FPGA-based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory

Shreyas K. Venkataramanaiah
Arizona State University
skvenka5@asu.edu

Han-Sok Suh
Arizona State University
hsuh6@asu.edu

Shihui Yin
Arizona State University
syin11@asu.edu

Eriko Nurvitadhi
Intel Corporation
eriko.nurvitadhi@intel.com

Aravind Dasu
Intel Corporation
aravind.dasu@intel.com

Yu Cao
Arizona State University
Yu.Cao@asu.edu

Jae-sun Seo
Arizona State University
jaesun.seo@asu.edu

ABSTRACT

Training convolutional neural networks (CNNs) requires intensive computations as well as a large amount of storage and memory access. While low bandwidth off-chip memories in prior FPGA works have hindered the system-level performance, modern FPGAs offer high bandwidth memory (HBM2) that unlocks opportunities to improve the throughput/energy of FPGA-based CNN training. This paper presents a FPGA accelerator for CNN training which (1) uses HBM2 for efficient off-chip communication, and (2) supports various training operations (e.g. residual connections, stride-2 convolutions) for modern CNNs. We analyze the impact of HBM2 on CNN training workloads, provide a comprehensive comparison with DDR3, and present the strategies to efficiently use HBM2 features for enhanced CNN training performance. For training ResNet-20/VGG-like CNNs for CIFAR-10 dataset with low batch size of 2, the proposed CNN training accelerator on Intel Stratix-10 MX FPGA demonstrates 1.4/1.7X energy-efficiency improvement compared to Stratix-10 GX FPGA with DDR3 memory, and 4.5/9.7 X energy-efficiency improvement compared to Tesla V100 GPU.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; **Application-specific VLSI designs**.

KEYWORDS

Convolutional neural networks, neural network training, back-propagation, hardware accelerator, FPGA

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415643

ACM Reference Format:
Shreyas K. Venkataramanaiah, Han-Sok Suh, Shihui Yin, Eriko Nurvitadhi, Aravind Dasu, Yu Cao, and Jae-sun Seo. 2020. FPGA-based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3400302.3415643

1 INTRODUCTION

Convolutional neural networks (CNNs) are extensively adopted in computer vision applications [1–3]. The training tasks of CNNs are commonly performed with GPUs using a mini-batch stochastic gradient descent (SGD) optimizer. To improve the CNN accuracy, higher batch sizes are employed for CNN training with GPUs, but this demands an excessive amount of memory and limits the capability to explore large models and tasks with high input resolution [4]. In addition, although GPUs provide very high throughput on CNN training with large batch sizes, they suffer from low utilization/throughput for smaller batch sizes [5]. This can be seen in Fig. 1, which reports the latency and Tesla V100 GPU utilization across different batch sizes for the task of training ResNet-20 CNN [6] for CIFAR-10 [7] dataset. Recently, new CNN training algorithms that efficiently support small batch sizes (e.g. 2, 4) have been proposed [4, 8, 9], which demonstrate on-par accuracy with state-of-the-art CNN training using large batch sizes.

Low-batch training greatly reduces memory requirement and unlocks opportunities for FPGAs, which provide higher configurability for custom architecture and better energy-efficiency than high-power GPUs. Training on edge FPGA devices also reduces latency overhead (due to limited data exchange with the cloud server), prevents privacy/security problems, and is well-suited to exploit new features such as low precision training, sparse weight updates, online learning, etc. However, training CNNs on FPGAs is a challenging task for two reasons: (1) it demands high memory bandwidth which is the primary limiting factor in many accelerators [10, 11], and (2) complexity arises in implementing a generalized flexible training accelerator supporting new advancements in CNN training algorithms.

Many prior works presented low-batch CNN inference on FPGAs and showed large improvements in storage and latency [12–18].

