DBPS: Dynamic Block Size and Precision Scaling for Efficient DNN Training Supported by RISC-V ISA Extensions

Seunghyun Lee*
School of Electrical Engineering
Korea University, South Korea

Jeik Choi*, Seockhwan Noh, Jahyun Koo
Department of EECS
DGIST, South Korea

Jaeha Kung†
School of Electrical Engineering
Korea University, South Korea

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10248013

***Abstract*—Over the past decade, it has been found that deep neural networks (DNNs) perform better on visual perception and language understanding tasks as their size increases. However, this comes at the cost of high energy consumption and large memory requirement to train such large models. As the training DNNs necessitates a wide dynamic range in representing tensors, floating point formats are normally used. In this work, we utilize a block floating point (BFP) format that significantly reduces the size of tensors and the power consumption of arithmetic units. Unfortunately, prior work on BFP-based DNN training empirically selects the block size and the precision that maintain the training accuracy. To make the BFP-based training more feasible, we propose dynamic block size and precision scaling (DBPS) for highly efficient DNN training. We also present a hardware accelerator, called DBPS core, which supports the DBPS control by configuring arithmetic units with custom instructions extended in a RISC-V processor. As a result, the training time and energy consumption reduce by 67.1% and 72.0%, respectively, without hurting the training accuracy.**

***Index Terms*—Block floating point, dynamic control, hardware accelerator, neural network training, RISC-V custom instruction**

I. Introduction

As the performance of deep neural networks (DNNs) continues to improve, they have been used in many real-world tasks, e.g., computer vision, natural language processing, robotics, and medical diagnosis. Recently, larger models, e.g., GPT-3 with 175B parameters [1], are designed and trained that are better at understanding complex and general contexts rather than learning a specific task. Unfortunately, training these large-scale DNNs significantly increases energy consumption and carbon emissions, making them a key challenge for data center maintenance [2]. Moreover, training in mobile devices is getting attention due to privacy issues of moving personal data to servers. Due to the limited memory capacity and energy budget, training accelerators at the edge need to be designed with better hardware efficiency.

Training DNNs consists of three steps: (i) forward pass that computes the loss ($\mathcal{L}$) of a given input data (X), (ii) backward pass that computes the local gradient ($\partial\mathcal{L}/\partial X$), and (iii) weight update that computes the weight gradient ($\partial\mathcal{L}/\partial W$). During these steps, floating point formats (i.e., `FP32` and `bfloat16`) are used to guarantee high training accuracy. However, floating point (`FP`) formats have two disadvantages in training DNNs. One is that they require a large memory footprint no matter how much precision and dynamic range are actually needed for representing intermediate data involved in the training process. Another disadvantage is the power consumption and area of `FP` arithmetic units, i.e., multipliers and adders, compared to integer (`INT`) counterparts. For instance, a `bfloat16` multiply-accumulate (MAC) unit consumes 1.29mW, while an `INT8` MAC consumes 0.23mW, i.e., 5.6× lower, in 45nm CMOS technology. Therefore, the use of `INT`-based MACs will significantly reduce the energy consumption of DNN training. Aligned with this, several recent studies [3]–[5] have proposed to utilize a block floating point (`BFP`) format that allows a group of numbers, called a *block* in this paper, to share an exponent. By sharing the exponent within the block, `INT`-based MAC operations for mantissas are performed and two shared exponents of two sub-tensor blocks are handled separately.

In this work, we present a co-processor attached to a RISC-V core that efficiently accelerates the `BFP`-based generalized matrix multiplication (GEMM) by dynamically configuring `BFP` precisions and block sizes (Fig. 1(a)). We extend the RISC-V ISA with custom instructions to realize the proposed dynamic block size and precision scaling (DBPS) that sets the precision (or block size) at its minimum (or maximum) for successful yet energy efficient DNN training (Fig. 1(b)).

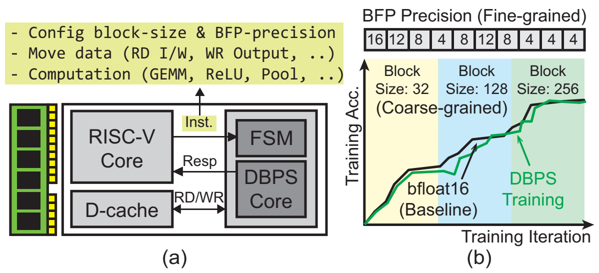


Fig. 1. (a) The overview of the proposed DBPS-enabled training accelerator and (b) an illustration of the training curve expected by applying the DBPS.

