound, and reducing precision requirements without greatly sacrificing accuracy can enable new applications where conventional inference implementations are too computationally expensive. Field programmable gate arrays (FPGAs) are an excellent platform for this type of application, particularly for edge devices where size, weight and power (SWaP) are design considerations.

Although many deep neural network (DNN) training and inference accelerators use 16 or 32 bit floating point encoding (such as IEEE-754 single-precision, half-precision [1], or Google Brain Floating-Point 16 bits (BFLOAT16) [2]), lower precision weights and activations offer significant implementation benefits including reduced memory bandwidth, memory requirements, and implementation area. One popular scheme for DNN accelerators design is fixed-point arithmetic which requires significantly fewer hardware resources than floating-point [3], [4]. However, one problem for fixed-point implementations is a small dynamic range, leading to reduced accuracy. Quantization aware training techniques have been proposed to address this problem [5]–[7], and commercial quantization-aware training products are available from vendors such as Advanced Micro Devices (AMD) and Intel (e.g. [8]).

There has been recent interest in low-precision floating-point data formats for the training of DNNs. These utilise 8, or even 6 bit, floating-point representations are able to train neural networks from scratch [9], [10]. Minifloat uses a standard floating point representation of exponent and mantissa with small word lengths, whereas BM has an additional shared block exponent to expand its dynamic range. Little research has been conducted on inference using these schemes but compared to a conventional floating-point fused multiply accumulate (FMA), BM is computationally less expensive, particularly for floating-point formats with small exponent lengths as large shifters for alignment are avoided. For example, 32-bit integer adders are approximately $10\times$ smaller and $4\times$ more energy efficient than 16-bit floating-point (FP16) units [11].

In this paper, we explore the applicability of BM to time series prediction using the NBEATS algorithm [12] and present a BM-based inference accelerator design. NBEATS is a DNN for time series prediction which, at the time of publication, achieved 3% improvement over the winner of the M4 forecasting competition. The main contributions of this work can be summarised as:

- The first implementation of an FPGA-based accelerator using BM arithmetic.
- We present a novel architecture for acceleration of NBEATS based on BM arithmetic and a systolic datapath.
- The accuracy and area trade-offs between BM, floating-point and fixed-point are explored. It is found that 8-bit BM achieves area and performance similar an 8-bit signed integer (INT8) datapath with accuracy on NBEATS similar to FP16.

The remainder of this paper is organised as follows. In Section II we review the BM number format and NBEATS. In Section III we present the architecture of our BM accelerator. Results are presented in Section IV and finally, conclusions drawn in Section V.

II. Background

A. BM data format

The real value of a $f\langle e, m\rangle$ format minifloat is given by Equation 1. This includes support for normal and denormalised (denorm) numbers, but saturating arithmetic is employed instead of IEEE-754 overflow and underflow, Inf and NaN.

$$f(s,E,M)=\begin{cases} S\times 2^{1-\beta^{<e>}} & E=0 \text{ (denorm)} \\ S\times 2^{E-\beta^{<e>}} & E\neq 0 \text{ (normal)} \end{cases} \tag{1}$$

where e is the number of exponent bits, m the number of mantissa bits, E and M are the unsigned integer exponent and mantissa values, s is the sign bit, and $\beta^{<e>} = 2^e - 1$ is the exponent bias for the binary-offset encoding scheme. Its significand is:

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

We also define an inverse which extracts the component sign, exponent and mantissa values from a minifloat value x:

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

Similar to block floating-point (BFP) [13], the BM format [10], $BM\langle e,m\rangle$, is used to describe a tensor X, where each element $r_i\in X$ has a shared exponent bias β_{share}:

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

As an example, the $BM\langle 2,5\rangle$ format (sometimes written as $BM8\langle 2,5\rangle$ to indicate the word length), with a shared bias of 0, has an exponent range of [0,3] and mantissa range of [0,31].

B. NBEATS network

Figure 1 illustrates the NBEATS architecture. It is composed from a number of NBEATS blocks, each having 4 FC layers with rectified linear unit (ReLu) activation, split into backcast and forecast branches. Each NBEATS block is organised in a doubly residual manner as illustrated, where for Block 2-N the residual activation is the difference between the previous block's input and ackcast output. All forecast outputs are summed to form the prediction output.