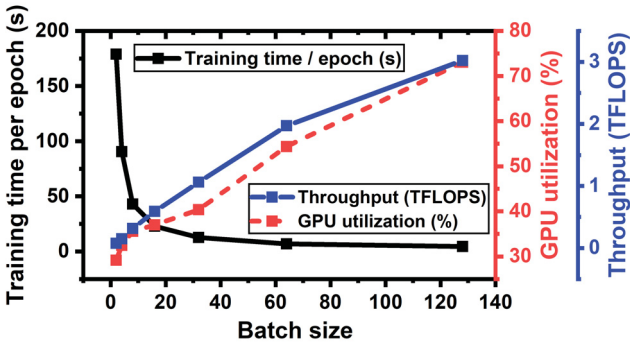

Figure 1: ResNet-20 CNN training performance for CIFAR-10 dataset on Tesla V100 GPU for different batch sizes.

While there are new algorithmic approaches to support low-batch training, FPGA hardware designs for CNN training tasks have been much less explored. A framework to map DNN training on FPGA clusters was presented in [19], but an excessive amount of on-chip memory is required for training on a single FPGA platform. Several prior works [20–22] attempted to accelerate a part of CNN training on FPGAs, while the remaining operations were performed by the host CPU. Training accelerators for non-CNN applications were proposed in [23–27]. Only a few prior works presented a CNN training accelerator supporting all three phases of training [28–30], but these works still did not include back-propagation of either residual connections or stride-2 convolutions that are necessary for modern CNNs. Furthermore, none of the aforementioned FPGA works studied the use of high bandwidth memory, which is critical for CNN training.

In this work, we present a programmable FPGA accelerator for CNN training, which uses HBM2 for efficient off-chip communication, and supports residual connections and stride-2 convolutions for modern CNNs. The key contributions of this work are:

- To the best of our knowledge, we present the first FPGA accelerator for CNN training that fully utilizes high bandwidth memory (HBM2) and executes end-to-end CNN training.
- Our programmable FPGA accelerator reads high-level descriptions of CNNs (similar to TensorFlow/PyTorch) including those with residual connections and stride-2 convolutions, and automatically generates RTL for synthesis.
- We analyze the impact of HBM2 on CNN training workloads, provide a comprehensive comparison with DDR3, and discuss the strategies to efficiently use the HBM2 features for enhanced performance.
- Our accelerator using Intel Stratix-10 (S-10) MX FPGA with HBM2 is evaluated for ResNet-20 and VGG-like CNNs for CIFAR-10 dataset, achieving up to 14X improvement in energy-efficiency, compared to Tesla V100 GPU.

The remainder of this paper is organized as follows. Section 2 introduces HBM2 memory on Intel FPGAs and modern CNN training algorithm. Section 3 presents the proposed training accelerator including HBM integration, handling residual and stride-2 convolutions. Section 4 describes experimental results and provides comparison among FPGA, CPU, and GPU hardware for low-batch CNN training tasks. The paper is concluded in Section 5.

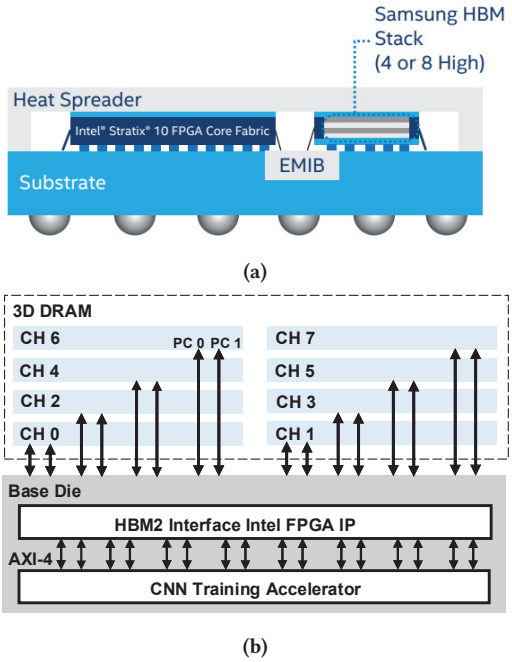

Figure 2: (a) Intel S-10 MX device integrated with HBM2 [11]. (b) Eight independent physical channels (CH) and corresponding pseudo channels (PC) of HBM2 connected to CNN training accelerator on the base die using Intel HBM2 interface FPGA IP.

