Highly-Parallel CNN Accelerator for RepVGG-like Network Training on FPGAs

Chuliang Guo[*,†], Binglei Lou[‡], David Boland[‡], Philip H.W. Leong[‡]
[*] Harbin Institute of Technology, Harbin, China
[†] Zhejiang University, Hangzhou, China
[‡] The University of Sydney, Sydney, Australia

***Abstract*—Training convolutional neural networks (CNNs) at the edge has obtained growing significance for it enables adaptation to evolving environments without frequent data communication between cloud GPU clusters. However, edge-based training faces challenges regarding limited computation resources and memory bandwidth. In this paper, we propose a generic FPGA-based training accelerator tailored for RepVGG-like networks [1], which strikes a balance between maximizing training-time accuracy and minimizing inference-time latency. The proposed accelerator leverages fine-grain channel-level parallelism within computational units specially designed for multiple branches of the basic building block within the RepVGG-like network. Specifically, we employ a Conv block for forward Conv and backward deConv, along with a dilated Conv block including a weight kernel partition scheme for efficient weight gradient calculation. Furthermore, we aggressively exploit a 2-stage coarse-grain task-level parallelism for low-latency CNN training: (1) parallelism among multiple branches of the basic building block of RepVGG, and (2) parallelism between error back-propagation and weight gradient calculation in the backward path. This achieves an end-to-end training throughput comparable to the peak performance of the most computation-intensive Conv block. Through experiments on the CIFAR-10 dataset using 16-bit fixed-point arithmetic, we demonstrate state-of-the-art batch 1 throughput of 150 GOPs for training and 183 GOPs for inference. This highlights the effectiveness and efficiency of CNN training for lightweight networks on edge FPGAs.**

***Index Terms*—CNN Training, RepVGG, FPGA, HLS**

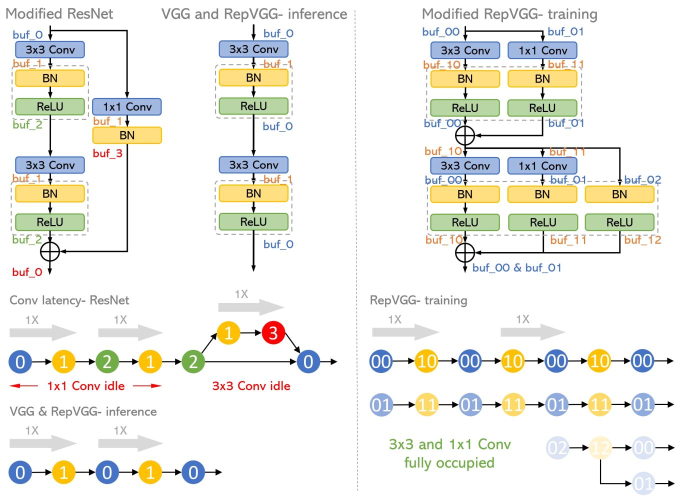


Fig. 1: In BOOST [8], 3×3 and 1×1 Conv are sequentially processed, and one of the Conv blocks is idle at any time (left). This work uses RepVGG to expose task-level parallelism among the branches. Three groups of static buffers are employed to fully utilize the Conv blocks (right).

I. Introduction

Convolutional neural networks (CNNs) have found extensive deployment in edge devices for various critical tasks including image classification [2], object detection [3], language processing [4], and autonomous driving [5]. The rise in concerns regarding latency, energy efficiency, and data privacy has led to the increasing prevalence of lightweight CNN networks for edge devices. However, these lightweight networks are often less adept at handling diverse environments and applications without the need for fine-tuning or re-training. GPUs are an excellent solution for many data-center applications where maximum acceleration without extreme power constraints is required. However, FPGAs have distinct advantages: (1) for embedded applications combining some other functionality with machine learning (ML) training, *e.g.* single FPGA solutions integrating a video interface, video decompression, and ML which utilize the FPGA's massive on-chip bandwidth for low size, weight and power, and (2) for applications where energy consumption is more important than speed, *e.g.* battery-powered space applications that do not require extremely high training speeds.