III. BM INFERENCE ACCELERATOR ARCHITECTURE

A. BM general matrix multiplication (GEMM) computation

The computational bottleneck of NBEATS lies in the computation of the FC layers, which translates into tensor multiplication, implemented as a sequence of BM matrix multiplications. The strategy for its acceleration is to use an efficient GEMM core which computes the inner product between each row and column of the input matrices, A and B (with shared exponents β_A and β_B) to produce the output matrix C. For the BM data format, each element of C is the inner product $c_{ij}=\sum_k f(s_{a_{ik}},M_{a_{ik}},E_{a_{ik}})f(s_{b_{kj}},M_{b_{kj}},E_{b_{kj}})2^{\beta_A+\beta_B}$

The computation of minifloat GEMM is done in three steps: computation of the inner-product, normalization of partial sum to BM format, and alignment of the shared exponent bias. The shared exponent bias of the new matrix is $\beta_{share}=\beta_A+\beta_B$.

For the inner product computation, operands a and b (in $BM\langle e_a,m_a\rangle$ and $BM\langle e_b,m_b\rangle$ format) are multiplied as:

$$\begin{aligned} S_{mul} &= S_a * S_b; \\ E_{mul} &= E_a + E_b \end{aligned} \tag{5}$$

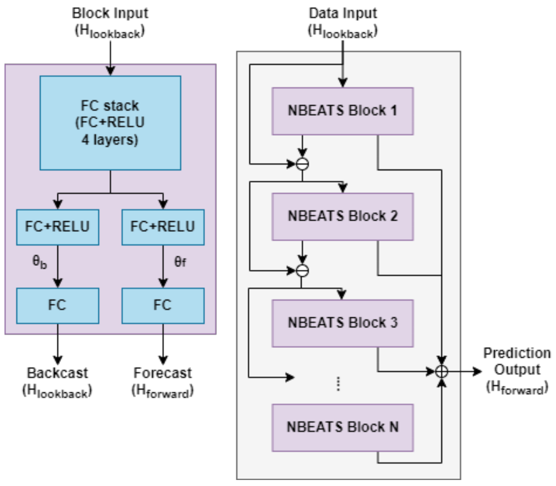

Fig. 1. NBEATS neural network model. The left-hand side shows the NBEATS block. The input of each block goes first to the 4 layer FC stack, and then the FC stack output activation goes to the backcast branch and forecast branch. The backcast output size is same as the block input size, and the forecast output size is same as the model output size. The right-hand side shows the NBEATS model.A series of NBEATS blocks are connected to each other. The input of each NBEATS block is the differential value of the output and the input from the previous NBEATS block, and the model output is the sum of all forecast values of the NBEATS block.

S_{mul} and E_{mul} are the significand and exponent values of the minifloat multiplication result. These numbers are combined and accumulated with the partial sum to form the signed integer inner product P_{sum}:

$$P_{sum}=P_{sum}+S_{mul}<<E_{mul}. \tag{6}$$

The inner product computation just described is implemented using an integer Kulisch accumulator with sufficient precision for error free accumulation [10], [14]. Its size K_{add} and shifter range K_{shift} is given in Equation 7. An extra WI bits are assigned to avoid overflow over multiple accumulations. For the BM data format, the minifloat exponent value is small, which limits the required size of the adder.

$$\begin{aligned} K_{shift} &= 2^{e_a}+2^{e_b} \\ K_{add} &= 1+2^{e_a}+2^{e_b}+(1+m_a+1+m_b)+WI \end{aligned} \tag{7}$$

The GEMM produces a block of N values, $P[N]$, which are converted to minifloats using the BM normalization function (Algorithm 1). This involves transforming the accumulator outputs to a normalized $C\langle e_{norm},m\rangle$ number with *KulToBM* and then converting into $A\langle e,m\rangle$ format with *ExpAlign*:

$$A,\beta'_{share}=ExpAlign(KulToBM(P),\beta_{share}) \tag{8}$$

where $e_{norm}\geq\lceil\log_2 K_{add}\rceil$ In *KulToBM*, leading zeros are determined using the count leading zero function *CLZ*, and denormal, normal, underflow cases detected by comparisons with the boundary values defined below:

$$\begin{aligned} Cnt_{denorm} &= WI+(2^{e+1}-\beta^{<e+1>}+\beta^{<e>}-1) \\ Cnt_{underflow} &= Cnt_{denorm}+m \end{aligned} \tag{9}$$