2 BACKGROUND

Modern FPGAs, such as Intel Stratix-10 (S-10) MX [11], are equipped with new high-speed memory technologies like high bandwidth memory (HBM2) [31]. HBM2 uses 3D stacked silicon dies connected through through-silicon vias (TSVs). The main DRAM stack is placed as a top die and the base die is used for I/O connections to the host device. Each die in the DRAM stack consists of two independent physical channels, which are further divided into two pseudo channels. As shown in Fig. 2(a), HBM2 is integrated with the Intel S-10 MX device using the system-in-package (SiP) technology. Fig. 2(b) depicts the interface between the DRAM stack and the base die. All the physical channels (CH) and corresponding pseudo channels (PC) are connected to the base die using HBM2 interface Intel FPGA IP. Dedicated customizable memory controllers are provided for each physical channel of HBM2. Overall, HBM2 provides higher bandwidth, I/O and capacity with a small form factor, compared to traditional off-chip memories such as DDR3.

However, designing an architecture that can fully leverage high memory parallelism provided by HBM2 is challenging. Large I/O capacity of HBM2 demands unique data storage (a number of different parameters can be read in single access) and complex on-chip buffer control logic to handle the incoming data from HBM2. Accessing parallel and independent HBM2 channels requires separate memory controllers and status monitoring for all channels.

2.1 Modern CNN training

CNNs are majorly trained with SGD optimizer using backpropagation algorithm, which is an iterative process used to find the best

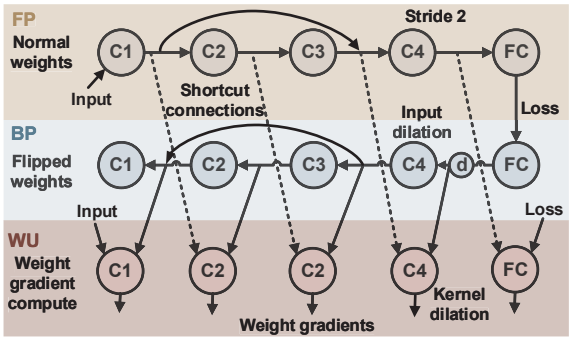

Figure 3: Training dataflow for CNNs involving stride-2 convolutions ($C4$) and shortcut connections. Ci is i^{th} convolution layer, d is the input pixel dilation, and FC is the fully-connected layer.

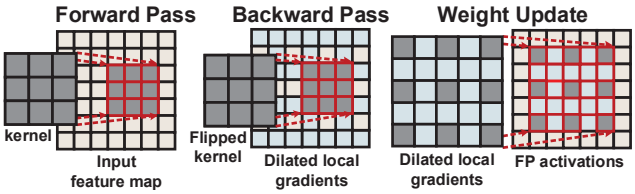

Figure 4: Stride-2 convolutions in different training phases are shown. Blue cells represent dilated positions of the activations (beige) or weights (grey).

parameters of a network that minimizes the loss function. SGD-based training involves three phases, namely forward pass (FP), backward pass (BP) and weight update (WU). In the FP phase, the output activations are computed layer by layer in the forward direction and the FP performance is estimated using a loss function. In the BP phase, the local gradients are computed at every layer in the backward direction. During BP, convolution operations use flipped kernels and the pooling (downsampling) operations are replaced by upsampling units. In the WU phase, weight gradients are computed using the local gradients and feed-forward activations, and weight updates are computed.