This work was supported by Samsung Research Funding Incubation Center of Samsung Electronics under Project Number SRFC-IT1902-03. The EDA tool was supported by the IC Design Education Center (IDEC) in South Korea.
* S. Lee and J. Choi are equally contributed authors.
† J. Kung is the corresponding author (*jhkung@korea.ac.kr*).

The main contributions can be summarized as follows:

1) **New `BFP` Data Format:** We keep the indices of values that have the maximum exponent within each block so that an implicit bit can be inferred without storing them. This effectively increases the precision by one bit for each operand.
2) **DBPS Control:** We propose a dynamic block size and precision control that minimizes the energy consumption of DNN training. The DBPS controller dynamically configures the appropriate block size and BFP precision by monitoring the number of underflows of each tensor.
3) **DBPS Core with Instruction Support:** We implement the DBPS core that actually performs BFP-based GEMM at various block sizes and precisions. We demonstrate the training efficiency by integrating the core with a RISC-V processor with custom instructions on a Xilinx Virtex-7 VC707 board.

II. Motivation

The proposed DBPS core in this paper accelerates the DNN training by leveraging block floating point arithmetic units. The challenge in using the BFP format is in determining the block size and precision of each tensor. The prior work on the BFP-based DNN training empirically determines the block size and its precision which limits its use in real applications [3], [5]. For instance, a small block size, i.e., eight in FAST [5], minimizes the mantissa bit-width while constraining the energy efficiency by having many FP32 accumulations. In addition, it cannot guarantee the training stability on a variety of DNN benchmarks. On the contrary, Flexpoint [4] blocks the entire tensor having one exponent per tensor while keeping the mantissa bit-width high, i.e., 16-bit. As shown in Fig. 2, the training accuracy highly depends on the precision and block size of the selected BFP format. In Fig. 2(a), various mantissa bit-widths are tested by fixing the block size of BFP format to 32 for all tensors. When the mantissa bit-width becomes 7-bit, the test accuracy significantly drops by 18% compared to the 9-bit case. In Fig. 2(b), various block sizes are evaluated by fixing the mantissa bit-width to 7-bit. When the block size becomes 16, the accuracy drops by 17% compared to the block size of 8. The minimum required precision or block size for training DNNs may vary by the model architecture and dataset. Therefore, we propose to automatically configure the required precision and block size for the next training iteration so that our BFP-based training accelerator can achieve its maximum energy efficiency while keeping the accuracy.

III. Dynamic Block Size and Precision Scaling (DBPS)

A. *Proposed Block Floating Point Format*

In the conventional FP format, such as IEEE 754, the leading bit of mantissa is always '1' for numbers in the normal range. To allow higher precision, this leading bit is implied and not stored in the memory. The implicit bits are colored in red in the `bfloat16` example in Fig. 3. However, in the BFP format, this is not feasible as only the numbers with the largest exponent have '1' as their leading bits, while others have '0'. As shown in the 'Naïve BFP4[1]' in Fig. 3, the 3-bit mantissa can only store 2 explicit bits from the `bfloat16` numbers. This limits the use of low-precision BFP format due to a larger quantization error. Thus, we propose to identify locations of numbers having the maximum exponent in each block so that leading bits can be easily inferred. As shown in 'Our BFP4' in Fig. 3, leading bits of the numbers with the maximum exponent (i.e., 128) are considered '1' even without storing them. This provides the same effect as having one implicit bit as in the conventional FP format with the overhead of storing the index data. Then, the required bits to store 'N' FP numbers using our BFP format becomes

$$\text{size}(\texttt{BFP}) = \text{len}(s{+}m)\cdot N + \text{len}(e_s) + \lceil \log_2(N) \rceil \cdot N_{max}, \quad (1)$$

where len(s+m) is the bit-width of 'sign+mantissa', len(e_s) is the bit-width of the shared exponent, and N_{max} is the number of values that have the maximum exponent. With '$\lceil \log_2(N) \rceil \cdot N_{max}$' additional bits for each block, we can extend the mantissa bit by one. Since tensors in DNN training have normal distributions, N_{max} from a single block is expected to be small. The average index bit overhead for various BFP precisions and block sizes are reported in Table I. For instance, a block of size 64 requires only 18.9 bits per block on average, which effectively extends the 64 implicit bits. Utilizing implicit bits helps improve the training accuracy especially when low BFP precision is used.