Note that the result is in an intermediate $C\langle e_{norm},m\rangle$ format. Exponent alignment back to the original BM format is done using *ExpAlign* in Algorithm 1 where the shared exponent bias is adjusted and normalized to $A\langle e, m\rangle$ format.

Algorithm 1: Kulisch adder output integer to $BM\langle e, m\rangle$ conversion

```
Function KulToBM($P[N]$)
    // Normalization (integer to $C\langle e_{norm}, m\rangle$)
    for $i \leftarrow 0$ to $N-1$ do
        $s \leftarrow (P[i] > 0)\ ?\ 0 : 1$; // Sign bit
        $Z \leftarrow CLZ(|P[i]|)$; // Leading zeros
        if $Z \geq Cnt_{denorm}$ and $Z < Cnt_{underflow}$ then
            $C[i] \leftarrow f(s, 0, |P[i]| << (Cnt_{denorm} - 1))$;
              // Denorm
        else if $Z \geq Cnt_{underflow}$ then
            $C[i] \leftarrow f(s, 0, 0)$; // Underflow
        else
            $E \leftarrow (2^{e+1} - 1 - \beta^{<e+1>}) + WI - Z + \beta^{<e>}$;
            $C[i] \leftarrow f(s, E, |P[i]| << Z)$; // Normal
    $E_{max} \leftarrow maxexp(C)$; // Max exponent in block
    return $C$, $E_{max}$
Function ExpAlign($C[N]$, $E_{max}$, $\beta_{share}$)
    // Exponent Alignment ($C\langle e_{norm}, m\rangle \rightarrow A\langle e, m\rangle$)
    $\beta'_{share} \leftarrow \beta_{share} + max(E_{max} - (2^e - 1), 0)$;
    for $i \leftarrow 0$ to $N-1$ do
        $s, E, M \leftarrow f^{-1}(C[i])$;
        $E \leftarrow E - max(E_{max} - (2^e - 1), 0)$;
        if $E \leq 0$ and $E > -m$ then
            $A[i] \leftarrow f(s, 0, M >> (1 - E))$; // Denorm
        else if $E \leq -m$ then
            $A[i] \leftarrow f(s, 0, 1)$; // Underflow
        else
            $A[i] \leftarrow f(s, E, M)$; // Normal
    return $A$, $\beta'_{share}$;
```

The GEMM implementation is based on de Fine Licht et al. [15], modified to support BM. As illustrated in Figure 2, it is a 1D weight-stationary systolic array with compile-time configurable size. The configuration used is 32 processing elements (PEs) each with 16 parallel multiply accumulate (MAC) units, and 512 memory tile size for both input matrices. It supports matrix multiplication of $N_{\text{gemm}} \times N_{\text{gemm}}$ blocks where $N_{\text{gemm}} = 512$. The input data are streamed through the PEs, with the width of this bus equal to the MAC per PE number (i.e. $16W$, where W is 8 or 16-bits for BM8 and FP16). Data transfers are double buffered and will flow to the following GEMM block without interrupting computation.

As shown in Figure 2, the GEMM block receives BM data as input (e.g. weights and input feature map (IFM) values), and each PEs performs multiplication and Kulisch accumulation. Intermediate results are stored in the Kul buffer. After the GEMM computation finishes, the result will be transformed to a BM format through *KulToBM* and the max exponent will be computed through *maxexp*.