Modern CNNs involve residual connections [6, 32], and multi-stride convolutions to downsample the data and improve the storage/throughput of CNN training. Fig. 3 illustrates the overall training flow of a CNN with identity residual connections and convolutions with stride of 2. Identity shortcut operations remain the same during FP and BP, but the flow direction and the accumulation node are changed. During FP, we accumulate the shortcut connection at the output of convolution layer C3, but during BP, we accumulate the output of C2 (Fig. 3). For convolutions with stride larger than 1, local gradients are computed by performing the convolution of the horizontally and vertically dilated gradients with flipped kernels. In the WU phase, the weight gradients are computed by convolving the FP activations with dilated BP local gradients, which is similar to dilating the kernels during the convolution. Fig. 4 shows different dilations used in stride-2 convolutions.

In this work, we benchmark the training tasks of ResNet-20 CNN [6] and VGG-like CNN [28] for CIFAR-10 dataset, using the proposed FPGA accelerator. ResNet-20 CNN has a convolution layer, followed by three stacks of 6 convolution layers, 9 residual

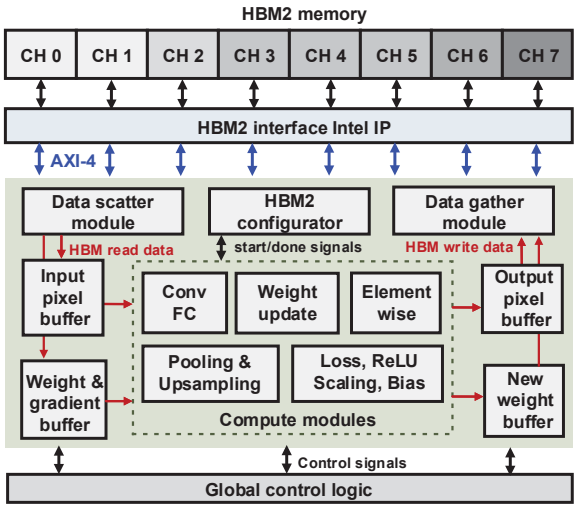

Figure 5: Top-level block diagram of the proposed CNN training hardware accelerator using HBM2 memory.

identity connections, a pooling layer, and a fully-connected layer. The feature maps are downsampled at the output of every stack using stride-2 convolutions. VGG-like CNN has 6 convolution layers (C), 3 max-pooling layers (MP) and a fully-connected layer (FC), with the structure of 16C3-16C3-MP-32C3-32C3-MP-64C3-64C3-MP-FC.

3 CNN TRAINING ACCELERATOR DESIGN

3.1 Overall architecture and operation

Fig. 5 shows the top-level block diagram of the CNN training accelerator architecture. The architecture can be mainly divided into five blocks:

(1) Compute block supports various operations required for FP, BP and WU phases of training.
(2) Buffer block stores input, output, weight and gradient data in on-chip buffers.
(3) HBM2 configurator block generates signals to access HBM2, stores it to an on-chip buffer and write the data back to HBM2.
(4) HBM2 memory stores all CNN training parameters, and HBM2 interface Intel IP communicates HBM2 memory and training accelerator.
(5) Global control logic governs all the modules and performs layer scheduling.

The compute block consists of a systolic MAC array to support convolution and fully-connected layers. A 8x8x16 MAC array is used to compute 8x8 pixels of 16 output feature maps in parallel. MAC array size is chosen to exhibit a high utilization ratio. A MAC array of higher size, for example 32x32x16, will suffer under utilization while computing convolution of deeper layers where the output feature map size is small. Flexibility to choose the MAC array size also helps map the algorithm on different sized FPGAs. The MAC array is used to support fully-connected layers, normal convolutions during FP, transposed convolution during BP, and intra-tile accumulation in WU phases. A data router module is

tightly coupled with the MAC array and distributes the parameters to the MAC array considering the padding and stride values.

Before the computed data is sent to the output buffer, it goes through a series of secondary layers including loss function, ReLU, bias and scaling unit. The scaling unit is used during BP where the derivative of a node is either 1 or 0 (e.g. ReLU, dropout layer). The secondary layers use outputs of key layers without any HBM access. In each layer, any of these secondary operations can be enabled or disabled based on the CNN structure.