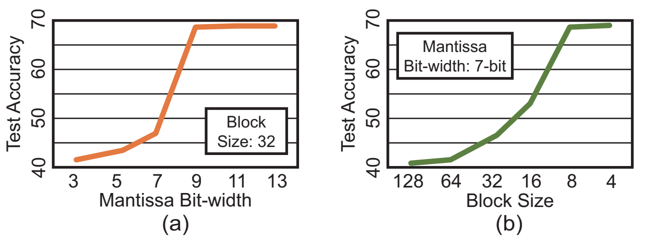


Fig. 2. Final test accuracy of ResNet-18 on ImageNet when trained with BFP formats having different (a) mantissa bit-widths and (b) block sizes.

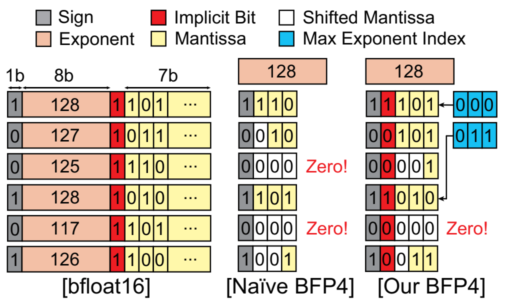


Fig. 3. An example of a tensor at different floating-point formats: i) `bfloat16`, ii) naïve BFP, and iii) our BFP format. Note that implicit bits (red) are not stored in the memory.

B. *Implication of DBPS Control on Training Efficiency*

The proposed DBPS has two parts: (i) block size scaling and (ii) precision scaling. For the block size scaling, we increase

[1]We define the precision of a BFP format with its 'sign+mantissa' bits. Thus, BFP4 has 1 sign bit and 3 mantissa bits (with 8-bit shared exponent).

TABLE I
ADDITIONAL BITS TO IDENTIFY THE LOCATIONS OF NUMBERS WITH THE MAXIMUM EXPONENT WITHIN A BLOCK

Block Size	32	64	128	256
BFP4	11.7	15.0	26.0	48.2
BFP8	11.0	15.4	29.4	66.9
BFP16	16.3	26.2	49.8	94.7

the block size when the learning rate decay happens, e.g., every 20 epochs. Compared to the block size scaling, we allow a more frequent change on BFP precision by monitoring the number of underflows for every 100 mini-batches to speed up the training when possible.

1) Impact of Block Size Scaling: Allowing a larger block size of BFP format reduces both the power consumption of computing logic and the required data size. The baseline of a bfloat16-based GEMM unit with 128 MACs consumes 163.8mW in 45nm technology (Fig. 4(a)). For the input and weight data, i.e., 128 operands each, 4KB of data is required. When we utilize the BFP8 format with block size of 8, the power consumption reduces by 63.7%, i.e., 59.5mW (Fig. 4(b)). Also, the required data size reduces by 43.8%, i.e., 2.25KB, by sharing an exponent for every eight operands. If we increase the block size to 64 instead of 8, the number of FP32 adders that are used to accumulate FP outputs reduces from 15 to 1 that additionally cuts down the power consumption by 14.6%, i.e., 50.8mW (Fig. 4(c)). As more operands are grouped together and share an exponent, the required data size further reduces by 9.7%. This data reduction is crucial since the energy consumed by accessing off-chip memory is significantly larger than running MAC units [6]. We explore these power/memory savings by changing the block size to a larger size in a coarse-grained manner.

2) Impact of Precision Scaling: Along with the block size scaling, the proposed DBPS allows precision scaling on the mantissa bit-width. The main objective of the precision scaling is the performance boost given by dealing with a larger number of MAC operations per clock cycle. Many prior works already have shown the effectiveness of precision-scalable MACs [7], [8]. However, the prior work only focus on the inference while our DBPS core focuses on the training with BFP-based computing. The INT MAC units in Fig. 4(b-c) can be designed with precision-scalable MACs that are equipped with INT2FP modules for the BFP support. Fig. 5 shows an example of a matrix-vector multiplication using a 16b$\times$16b precision-scalable MAC. In 16-bit mode, 16 INT4 MACs process one MAC operation at each cycle, e.g., Y0* = X0$\cdot$W0 where Y0* represents a partial sum of Y0. Thus, it takes 16 clock cycles to complete the multiplication. By reducing the precision to 8-bit, 4 MACs are grouped together to process one MAC operation. In the 8-bit mode, four MAC operations are processed at each cycle (4$\times$ speed-up). Note that two partial sums (Y0* and Y1*) are computed in parallel to share input datapath {X0, X1}. Similarly, we can expect another 4$\times$ performance boost by reducing the precision to 4-bit, i.e., 16 MAC operations per cycle. In the proposed DBPS scheme, we maximize the