Multi-branch CNNs such as ResNet [6] are widely employed in practice as they offer accuracy advantages over non-residual approaches such as VGG [7]. In a previous publication, we proposed a generic training accelerator BOOST [8], composed from multiple computational units where the synthesis was to invoke a single instance of each type of layer, *e.g.,* convolution (Conv), fully connected (FC), and normalization. This maximizes the reuse of on-board DSP and LUT resources. Compared with prior work that employed generic hardware for distinct layers, such a design strategy balances hardware efficiency and resource consumption. Residual connections, however, introduce data dependencies inside the basic block, resulting in inefficient utilization of the computational units. As shown in Fig. 1, we must process the 2 branches sequentially due to data dependencies and different computation speeds for the 1×1 Conv compared with the 3×3 Conv. This leads to a reduction in performance compared with simpler VGG-like networks.

To address the problem described, we consider RepVGG [1] which decouples inference- and training-time network architectures to achieve similar accuracy to ResNet. Training employs a residual architecture for accuracy, but prior to inference, the resulting neural network is transformed into an equivalent VGG-like network comprising only 3×3 Conv operations VGG [1]. In this work, our strategy is based on optimizing the basic block to achieve high throughput and high utilization of computational units. Specifically, each branch of RepVGG's basic block has at most one Conv, thus eliminating data-dependent computations. As shown in Fig. 1, this allows us to allocate 3 groups of 2 static buffers working in a double-buffered manner. The 3×3 and 1×1 Conv operations are never idle, and full utilization of the computational resources is achieved. We also offload input activations, weights, and momentum velocities to DDR memory and, therefore, have adequate BRAM resources for static buffers. This strategy overcomes limitations in batch size and on-chip storage of input activations present in BOOST [8] and is thus capable of complete full back-propagation-based training beyond transfer learning. Therefore, this work has better utilization of processing engines, and supports larger neural networks and batch sizes as storage of weights and input activations is off-FPGA.

This is the first work to integrate RepVGG training on an FPGA. Our main contributions are as follows:

[*]Chuliang Guo was formerly at Zhejiang University and is now with Harbin Institute of Technology.

- We propose an FPGA-based generic training accelerator comprising distinct computational units with fine-grain channel-level parallelism for various layer operations, which targets high throughput and low latency for multi-branch networks.
- Two Conv blocks are introduced for efficient Conv/deConv and dilated Conv respectively, enabling coarse-grain task-level parallelism (1) within the multi-branch basic block, and (2) between error back-propagation and gradient calculation, leading to a computational throughput similar to the peak throughput of 3×3 Conv blocks.
- We evaluate the proposed RepVGG-like training accelerator on the CIFAR-10 dataset [9] using 16-bit fixed-point arithmetic, achieving the state-of-the-art normalized throughput at batch 1 of 150 GOPs for training and 183 GOPs for inference.

II. Preliminaries

CNN training using the back-propagation algorithm [10] and SGD optimizer [11] as in Fig. 2 includes (1) forward path, (2) backward path, (3) gradient calculation, and (4) weight update.

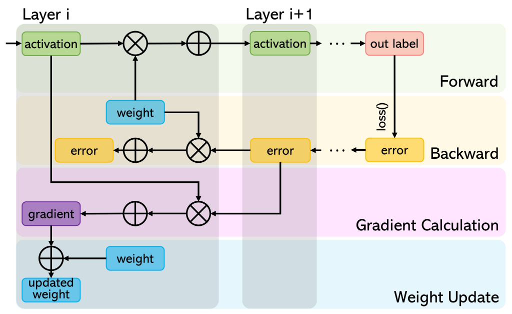


Fig. 2: SGD-based back-propagation training workflow.

The forward path is known as *inference* which involves the flow of input activations from the initial to the final layer. Convolution (Conv) performs a sliding window dot product operation between weight kernels and input activations, accumulating partial sums along channels. Batch normalization (BN) applies an affine transformation, and ReLU replaces negative values with zeros. AvgPool reduces the size of the feature map by replacing pixels with their average. FC applies a linear transformation on its input activations to generate a predicted label to compare with the categorical ground-truth label.

The backward path propagates the error (*i.e.,* neural gradient) in the reverse direction to the forward path. During this process, backward deConv follows the same connectivity pattern as the forward Conv operation, but with 180°-rotated weights [12]. In cases with a non-unit stride, errors are initially dilated with zeros between adjacent pixels. Normalized input activations are reused in the BN layers. ReLU uses the same 0/1 mask as in the forward path, while AvgPool replaces a single pixel with multiple pixels representing their average. FC maintains the same computational pattern as the forward path except for the transposed linear weights.