Element-wise (Eltwise) module performs the element-wise addition of two input layers supporting identity shortcut connections required for ResNet CNNs [6]. If the volume of the input layers is different, then the smaller input layer is padded with zeros. Eltwise module is enabled once all the output data of the current layer are computed. Data from the other branch of the identity shortcut connection is read from the HBM2 to the input buffers. Finally, the accumulated data is written back to output buffers.

The pooling module is used during FP, and downsamples the input feature map by taking the maximum value within a kernel window (e.g. 2x2). During BP, the gradients will only flow through the selected pixel positions and non-selected pixel positions are padded with zeros. This operation is carried out by the upsampling unit. The compute array sizes of Eltwise, pooling, and upsampling modules are configurable.

The weight update module performs weight gradient accumulations, following the SGD algorithm. The accumulation of weight gradients is carried out for all training images in a batch. New weights are computed using the final weight gradient value and is written back to HBM2. Data scatter/gather module rearranges the data for HBM communication.

The global control logic governs the layer scheduling, enabling the secondary operations and configures the modules as required for the given network. The global control logic reads the detailed CNN structure through configuration registers. An RTL compiler is developed to generate these configurations, where the CNN structure, MAC array sizes and other control parameters are inputs to the compiler framework. The compiler framework reads the high-level inputs and translates the layer by layer execution schedule as parameters for the configuration registers, which is read by the global control logic in run-time. The RTL compiler consists of a highly parameterized hand-written RTL library which is optimized for CNN training. The overall accelerator consisting of configurable modules is shown in Fig. 5. The RTL compiler only compiles the required modules based on each CNN structure, without including any unused modules.

CNN training involves various parameters such as activations, weights, weight gradients, local gradients, momentum gradients, etc. The parameters required for a given layer is read from the HBM2 and stored in on-chip buffers (Fig. 5). Input/output pixel buffers are used to store the inputs/outputs of the compute blocks. Weight buffer is designed to support efficient weight access in both non-transpose and transpose directions (for FP and BP phases, respectively), following the schemes proposed in [28, 33]. Weight gradient buffers are used during the WU phase to read the old gradients and momentum gradients. All the parameters required for the entire CNN training are stored in HBM2 memory.

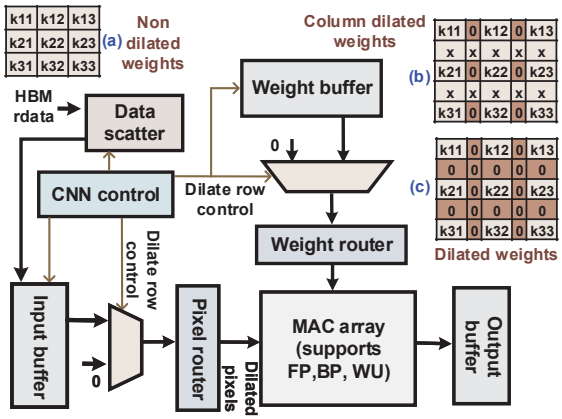

Figure 6: Flexible MAC unit with a dilation control block for both weights and activations. The non-dilated weights/activations from the HBM2 is rearranged by the dilation control block and the data scatter unit.

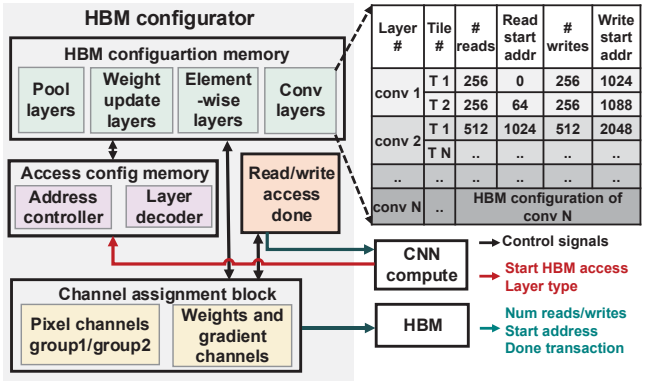

Figure 7: HBM2 configurator module generates HBM2 read and write configuration signals based on the layer type.