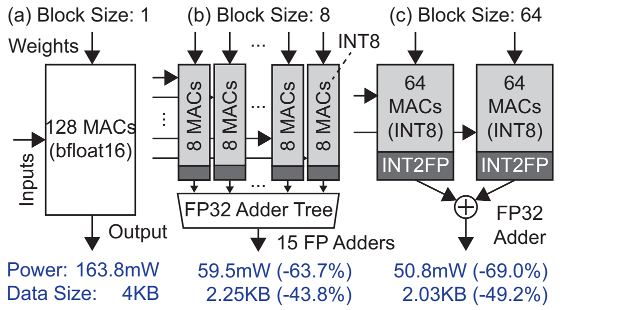


Fig. 4. MAC arrays at various data formats: (a) bfloat16, (b) BFP8 with block size 8, and (c) BFP8 with block size 64. The power numbers are estimated by synthesizing each array in 45nm technology.

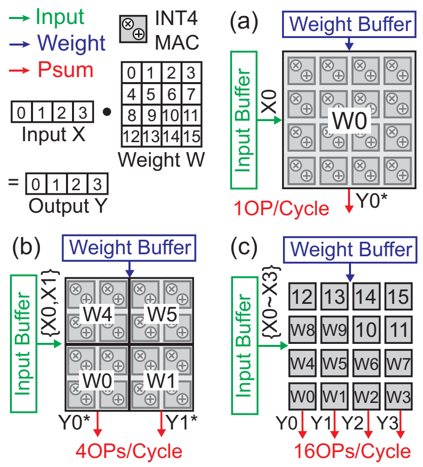


Fig. 5. An example of a matrix-vector multiplication showing the throughput improvement by scaling mantissa bits: (a) 16-bit (1$\times$), (b) 8-bit (4$\times$) and (c) 4-bit (16$\times$).

training speed but keeps the training accuracy by changing the precision mode in a fine-grained manner.

IV. Hardware Architecture

A. DBPS Core: A Training Accelerator with Block Floating Point Support

In this section, we present how DBPS is realized in our BFP-based training accelerator, named DBPS core. As shown in Fig. 6, the DBPS core consists of four sub-cores, a selective INT/FP adder tree, an output buffer, a batch norm (BN) unit, a ReLU-Pool unit, and an FP2BFP converter. Each sub-core can process '$N = 128$' INT4 MAC operations in Fig. 6, where the INT4 MACs are grouped to perform higher bit MAC operations, e.g., '$N/4$' INT8 operations. Two examples show how the sub-core can be configured to process different precision combinations of input and weight operands[2] (*precision scaling*). For instance, in X8W4, X8 means 8-bit mantissa for inputs and W4 refers to 4-bit mantissa for weights. Here, the mantissa bit includes the implicit bit as well (Section III-A). For higher bit computations, shifters representing the bit significance are selectively used. Also, the number of outputs varies depending on the weight precision to

[2]During the backpropagation, the input operand becomes the propagated local gradient and the weight operand becomes the transposed weight.

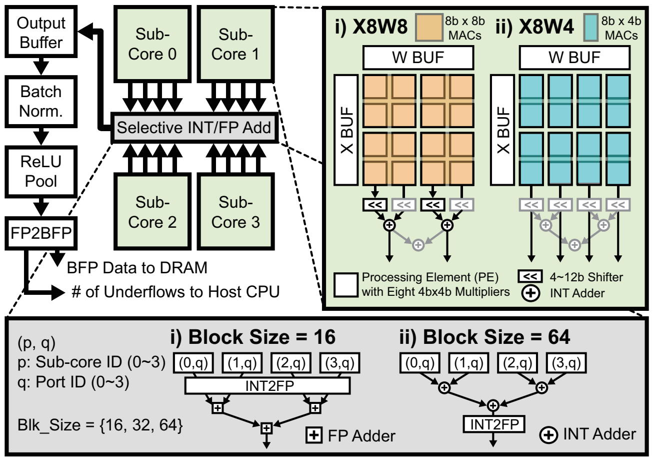


Fig. 6. The architecture of our DBPS core which consists of four sub-cores.