Weight gradients are calculated by employing dilated Conv between forward input activations and backward errors [13], [14]. This back-propagation method eliminates the need for repetitive gradient derivation using the chain rule for each layer. When dealing with non-unit stride, errors are dilated with zeros between neighboring pixels and enlarged to match the receptive field size of the input activations in the previous layer. Dilated Conv operates independently on input/error feature maps from different channels.

Once the weight gradients are generated, network parameters, including weights and biases, can be promptly updated. The SGD optimizer with momentum is utilized, considering both current and past gradients to determine the subsequent direction of gradient descent. Momentum velocities accumulated by weight gradients per iteration are also stored off-chip.

III. Proposed Accelerator

A. Overall Architecture

To maximize the hardware reuse and flexibility to support various CNNs, the proposed training accelerator adopts a general architecture with at most one instance of specified computation block for each layer operator of the RepVGG-like, including 3×3 and 1×1 Conv, deConv, and dilated Conv in stride 1 and 2 (as down/up-sampling blocks when layer switching), AvgPool, FC, BN, ReLU, and shortcut addition. Note that although we target a RepVGG-like network as an example, our accelerator can be employed for various networks such as VGG and ResNet as we support all their arithmetic operators.

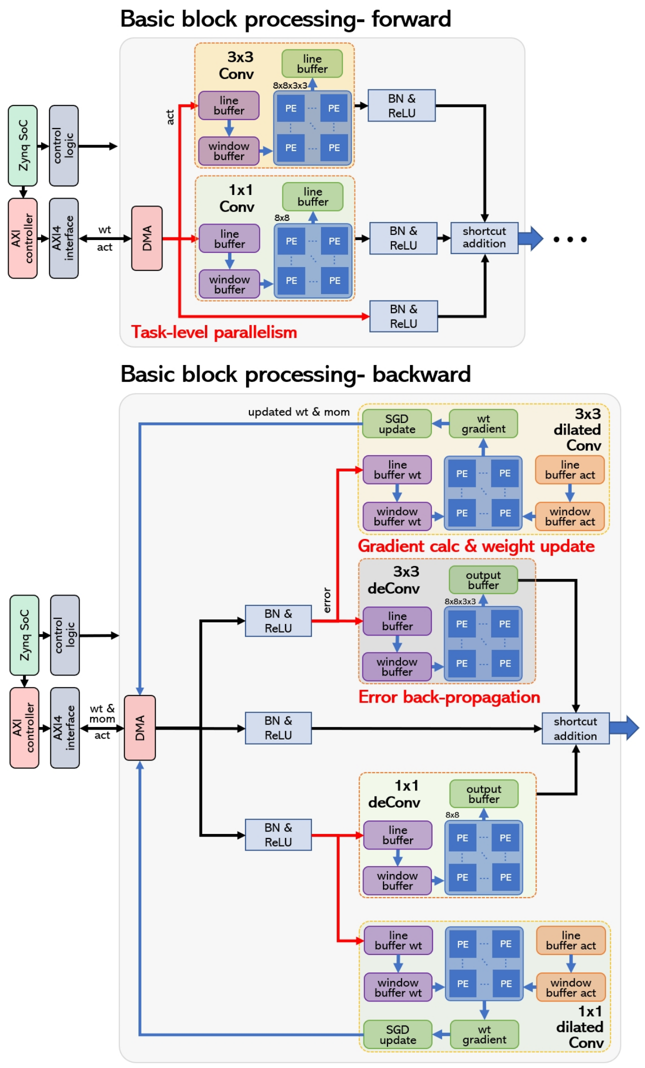


Fig. 3: Overall architecture of the HLS-based RepVGG-like training accelerator. Besides fine-grain channel-level parallelism within computational blocks, we arrange coarse-grain task-level parallelism within multi-branch RepVGG-like basic blocks, and sequentially process Conv, fused BN&ReLU, and shortcut addition. In the backward path, the 2nd-stage coarse-grain task-level parallelism is implemented for weight gradient calculation and error back-propagation, where dilated Conv and deConv are simultaneously executed.