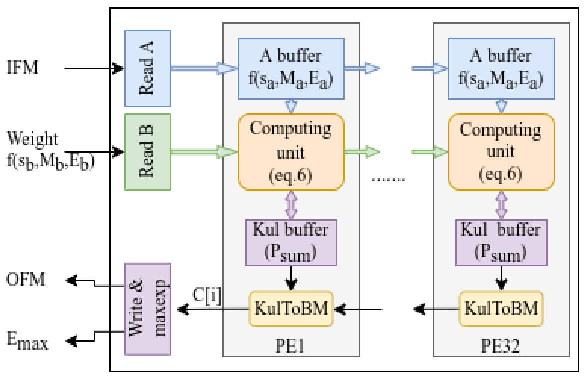

Fig. 2. Systolic GEMM block used for the NBEATS Inference Accelerator

B. BM Vector Addition

The BM addition is also conducted in fixed-point format and computed as follows:

$$a \pm b = \begin{cases} (S_a 2^{E_a} \pm S_b 2^{E_b} * 2^{\beta_B - \beta_A}) * 2^{\beta_A}, \beta_A \geq \beta_B \\ (S_a 2^{E_a} * 2^{\beta_A - \beta_B} \pm S_b 2^{E_b}) * 2^{\beta_B}, \beta_A < \beta_B \end{cases} \tag{10}$$

where S_a, S_b are the significands of a and b (Equation 2). The shared exponent result used is the larger of β_A and β_B:

$$\beta_{share} = \max(\beta_A, \beta_B) \tag{11}$$

Algorithm 2: BM Vector Addition

```
Function BMVecAdd($A[N]$, $B[N]$, $\beta_A$, $\beta_B$)
    $ebias \leftarrow |\beta_A - \beta_B|$;
    for $i \leftarrow 0$ to $N-1$ do
        $s_a, E_a, M_a \leftarrow f^{-1}(a[i])$;
        $s_b, E_b, M_b \leftarrow f^{-1}(b[i])$;
        $S_a \leftarrow g(s_a, M_a)$; // Equation 2
        $S_b \leftarrow g(s_b, M_b)$;
        $A_{isnorm} \leftarrow (E_a == 0)\ ?\ 0 : 1$;
        $B_{isnorm} \leftarrow (E_b == 0)\ ?\ 0 : 1$;
        if $\beta_A >= \beta_B$ then
            $K_a \leftarrow E_a - A_{isnorm}$;
            $K_b \leftarrow E_b - ebias - B_{isnorm}$;
        else
            $K_a \leftarrow E_a - ebias - A_{isnorm}$;
            $K_b \leftarrow E_b - B_{isnorm}$;
        $t \leftarrow S_a[i] << K_a + S_b[i] << K_b$;
        $R[i] \leftarrow KulToBM'(t)$;
    $\beta'_{share} \leftarrow \max(\beta_A, \beta_B)$;
    return $R$, $\beta'_{share}$;
```

The BM vector addition pseudocode is given in Algorithm 2, with inputs and outputs being in BM format. The significands are first extracted. Next K_a and K_b are computed. These are the shifts required for alignment considering their exponents, denormalisation and shared exponents. The sum

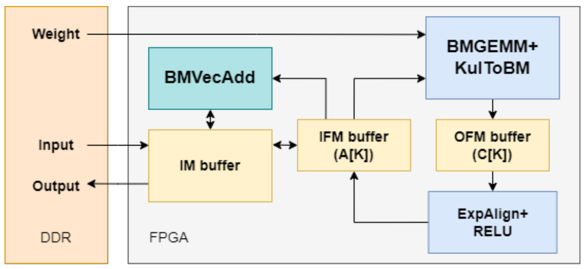

Fig. 3. NBEATS Inference Accelerator.

is then computed in t. Finally, t is converted into BM format using *KulToBM'* (same as *KulToBM* with output format $BM\langle e, m\rangle$ instead of $BM\langle e_{norm}, m\rangle$) and returned along with the new shared exponent.

C. NBEATS Inference Accelerator

The system-level architecture of our NBEATS inference accelerator is illustrated in Figure 3. The weight, input dataset, and output tensors are initially stored in double data rate (DDR) memory. The IFM, output feature map (OFM) of each FC layer are buffers on the FPGA. The intermediate buffer (IM) buffer stores some intermediate results, including the backcast output, forecast sum, and the FC stack output. *BMGEMM+KulToBM* is used to compute the FC layers and combines the GEMM block with integer outputs and *KulToBM* so that inputs and outputs are in BM format. The *BMVecAdd* block is used for computing residual inputs to each NBEATS block and the final prediction output.