maximize the MAC utilization and data sharing. The sub-core outputs with the same port ID (q $=0\sim3$) are accumulated together to compute one partial sum. When computing the partial sum, an FP32 adder tree is used with the block size 16 since each sub-core processes operands with a different shared exponent. However, if the block size is scaled to 64, four sub-cores process the same block which shares a single exponent value (*block size scaling*). Thus, an INT32 adder tree is used instead. The INT2FP module in the selective INT/FP adder tree handles the shared exponents of two operand blocks. In our DBPS core, we perform data gating for adders that are not used at a certain block size and/or precision to remove the dynamic power consumption (Section III-B). The output buffer attached to the DBPS core queues 32 outputs from the core and passes the data to the BN unit at once when the buffer is full. The following BN and ReLU-Pool units have separate modules for forward and backward passes, respectively, since the operations behind each pass are different. Finally, the FP2BFP converter extracts the maximum exponent of a set of outputs that needs to be blocked together with a given block size for the computations in the next DNN layer. Then, mantissas in the block are shifted according to the extracted max exponent and quantized to a given precision. Meanwhile, the FP2BFP converter counts the number of underflows and returns it to the host processor for the DBPS control.

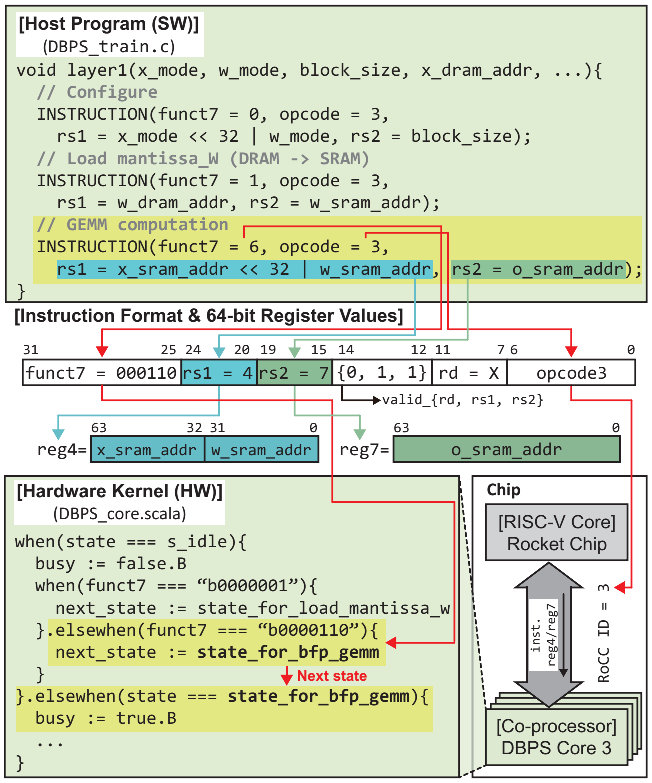
B. RISC-V ISA Extensions for DBPS Control

We implemented the DBPS core presented in Section IV-A as a co-processor attached to a RISC-V Rocket core [9]. As an on-chip co-processor, the DBPS core receives a custom 32-bit instruction via Rocket custom co-processor (RoCC) interface, and directly communicates with L1 data cache and DRAM of the Rocket core. Table II lists a set of custom instructions that is extended to configure and run the DBPS core for efficient DNN training. At the beginning, the DBPS core is configured by ‘dbps_config’. The ‘dbps_config’ sets the matrix size, the block size of BFP tensors, and the precision of each tensor (e.g., X and W). In other words, it groups/activates the required hardware resources, i.e., multipliers, adders, shifters, etc., depending on the block size and precision as depicted in Fig. 6. Then, the required BFP data is loaded from DRAM to a local SRAM by ‘mvin’ instructions. With the data being ready, the DBPS core processes DNN layers by instructions ‘bfp_gemm’, ‘batch_norm’, and ‘relu_pool’. The computation results are written back to the data cache by ‘mvout’. Finally, we initialize the state of an FSM controller by using ‘flush’ for the next operation.

Fig. 7 shows how the host program “DBPS_train.c” invokes the hardware kernel “DBPS_core.scala” for efficient DNN training. The DBPS core is implemented in Chisel, i.e., “DBPS_core.scala”. There are four important fields in the custom 32-bit instruction. First of all, the ‘opcode’ is used to specify one of available DBPS cores. Each DBPS core is assigned a core ID (equivalently, RoCC ID) so that the host program can identify it. In Fig. 7, we assume that the fourth core with RoCC ID = 3 is selected. Second, the ‘funct7’ field defines the extended instruction that lets the DBPS core know what to do next. Upon a given instruction, the FSM controller in the core sets the hardware state (next_state)