We facilitate coarse-grain task-level parallelism among 3×3 Conv, 1×1 Conv, and identity branches by employing 3 groups of static buffers for the RepVGG-like basic block. 3×3 Conv and deConv dominate the critical path in the RepVGG basic block, and therefore, the critical latency could be hidden in the main 3×3 Conv branch. Fine-grain channel-level parallelism is employed within computational blocks such as Conv blocks that simultaneously execute multiple input channels and produce corresponding output channels. This minimizes the initiation interval and has a minor impact on throughput when processing a small batch size or feature map [15].

We further implement the 2nd-stage task-level parallelism between error back-propagation and weight gradient calculation. This allows the latency in the backward path to be halved compared with sequential processing in prior works [13], [14]. Fig. 3 illustrates the execution of the modified basic building block in the forward path with a 2-stage coarse-grain task-level parallelism. In both forward and backward paths, the RepVGG-like network is processed layer-by-layer with 3×3 Conv and BN&ReLU, 1×1 Conv and BN&ReLU, and identity BN&ReLU being executed concurrently using 3 groups of static buffers. Shortcut addition is performed after processing the three branches and accounts for a significant portion of the overall latency throughout the training time.

On the other hand, BN in the backward path has a much longer latency since it is not a simple affine transformation. To balance the forward and backward latency and reuse computation blocks as much as possible, we move the ReLU in the basic building block of RepVGG before the shortcut addition and thus fuse BN and ReLU. No accuracy degradation is observed by such a modification.

In BOOST [8], we utilize unified bm(2,5) as a special case for transfer learning, while for more general cases with higher dynamic range, bm(4,3) and bm(5,2) are favored in backward [16]. However, such hybrid precision requires a multiplier/adder/MAC capable of handling all these data formats or separate ones, which would result in idle components for a significant proportion of the time. We therefore employ a more general 16-bit fixed-point that has been natively supported by the AMD Xilinx HLS math library with similar accuracy to FP32 in the RepVGG-like network, *i.e.,* 89.69% top-1 accuracy on CIFAR-10 at batch 4, compared with 89.93% of FP32.

B. *Conv Block Designs*

As illustrated in Alg. 1 and Fig. 3, there are 3 Conv patterns during forward, backward, and gradient calculation, referred to as Conv, deConv, and dilated Conv respectively [12]. Conv and deConv share the same connectivity pattern between feature maps and weights where partial sums from different input channels are accumulated, while different weight filters correspond to independent output channels. However, dilated Conv exhibits a different convolution pattern from Conv and deConv:

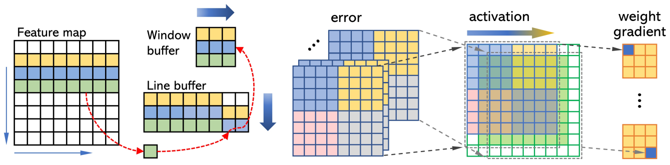
- Dilated errors as weights are in the same size as input feature maps, *e.g.,* from 4×4 to 32×32 for RepVGG-like on CIFAR-10;
- Partial products from different mini-batches (instead of input channels) are accumulated, and thus dilated Conv usually features batch-level as opposed to channel-level parallelism.

This highlights the necessity to employ separate Conv blocks for Conv/deConv and dilated Conv. We first propose a Conv block handling both patterns to maximize hardware reuse, by employing zero dilation before discrete Conv in the backward path. Using the `HLS pipeline` and `HLS array_partition` optimization directives, feature map dimensions (*i.e.,* width and height of activations) are outside the unrolled HLS `for` loops and output stationary dataflow [17] is adopted for reduced data communications. To minimize the latency and energy consumed during data fetching, we utilize line buffer and window buffer techniques to locally buffer input activations as shown in Fig. 4(a). For non-unit stride circumstances, we conduct unit-stride Conv and discard output feature map pixels every *stride* steps in the forward direction, dilate *stride* number of zeros between the error, and then perform unit-stride Conv in the backward propagation. This method guarantees a balanced latency between the forward Conv and backward deConv while processing equal-size feature maps.