3.2 Dilated convolutions

The BP and WU phases of stride-2 convolutions require dilations in both weights and input feature maps, (Fig. 4). Fig. 6 shows the design of control logic to support BP and WU of stride-2 convolutions. The non-dilated data (a) is loaded from HBM2 to on-chip buffers. Storage of dilated images/kernels in HBM2 is avoided to reduce latency. The non-dilated data is rearranged by the scatter unit according to the on-chip buffer storage pattern requirement. During this data rearrangement, the data scatter unit dilates the data (pixels or weights) in the x-dimension (b). Dilations in y-dimension is performed by the address control logic by skipping the writes of every dilated row. While reading the data to the convolution engine, every dilated row is detected and padded with zeros (c). This dataflow is replicated for both weights and input feature maps, and can be configured as needed using global control logic.

3.3 HBM2 configurator module

Fig. 7 shows the HBM2 configurator, which generates HBM2 read/write transaction details. The HBM2 configurator consists of a configuration memory that is preloaded with the information of every

Table 1: HBM channel allocation for training parameters. 16 pseudo channels of the HBM2 is divided into four groups, for which input/output activations (in./out act), local gradients (LG), transposable weights and weight gradients (Wt gradients) are assigned.

Phase	Layer	Activations & local gradients		Weights	Wt gradients
		channel 0-3	channel 4-7	channel 8-11	channel 12-15
FP	C	in./out. act	in./out. act	transposable weights	NA
	P,EW	in./out. act	in./out. act	NA	NA
	FC	NA	NA	transposable weights	NA
BP	C	in./out. LG	in./out. LG	transposable weights	NA
	P,EW	in./out. LG	in./out. LG	NA	NA
	FC	NA	NA	transposable weights	NA
WU	C	in. act	in. LG	old/new weights	old, new moment gradients
	FC	NA	NA		

transaction. The information in the configuration memory includes the number of read/write transactions, and the read/write start addresses. Each layer has its own configuration memory as depicted in Fig. 7. Given the current layer details and tile count, the address controller generates the read address for configuration memory selected by the layer decoder. Once the transaction information is read from the memory, it is assigned to channels in the channel assignment block.

16 pseudo channels of HBM2 provide a high number of I/O data pins. To effectively utilize this parallelism provided by HBM2, proper channel allocation and organized parameter storage become a necessity. For our application, 16 pseudo channels of HBM2 are divided into four groups of four channels. Each CNN training parameter that is stored in HBM2 is assigned with one of the four-channel groups. Table 1 provides the details of channel group allocation and the channel groups used in each phase of training. Channels 0-3 (group 1) and channels 4-7 (group 2) are used to store the local gradients and activations, channels 8-11 (group 3) are used to store the weights, and channels 12-15 (group 4) are used to store the weight gradients (both current weight gradients and momentum gradients). This channel allocation is done to reduce the off-chip latency of the WU phase.

In the WU phase, we need to read both activations and the local gradients to compute the weight gradients. To maximize the channel utilization, the local gradients and activations are stored in the two channel groups in a ping-pong manner. For example, if convolution layer 1 outputs are stored in channel group 2, then its corresponding local gradients are stored in channel group 1, and during the WU phase, we read channel groups 1 and 2 simultaneously. Using this channel allocation, all 16 channels will be active during the WU phase. The channel assignment block assigns the transaction information read from the configuration memory to one of the channel groups based on the request from the CNN compute module. The done logic module monitors transactions of every channel and HBM2 status signals, and generates a 'done' signal when the transaction is complete.