TABLE II
A LIST OF EXTENDED CUSTOM INSTRUCTIONS

Instruction	funct7	Instruction	funct7
dbps_config	0000	mvout	0101
mvin_man_w	0001	bfp_gemm	0110
mvin_exp_w	0010	batch_norm	0111
mvin_man_i	0011	relu_pool	1000
mvin_exp_i	0100	flush	1001


[Host Program (SW)]
(DBPS_train.c)
```
void layer1(x_mode, w_mode, block_size, x_dram_addr, ...){
  // Configure
  INSTRUCTION(funct7 = 0, opcode = 3,
    rs1 = x_mode << 32 | w_mode, rs2 = block_size);
  // Load mantissa_W (DRAM -> SRAM)
  INSTRUCTION(funct7 = 1, opcode = 3,
    rs1 = w_dram_addr, rs2 = w_sram_addr);
  // GEMM computation
  INSTRUCTION(funct7 = 6, opcode = 3,
    rs1 = x_sram_addr << 32 | w_sram_addr, rs2 = o_sram_addr);
}
```
[Instruction Format & 64-bit Register Values]
```
31       25 24     20 19     15 14      12 11     7 6         0
 funct7 = 000110 | rs1 = 4 | rs2 = 7 | {0, 1, 1} | rd = X | opcode3
                                      valid_{rd, rs1, rs2}

      63          32 31          0          63                0
reg4= x_sram_addr | w_sram_addr        reg7=     o_sram_addr
```
[Hardware Kernel (HW)]
(DBPS_core.scala)
```
when(state === s_idle){
  busy := false.B
  when(funct7 === “b0000001”){
    next_state := state_for_load_mantissa_w
  }.elsewhen(funct7 === “b0000110”){
    next_state := state_for_bfp_gemm
  }
}.elsewhen(state === state_for_bfp_gemm){
  busy := true.B
  ...
}
```

Fig. 7. An example on how the DBPS core (DBPS_core.scala) operates with the extended instructions programmed by the host (DBPS_train.c).

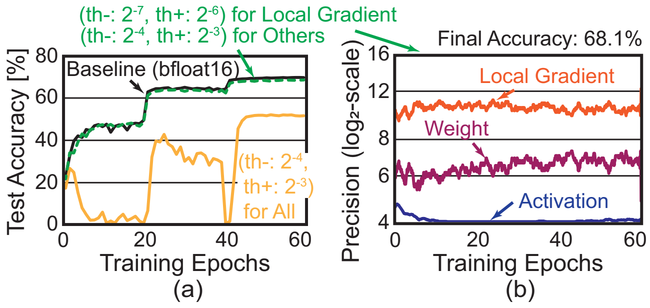


Fig. 8. (a) Accuracy of ResNet-18 on ImageNet at two different threshold conditions. The underflows on local gradients are more critical than other tensors. (b) Dynamic BFP precision on activations, weights, and local gradients.

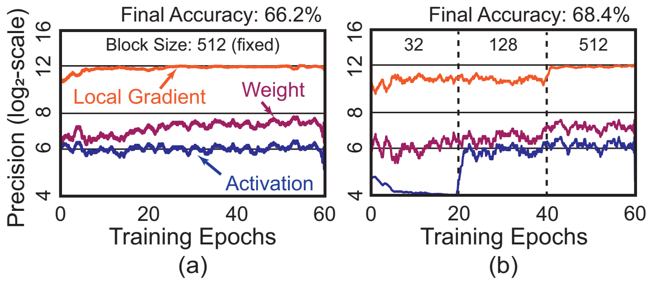


Fig. 9. The impact of block size on the BFP precision when training ResNet-18 on ImageNet: (a) a block size fixed at 512, (b) a block size gradually increased from 32 to 512.

TABLE III
FINAL TEST ACCURACY ON WIDELY USED DNN BENCHMARKS

Dataset	bfloat16	BFP16	BFP8	BFP4	DBPS
ResNet-18					
CIFAR-10	95.3	95.4	95.3	94.3	95.4
CIFAR-100	78.2	78.2	78.1	75.1	78.5
ImageNet	68.9	68.7	50.2	48.6	68.4
MobileNetV2-0.5					
ImageNet	65.2	64.3	63.0	0.1	64.5

to a new pre-defined state for that instruction. Among ten custom instructions provided in Table II, '`dbps_config`' is the most important instruction that sets up all the required control signals of the DBPS core for a given block size and precision. Also, it tells the core which training step is being handled, i.e., forward pass, backward pass, or weight update. Third, the '`rs1`, `rs2`' field locates registers in the host CPU that store input/output memory addresses, matrix size, batch size, block size, or precision. The last important instruction field is '`rd`' that identifies the register address in the host CPU at which the returned value from the DBPS core will be written. For instance, the number of underflows that is counted in the FP2BFP converter in the DBPS core will be used by the host CPU to determine the block size and precision for the next training iterations.