Algorithm 1: Conv patterns in SGD-based back-prop training.

```
Require: activation $A$, weight $W$, error $E$, weight gradient $G$, stride $s$, padding $p$.
/* Forward- Conv                                              */
$A^l[c_o][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
$W^l[c_o][c_i][h_k][w_k] * A^{l-1}[c_i][h * s + h_k - p][w * s + w_k - p]$;
/* Backward- deConv                                           */
$E^{l-1}[c_o][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
$W^l_{rot}[c_o][c_i][h_k][w_k] * E^l[c_i][h/s + h_k - p][w/s + w_k - p]$;
/* Weight gradient- dilated Conv                              */
$G^l[c_o][c_i][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
$E^l[c_o][h_k][w_k] * A^{l-1}[c_i][h/s + h_k - p][w/s + w_k - p]$;
```

For weight gradient calculation, we propose a dilated Conv block aiming at optimized data communication and low latency. As shown in Alg. 1, when the error (*i.e.,* convolutional weight in dilated Conv) is same-sized as the input feature map, loading in the entire error at an initiation interval of one is impractical because I/O ports can only read/write once per clock cycle. To solve this problem we present a weight kernel partition scheme for dilated Conv and utilize an additional group of line buffers and window buffers for local buffering of errors. Taking the final-layer feature maps of the RepVGG-like in CIFAR-10 training as an instance, as illustrated in Fig. 4(b), an 8×8-sized error is mapped into multiple non-overlapping 4×4 smaller regions for local buffering. The partition region could be principally determined with an arbitrary size. A small region requires frequent window buffer updates and results in the under-utilization of MAC units, while a large region would consume significant IO resources in pipeline allocation and make the initial interval of one non-trivial. We select a partitioning kernel size of 4×4 to achieve a similar latency as forward Conv and backward deConv when processing the same-size input feature maps. This ensures neither of the Conv blocks degrades the backward performance of the coarse-grain task-level parallelism between error back-propagation and weight gradient calculation.



(a) Local buffering. (b) Weight kernel partition.

Fig. 4: Conv block details: (a) local buffering of input activations (and partitioned error as weight kernels) in Conv and dilated Conv blocks, consisting of 8 parallel channels of line buffers and window buffers; (b) weight kernel partition in dilated Conv. Error as weight kernel is partitioned into four 4×4 smaller ones for dilated Conv.

The indexing of feature maps determines whether to incorporate partial sums of partitioned errors, a process commonly utilized in HLS `for` loops of the sliding window Conv. As a unit-stride instance in Fig. 4, the partial sum in blue at step 0 contributes to the weight gradient pixel [0][0], while at the next sliding window step, it contributes to the weight gradient pixel [0][1]. The partitioned error in yellow further contributes to the weight gradient pixel [0][0] after 4 sliding window steps (*i.e.,* partitioned kernel size). Weight gradient buffers are reinitialized right after velocity accumulation following the pipeline processing of dilated Conv and weight update. To hide the latency of multiple branches of the RepVGG-like network, we

TABLE I: Performance comparison with prior FPGA-based CNN training accelerators.

	[18]	[19]	[13]		[20]	[14]		[8]		**Ours**
FPGA Device	ZU19EG	ZCU111	Stratix 10 MX		MAX5	VC709		ZCU102		ZCU102
Data Format	FP32	INT8	FP16		INT8	PINT8		BM8		Fixed16
Freqency (MHz)	200	180	185		200	200		225		225
CNN Network	LeNet10	VGG16	VGG-like	ResNet20	VGG-like	ResNet20	VGG-like	VGG-like	ResNet20	RepVGG-like
DSP	1500(76%)	1037(25%)	1046(26%)	1040(26%)	6241(91%)	1728(48%)		373(15%)	502(20%)	864(34%)
LUT/ALM	33K(63%)	73K(17%)	221K(31%)	239K(34%)	679K(57%)	132K(30%)		147K(54%)	189K(69%)	158K(58%)
BRAM/M20K	174(18%)	1045(97%)	2998(44%)	2558(37%)	1232(29%)	240(14%)		1255(69%)	1735(95%)	818(45%)
Normalization	-	-	-	-	-	L1-FRN	L1-FRN	BN	BN	BN
Batch Parallelism	-	1	1	1	128	16	16	1	1	1
Batch 1 Tput. (GOPs)	86	20	160	180	11 (1417†)	38 (611†)	41 (659†)	209	131	150 (183*)
Power (W)	14.2	-	20	20	13.5	8.4	8.6	7.7	8.7	7.2 (4.9*)
Eff. (GOPs/W)‡	6.1	-	8	9	0.8	4.5	4.7	27.1	15.1	20.8 (37.3*)
Eff. (GOPs/#DSP)‡	0.057	0.019	0.15	0.17	0.002	0.02	0.02	0.56	0.26	0.17 (0.23*)