3.4 HBM2 integration

HBM2 communication uses the Intel HBM2 controller (HBMC) following the AMBA AXI-4 protocol. HBMC provides independent

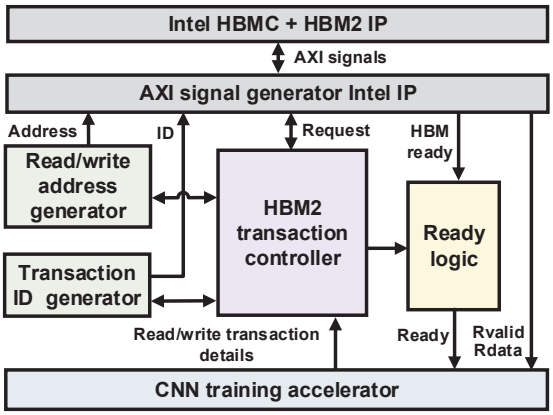

Figure 8: CNN training accelerator integrated with HBM2, following AMBA AXI4 protocol.

AXI ports for each channel. The read/write transaction information obtained from the HBM2 configurator is sent to the HBM transaction controller (HTC). Fig. 8 shows the integration of HTC and other modules with Intel HBM IPs, enabling successful HBM2 communication. HTC monitors the request from the CNN training accelerator and the status of HBM2 memory. Based on the read/write request from the training accelerator, HTC enables corresponding address/transaction ID tag generators. The generated address and transaction IDs are converted to AXI signals using the AXI signal generator. Ready logic monitors the status of HTC, address generators, and HBM2 and indicates whether the HBM is ready to accept the next transaction.

3.5 Data scatter/gather unit

HBM2 demands complex and flexible data collection/gathering units as more data is streamed in one cycle. To achieve this, customized data scatter/gather units were designed which can collect/send the data based on the channel allocation. The storage pattern of the parameters on on-chip buffers depends on the layer and parameter types. The continuous data stream from the HBM2 channels are collected by the data scatter unit where the data is rearranged and distributed to the on-chip buffers. The scatter unit also separates channels based on the parameter channel allocation and processes all channel groups in parallel. The data gather unit collects the data from output buffers (or new weight and weight gradient buffers in WU phase) and reorganizes the data before sending it to HBM2 channels. Data scatter/gather unit considers the channel allocations of different parameters, data precision, unroll factors and layer type.

3.6 HBM2 initialization

The HBM initialization module loads the HBM with training data and other initial parameters. To initialize HBM with the training data, the data is loaded from the host PC to the on-chip memory (M20K) of the FPGA. The on-chip memory (M20K) works as an intermediate buffer for each pseudo-channel. Due to the limited on-chip memory resources, we used small buffers, and these buffers will be used multiple times to load a large amount of data to HBM.

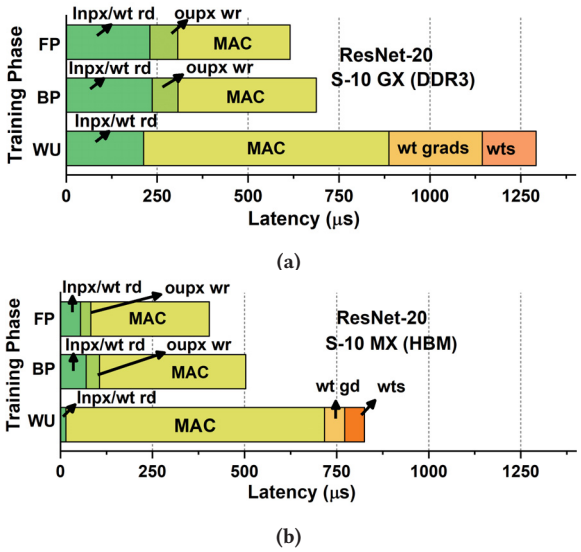

Figure 9: Training latency breakdown of ResNet-20 CNN for (a) S-10 GX device with DDR3 and (b) S-10 MX device with HBM2, both running at 185 MHz.