V. Experimental Results

A. Accuracy Analysis of DBPS Control

PyTorch-based BFP Trainer: We implemented custom CUDA kernels to train DNNs using various BFP formats. Given any PyTorch models, our BFP trainer detects and replaces convolution (e.g., `torch.nn.Conv2d`) and linear (e.g., `torch.nn.linear`) modules to '`bfp.Conv2d`' and '`bfp.linear`', respectively. Replaced modules do the same behavior as the original ones, but it emulates the FP2BFP converter and forms sub-tensor blocks with a given block size. To run our BFP trainer, a configuration file is provided to set the initial block size and BFP precision for each tensor type at each layer, e.g., input activations, weights, or local gradient.

DBPS Control Mechanism: For the precision control, we count the number of underflows at the FP2BFP converter. The underflow count is returned to the RISC-V core that determines the next BFP precision at every 100 training mini-batches (*fine-grained control*). We allow each tensor to select 4-bit, 8-bit or 16-bit as a 'sign+ mantissa' bit-width, which excludes the implicit bit. In the proposed DBPS, we utilize a hysteresis controller that reduces (or increases) the precision when the underflow ratio is below the lower threshold, th_- (or above the upper threshold, th_+). We empirically select the threshold values that are reciprocals of powers of 2, e.g., $th_- = 2^{-4}$ and $th_+ = 2^{-3}$. According to our experiments, other tensors are less sensitive to underflows except the local gradient. Fig. 8(a) shows training curves at different threshold conditions for the precision control with the block size of 32. We should keep the threshold extremely low, near 1% for local gradients, while the threshold for the others may stay close to 10%. As shown in Fig. 8(b), the required BFP precision for each tensor is well selected by the proposed scheme.

For the block size control, we allow a more *coarse-grained control* than the precision control. As the training is noisier with a larger learning rate, we keep the block size small at the beginning then double the block size whenever the learning rate decay happens (e.g., every 20 epochs). Fig. 9 shows how the precision changes with respect to the block size control. When we keep a large block size, i.e., 512, from the start to the end, the required mantissa bits for each tensor are higher (compare Fig. 9(a) with Fig. 8(b)). If we gradually increase the block size from 32 to 512 (Fig. 9(b)), we can keep the average BFP precision lower and get 2.2% higher accuracy. Note that the dynamic precision control successfully sets the appropriate mantissa bit-widths for any block sizes.

Training Accuracy: We evaluated the proposed DBPS control on ResNet-18 with CIFAR-10, CIFAR-100 and ImageNet. In addition, MobileNetV2 with ImageNet dataset is selected as another benchmark. As in Table III, the final accuracy after training with the proposed DBPS is almost identical to the `bfloat16` baseline for all benchmarks. If we fix the BFP precision to `BFP16`, the accuracy is preserved. However, when we simply reduce the precision to `BFP8` or `BFP4`, the accuracy significantly degrades by 5.3% or 22.4% on average.

B. Energy Analysis on DBPS Cores

To analyze the energy efficiency of the proposed DBPS, we performed energy analysis using an FPGA prototype implemented on a Xilinx Virtex-7 VC707 board. The implementation result of four DBPS cores with a RISC-V core is shown in Fig. 10(a). According to the synthesis result, the number

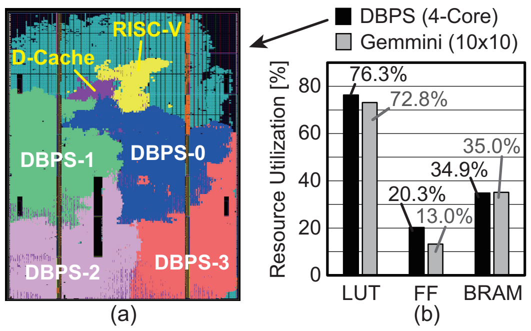


Fig. 10. (a) An FPGA implementation of four DBPS cores with a RISC-V processor. (b) Resource utilizations of four DBPS cores and a 10x10 Gemmini with a RISC-V core on a Xilinx Virtex-7 VC707.