† Total throughput of parallel mini-batches.
‡ Normalized throughput at batch 1.
* Inference-time RepVGG-like network.

also apply task-level parallelism to 3×3 and 1×1 dilated Conv of channel parallelism and output stationary dataflow.

IV. Experimental Results

We synthesize and implement the proposed training accelerator using Vivado Design Suite 2019.2 and evaluate the on-board performance on an AMD Xilinx ZCU102 FPGA. This board is equipped with an embedded ARM CPU, 274K LUTs, 2520 DSPs, and 64 Mb BRAMs. The representative RepVGG-like network is in a hierarchy of 16C-16C-32C-32C-32C-64C-64C-64C-64C-AvgPool-FC. Channel parallelism is explored and images from different batches are sequentially processed, *i.e.,* at batch parallelism of 1. Off-FPGA storage of weights and input activations delivers the same throughput at arbitrary batch sizes.

Algorithm 2: Channel tiling and task-parallelism in RepVGG.

```
Require: activation, $A$; error, $E$; weight, $W$; velocity, $V$;
  #input/output channel, $ch\_in, ch\_out$; channel parallelism,
  $ch\_t = 8$;
...
/* layer_1_0 Conv1 forward                                  */
$ch\_in = ch\_out = 16$;
for $c\_out = 0$ to $ch\_out/ch\_t - 1$ do
    activation index update;
    for $c\_in = 0$ to $ch\_in/ch\_t - 1$ do
        function load_weight $(W)$;
        /* task-level parallelism                           */
        function $3 \times 3$ Conv $(A, W)$;
        function $1 \times 1$ Conv $(A, W)$;
        % identity branch $(A)$;
    end
    function BN&ReLU $(A)$;
end
function shortcut_addition $(A, A)$;
...
/* layer_1_0 Conv1 backward                                 */
$ch\_in = ch\_out = 16$;
for $ch\_out/ch\_t - 1$ to $c\_out = 0$ do
    activation index update;
    function BN&ReLU $(A)$;
    for $ch\_in/ch\_t - 1$ to $c\_in = 0$ do
        function load_weight $(W, V)$;
        /* task-level parallelism                           */
        function $3 \times 3$ deConv $(E, W)$;
        function $1 \times 1$ deConv $(E, W)$;
        function $3 \times 3$ dilated Conv $(A, E)$;
        function $1 \times 1$ dilated Conv $(A, E)$;
        % identity branch $(A)$;
    end
end
function shortcut_addition $(A, A)$;
```

For functional verification, we present the training loss curve on the CIFAR-10 dataset for 200 iterations, as shown in Fig. 5, with streaming CIFAR-10 images using a learning rate of 0.05 and a momentum of 0.9 for the RepVGG-like network. The software plots the GPU simulation with FP32 numerical precision and generic algorithm description of Conv patterns, which computes BN for the entire output channels in the Conv layer at once. The hardware plots the execution of the proposed accelerator using 16-bit fixed-point arithmetic in end-to-end SGD-based back-propagation training. Each layer of the RepVGG-like network is processed sequentially using channel tiling, as shown in Alg. 2. The hardware and channel-tiling converge slightly faster since BN is performed after each tiled output channel group of Conv. *i.e.,* partial sums from 8 Conv channels are normalized and added to other normalized channel groups, and the use of 16-bit fixed-point arithmetic uses half the bandwidth and much simpler arithmetic units.

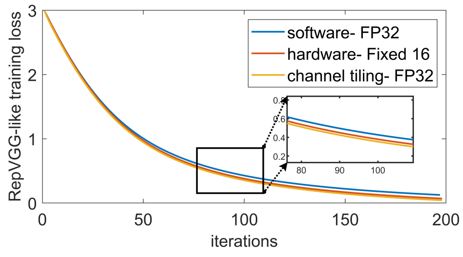


Fig. 5: Training loss using streaming CIFAR-10 images.