During inference computation, a batch of input data are fetched from DDR and written to the IM buffer. The same data are transferred to the IFM buffer to serve as input activations. Off-chip weights are passed to the *BMGEMM+KulToBM* block to compute each FC layer and the result placed in OFM. The OFM result is then aligned, ReLu applied and written to IFM. The backcast residual computation and prediction output forecast sum are computed from the intermediate data in the IFM and written to IM.

The activation and weights of each layer are organised in separate $BM\langle e, m\rangle$ blocks. Each weight layer has a shared exponent bias, as does each activation layer.

IV. Results

A. Experimental Configuration

We evaluated the performance of our implementation on Xilinx Alveo U250 accelerator card with the Virtex UltraScale+ XCU250-L2FIGD2104E FPGA and 4 DDR memories. The inference accelerator is written in Vitis HLS 2020.2 and implemented using Vivado 2021.2 with target frequency of 300 MHz. The BM data representation and computations, including Kulisch accumulation, conversion and vector addition, are implemented using the HLS *ap_int* or *ap_uint* data types.

We use the model architecture and benchmarks from [12] with parameters detailed in Table I. The model used M_{blk}=30 NBEATS blocks, with the weight matrix $N_{\text{gemm}} \times N_{\text{gemm}}$ being 512×512 for all fully connected layers. Input data is arranged as a 2D array of dimension $B \times H_{\text{lookback}}$ where

TABLE I
NBEATS Experiment Model Configurations

	Yearly	Quarterly	Monthly	Daily
$H_{forward}$	6	8	18	14
$H_{lookback}$	12	16	36	28
$dim(\theta^f)$	18	24	54	42
$dim(\theta^b)$	18	24	54	42
Samples	22850	64144	136603	187930

TABLE II
NBEATS Accuracy (sMAPE loss) Comparison Across BM8, INT8, FP16

	Yearly	Quarterly	Monthly	Daily
FP32	13.462	11.992	11.278	3.869
FP16	13.454	11.986	11.273	3.868
INT8	19.661	18.713	15.279	9.489
BM8+round nearest	14.780	13.548	11.972	4.675

the batch size is $B = 512$ and a forecast horizon H_{forward} is 6, 8, 18, 14 for each seasonal pattern. The backcast length is $H_{\text{lookback}} = 2H_{\text{forward}}$. Experiments include training and validation on the M4 benchmark of seasonal patterns, i.e. M4-Yearly, M4-Quarterly, M4-Monthly and M4-Daily. The other patterns, i.e. M4-Weekly, M4-Hourly, are discarded due to the limited number of samples available in the original dataset.

For our hardware implementation, we maintain the same input data size configuration as Table I. To match the GEMM size, inputs, θ^b, θ^f are zero padded to N_{gemm}. Both weight and input data are stored in off-chip DDR memory encoded with $BM8\langle 2, 5\rangle$ data type, and the IFM data type is $BM8\langle 2, 5\rangle$, the OFM data type is $BM12\langle 6, 5\rangle$. The Kulisch accumulator length $K_{add} = 25$, which is derived from Equation 7 with $WI = 4, m_a = m_b = 5, e_a = e_b = 2$. All the shared exponents are placed in on-chip buffers and encoded using INT8.

B. Accuracy and Performance

Table II shows the symmetric mean absolute percentage error (sMAPE) for various data types using the Yearly, Quarterly, Daily datasets from the M4 benchmark. This experiment was conducted using post training static quantization [16] with BM8 being a significant improvement on INT8 and close FP16 accuracy.

Table III compares the hardware resource and power consumption of individual multipliers and adders for the different data types. All of the designs are synthesized from C. It can be seen that the BM8 area and power consumption is close to INT8, and much smaller than FP16.