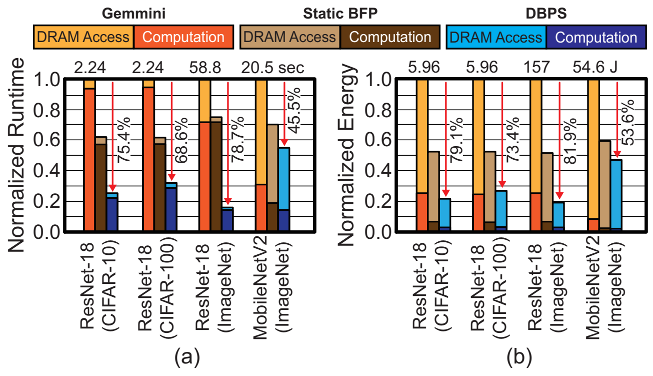


Fig. 11. Improvements in performance and energy efficiency of the proposed DBPS scheme on various benchmarks.

of MACs is limited by the available LUTs (Fig. 10(b)). As a baseline, we also implemented `bfloat16`-based Gemmini accelerator attached to a RISC-V core [10]. Similar hardware resources are utilized by the both designs for fair comparisons, where both operate at 200MHz. As a result, Gemmini has a small 10×10 systolic array, which is not sufficient for DNN training. On the contrary, four DBPS cores have 2,048 `INT4` MACs that can work as independent MAC units or group together to do 512 (or 128) `INT8` (or `INT16`) MACs. Thus, you can expect up to $20\times$ speed up when `BFP4` is used for the training. However, training only with `BFP4` fails due to a limited precision (Table III). Thus, the DBPS comes into play to safely set the minimum required precision for each tensor as explained in Section V-A.

Fig. 11(a) shows the normalized training time of one mini-batch (i.e., 128 for CIFAR datasets and 256 for ImageNet). The most of training time is consumed by GEMM operations (Computation) in ResNet-18. This is due to limited hardware resources on a small Virtex-7 FPGA and high data reuse rate. On the contrary, DRAM access time dominates the training time of MobileNetV2. This is due to lower data reuse rate and a huge amount of intermediate features in MobileNetV2 that is deeper and wider than ResNet-18. If we compare the training time of DBPS with Gemmini, it reduces by 67.1% on average. This performance improvement comes from the increase of effective MACs per cycle by reducing the `BFP` precision when necessary. We also compared the performance of DBPS with ones using a static `BFP` format. As a 'Static `BFP`', we used `BFP8` for activations/weights and `BFP16` for local gradients for the stable DNN training. The block size is fixed to 32 for 'Static `BFP`'. Compared to the Static `BFP`, DBPS improves the training throughput by 37.2% on average.

For the energy analysis, we used power numbers of DRAM and on-chip logics reported by Xilinx Vivado tool. Accessing DDR3 with 64-bit datapath on the VC707 board consumes 1.97W while Gemmini and DBPS cores consume 700mW and 303mW, respectively. As a result, the portion of DRAM access energy becomes dominant as shown in Fig. 11(b). By using the proposed DBPS, we can reduce the energy consumption by 72.0% and 47.7% on average when compared with Gemmini and Static `BFP`, respectively. The significant energy savings come from the reduced computation time and reduced data size that travels between DRAM and DBPS cores.

VI. Conclusion

In this work, we propose DBPS that automatically sets the minimum required `BFP` format for successful yet energy efficient DNN training. To support the DBPS control in hardware, a co-processor that computes `BFP`-based GEMM at various block sizes and `BFP` precisions is presented. For the deployment of DBPS cores, we extend RISC-V instructions and provide an example of a custom SW-HW interface. This full stack research shows the effectiveness of DBPS by demonstrating the implementation result on an FPGA prototype. Compared with the `bfloat16` GEMM accelerator, i.e., a typical systolic array, four DBPS cores can save 67.1% of training time and 72.0% of energy consumption at fixed hardware resources.

References

[1] T. B. Brown *et al.*, "Language models are few-shot learners," *arXiv:2005.14165*, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[2] D. A. Patterson *et al.*, "Carbon emissions and large neural network training," *arXiv:2104.10350*, 2021. [Online]. Available: https://arxiv.org/abs/2104.10350

[3] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training DNNs with hybrid block floating point," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[4] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 1740–1750.

[5] S. Qian Zhang, B. McDanel, and H. T. Kung, "Fast: Dnn training under variable precision block floating point with stochastic rounding," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 846–860.

[6] A. Pedram *et al.*, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design and Test*, vol. 34, no. 2, pp. 39–50, 2017.

[7] H. Sharma *et al.*, "Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.

[8] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "BitBlade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[9] K. Asanović *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016.

[10] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of Design Automation Conference (DAC)*, 2021.