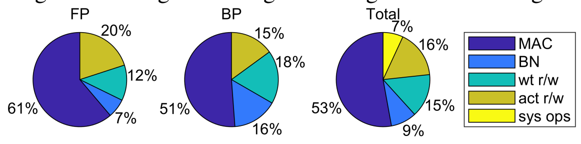


Fig. 6: Latency breakdown of the RepVGG-like training accelerator at 225 MHz, including computation and external memory access.

Constrained by the limited scale of pipeline regions in Vivado HLS tool, the highest achievable channel parallelism is 8, although sufficient resources are available to exceed 8. This implies simultaneous processing of 8 input channels and 8 output channels, and therefore, requires 8×8×3×3 PEs in 3×3 Conv block as shown in Fig. 3. This configuration achieves a peak throughput of approximately 250 GOPs. DSP48E2s are primarily used for MAC units in computational blocks 3×3 Conv and 1×1 Conv and index calculations. Dilated Conv blocks are primarily synthesized with LUTs for addition and multiplication to save DSP resources (as dilated Conv blocks remain idle during inference). However, with a maximum bandwidth of 38.4 Gbps on the ZCU102 board, the design becomes memory-bound and incurs additional latency for external memory access. To address these challenges, we employ task-level parallelism and organize the 3×3 and 1×1 Conv branches accordingly. Weights, activations, and momentum velocities are packed in 128 bits and read/written through different AXI4 bundles and buses, which fully saturates the DDR4 bandwidth and prevents undue data access delays.

Fig. 6 illustrates the latency breakdown during back-propagation training for processing the last training image of a batch in the RepVGG-like network, as it minimizes iteration latency from when an event occurs to the weight update. The breakdown components include the forward path, backward path (involving error back-propagation, weight gradient calculation, and weight update), and the total latency. Additionally, the total latency considers the time for reading/writing weights (*i.e.,* wt r/w) and activations (*i.e.,* act r/w) from/to off-chip DDR4 memory and the system operation (*i.e.,* sys ops) time related to the communication with the ARM processor. The latency is measured directly on the ZCU102, and the proportion of time for MAC (including Conv and element-wise shortcut additions) and BN is obtained from the co-simulation report.

External memory access in the forward path includes reading Conv weights from DDR4 in the PS logic and writing inputs of Conv and BN layer to off-chip. While during the backward path, it takes time for reading and writing weights and momentum velocities, and reading BN and Conv inputs. Backward BN takes more time than forward as gradients of BN weights and biases are generated before error back-propagation, and used for weight update. The total latency additionally accounts for Zynq system operations, such as communications with the ARM processor through the AXI4 bus.

Training-time MAC computation achieves a batch 1 throughput of 150 GOPs with external memory access in an end-to-end iteration. During inference, 1×1 Conv and identity branches, as well as shortcut additions, are by-passed where weights and biases are structurally re-parameterized to the main branch as in [1], *e.g.,* a plain structure in the forward path with merely 3×3 Conv followed by BN&ReLU. The inference-time RepVGG-like network requires off-chip weight fetching but with no activation or momentum velocity access. This leads to a batch 1 throughout of 183 GOPs with external memory access in an end-to-end iteration. Power consumption was measured with Xilinx Vivado and Maxim Digital Power Tool.

Table I compares the proposed accelerators with prior CNN training works [8], [13], [14], [18]–[20] primarily focusing on VGG-like and ResNet networks. This is a relatively fair comparison since (1) training-time RepVGG-like has similar building blocks of multi-branch and residual connections to the ResNet series, and inference-time RepVGG-like follows the same plain network architecture as the VGG series; (2) these accelerators have single-engine rather than streaming architectures so resource consumption does not change with the size of the network. Despite an overall throughput that may lag behind batch parallelism works [14], [20], where several mini-batches are processed concurrently in a single pass, we achieve the second-highest training-time 150 GOPs (due to doubled element-wise shortcut additions than ResNet and increased data access time of DDR4 compared to HBM2 in [13]) and inference-time 183 GOPs for normalized batch 1 throughput. Compared with training on an Nvidia RTX 2080Ti GPU using PyTorch v1.6.0, we observe GPU throughputs of 49, 90, and 145 GOPs at batch sizes of 2, 4, and 8 due to under-utilization of tensor cores (with a peak throughput of 448 GOPs at batch 512 and above), which are below the 150 GOPs of the FPGA-based proposal. This highlights the efficacy and efficiency of low-batch CNN training for lightweight networks on edge devices.