Table IV shows the NBEATS accelerator resource utilisation, with the percentage being that of the total FPGA

TABLE III
MAC Unit Resource Utilisation and Power consumption across BM8, FP16, INT8

	LUT	DSP	FF	Power
FP16 multiply	44	2	34	1.582W
FP16 add	108	2	34	2.5W
BM8 multiply	47	0	0	0.524W
BM8 add	24	1	0	1.047W
INT8 multiply	35	0	0	0.328W
INT8 add	8	0	0	0.117W

TABLE IV
NBEATS INFERENCE ACCELERATOR RESOURCE CONSUMPTION USING BM8, INT8, FP16

	LUT	REG	BRAM	URAM	DSP
BM8	38131	50762	373	23	192
	2.45%	1.59%	15.67%	1.8%	1.56%
INT8	27112	38418	439	16	512
	1.74%	1.2%	18.45%	1.25%	4.17%
FP16	119246	157588	1019	0	2048
	7.66%	4.94%	42.82%	0	16.68%

TABLE V
NBEATS INFERENCE ACCELERATOR PERFORMANCE AND POWER CONSUMPTION USING BM8, INT8, FP16

	Frequency	Latency	Peak Performance	Power
BM8	300MHz	0.232s	277 GOPS	21.44W
INT8	300MHz	0.228s	282 GOPS	21.97W
FP16	200MHz	0.354s	182 GOPS	22.674W

resources. Table V shows peak performance which is achieved when inputs, θ^b, θ^f are divisible by N_{gemm}. The BM8 and INT8 inference accelerator can operate at 300 MHz frequency, but the FP16 accelerator can only satisfy a 200 MHz timing constraint. As a result, the BM8 design can achieve 277 GOPS performance which is similar to INT8 and significantly better than the FP16 design. The total operations can be estimated by Equation 12 with $2M_{blk}$ accounting for the number of operations per MAC and the number of NBEATS blocks. The first term in braces accounts for the 3 FC layers; the next term for the FC layer input, backcast and forecast branches; and the final term corresponds to the last 2 FC layers for the backcast and forecast. The computation for forecast sum and backcast residual is small and hence omitted in Equation 12. For the parameters in Table I, performance (Op/Op_{peak}) varies from 38.7% to 41.2% of the peak, where Op_{peak} is achieved when $H_{\text{forward}} = H_{\text{loopback}} = N_{\text{gemm}}$.

$$\begin{aligned} Op \approx 2M_{\text{blk}}\{&3N_{\text{gemm}}^3 + N_{\text{gemm}}^2(2H_{\text{forward}} + 3H_{\text{lookback}}) \\ &+ BN_{\text{gemm}}(H_{\text{forward}} + H_{\text{lookback}})^2\} \end{aligned} \tag{12}$$

The power in Table V is the total on-chip power. All three implementations are configured with two DDR bank with one bank for input/output and the other for weight. The power consumption of BM8 and INT8 is slightly better than FP16. Using a single DDR bank results in the same performance but decreases power consumption by 11%.

V. Conclusion

We presented an HLS C library for the implementation of matrix multiplication using BM arithmetic. This was used to implement an FPGA-based NBEATS accelerator which can achieves a peak performance of 277 GOPS on a Xilinx Alveo U250 board. The performance in terms of power, area and throughput is similar to INT8 (282 GOPS) with accuracy similar to FP16 (182 GOPS).

NBEATS acceleration is ideally suited to acceleration via a GEMM accelerator because its computational bottleneck are fully connected layers. In future work we will explore inference and training using BM arithmetic with other deep neural networks.

Acknowledgment

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program (formerly known as the XACC program - Xilinx Adaptive Compute Cluster program).

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] 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.

[3] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 26–35, 2016.

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

[5] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded FPGA. *IEEE transactions on computer-aided design of integrated circuits and systems*, 37(1):35–47, 2017.

[6] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International conference on machine learning*, pages 2285–2294. PMLR, 2015.

[7] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[8] Convolutional neural network with INT4 optimization on Xilinx devices. *Xilinx White Paper*, 2020.

[9] 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.

[10] 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.

[11] William J Dally. High performance hardware for machine learning, 2015. https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.

[12] 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.

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

[14] Ulrich W Kulisch and Willard L Miranker. *Computer arithmetic in theory and practice*. Academic press, 2014.

[15] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, 2020.

[16] Introduction to quantization on pytorch. https://pytorch.org/blog/introduction-to-quantization-on-pytorch/.