V. Conclusion

We present an FPGA-based CNN training accelerator tailored for RepVGG-like networks in back-propagation training. To enhance training efficiency besides fine-grain channel-level parallelism within computational units, we employ coarse-grain task-level parallelism to simultaneously process multiple branches within the RepVGG-like basic building block, which effectively conceals critical latency into 3×3 Conv and reduces performance degradation due to the less computation-intensive arithmetic. We aggressively exploit coarse-grain task-level parallelism between error back-propagation and weight gradient calculation, resulting in similar latency in forward and backward paths. This decoupling of the training-time and inference-time architecture of the RepVGG-like network leads to a significant improvement in end-to-end throughput, achieving a throughput around the peak performance of the Conv block and, therefore maximizing hardware efficiency. Our future work will focus on exploring applications in online/continuous learning, with specific emphasis on semi-supervised and unsupervised approaches.

References

[1] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, J. Sun, RepVGG: making VGG-style convnets great again, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 13733–13742.

[2] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, W. Xu, CNN-RNN: a unified framework for multi-label image classification, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2285–2294.

[3] S. Ren, K. He, R. Girshick, J. Sun, Faster R-CNN: towards real-time object detection with region proposal networks, IEEE Transactions on Pattern Analysis and Machine Intelligence 39 (6) (2016) 1137–1149.

[4] W. Yin, H. Schütze, B. Xiang, B. Zhou, ABCNN: attention-based convolutional neural network for modeling sentence pairs, Transactions of the Association for Computational Linguistics 4 (2016) 259–272.

[5] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, J.-M. Frahm, Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: addressing an industrial challenge, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, 2019, pp. 305–317.

[6] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[7] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

[8] C. Guo, B. Lou, X. Liu, D. Boland, P. H. Leong, C. Zhuo, BOOST: block minifloat-based on-device CNN training accelerator with transfer learning, in: 2023 IEEE/ACM International Conference on Computer-Aided Design, IEEE, 2023, pp. 1–9.

[9] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images (2009).

[10] C. F. Higham, D. J. Higham, Deep learning: an introduction for applied mathematicians, Siam Review 61 (4) (2019) 860–891.

[11] L. Bottou, Stochastic gradient descent tricks, in: Neural networks: tricks of the trade, Springer, 2012, pp. 421–436.

[12] V. Dumoulin, F. Visin, A guide to convolution arithmetic for deep learning, arXiv preprint arXiv:1603.07285 (2016).

[13] S. K. Venkataramanaiah, H.-S. Suh, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, J.-s. 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, 2020, pp. 1–8.

[14] J. Lu, C. Ni, Z. Wang, ETA: an efficient training accelerator for DNNs based on hardware-algorithm co-optimization, IEEE Transactions on Neural Networks and Learning Systems (2022).

[15] Y. Tang, X. Zhang, P. Zhou, J. Hu, EF-train: enable efficient on-device CNN training on FPGA through data reshaping for online adaptation or personalization, ACM Transactions on Design Automation of Electronic Systems 27 (5) (2022) 1–36.

[16] S. Fox, S. Rasoulinezhad, J. Faraone, P. Leong, et al., A block minifloat representation for training deep neural networks, in: ICLR, 2021.

[17] Y.-H. Chen, J. Emer, V. Sze, Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks, ACM SIGARCH Computer Architecture News 44 (3) (2016) 367–379.

[18] Z. Liu, Y. Dou, J. Jiang, Q. Wang, P. Chow, An FPGA-based processor for training convolutional neural networks, in: 2017 International Conference on Field Programmable Technology, IEEE, 2017, pp. 207–210.

[19] S. Fox, J. Faraone, D. Boland, K. Vissers, P. H. Leong, Training deep neural networks in low-precision with high accuracy using FPGAs, in: 2019 International Conference on Field-Programmable Technology, IEEE, 2019, pp. 1–9.

[20] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, C. Guo, Towards efficient deep neural network training by FPGA-based batch-level parallelism, Journal of Semiconductors 41 (2) (2020) 022403.