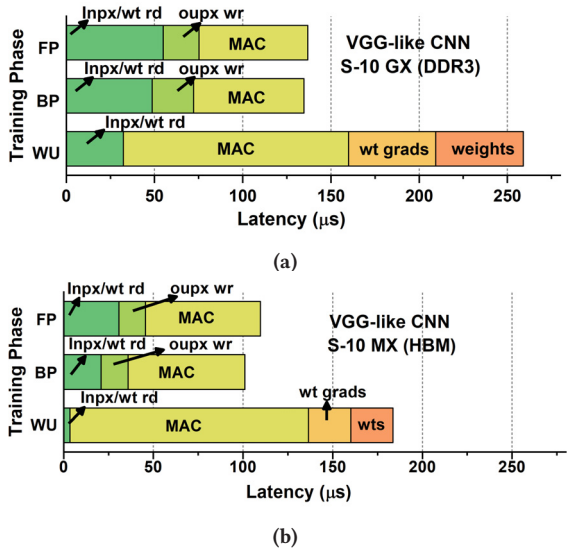

Figure 10: Training latency breakdown of VGG-like CNN for (a) S-10 GX device with DDR3 and (b) S-10 MX device with HBM2, both running at 185 MHz.

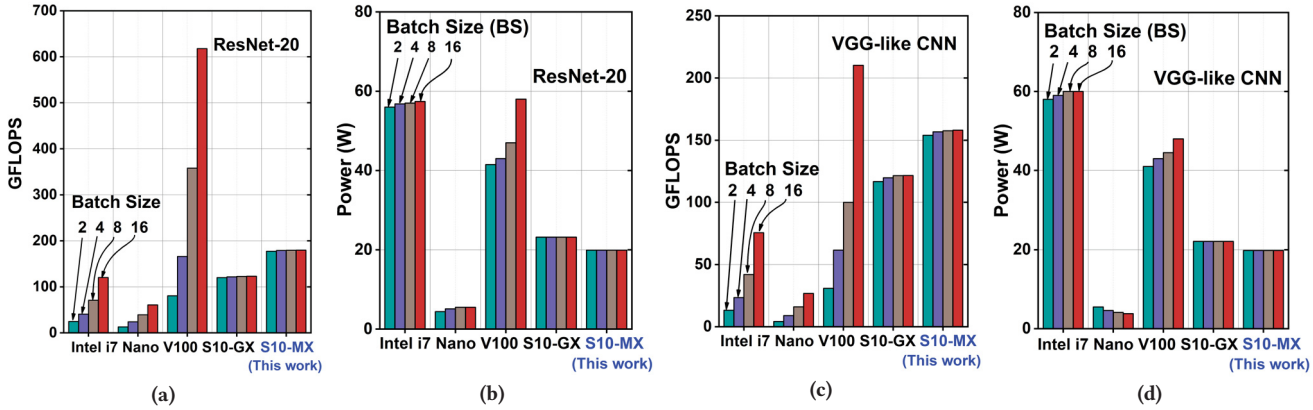

Figure 11: Throughput and power comparison of training tasks using Intel i7-9800X CPU, Tesla V100 GPU, Jetson Nano, S-10 GX FPGA with DDR3, and S-10 MX FPGA with HBM2. (a) Throughput and (b) power for ResNet-20 CNN training and (c) throughput and (d) power of VGG-like CNN training are shown.

The HBM configurator and HBM control modules of the CNN training accelerator is reused to perform the HBM initialization. After loading all required training data, the CNN training accelerator is enabled. The HBM initialization is controlled by the host system.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We evaluate our FPGA accelerator on two CNNs (ResNet-20 and VGG-like CNN) with CIFAR-10 as the training dataset. The control logic is also configurable to support large dataset (ImageNet) training, but consumes more FPGA resources to store and process larger input images. The initial weight parameters and configuration register values of benchmark CNNs are generated from our RTL compiler framework developed in Matlab. Intel S-10 MX (1SM21BHU2F53E2VGS1) [34] and S-10 GX(1SG280LU3F50E3VGS1) [35] were used as the target FPGA hardware. S-10 MX is equipped with 133 Mbits of M20K, 3,960 DSP blocks, 702K ALMs and 8GB HBM2 memory providing peak memory bandwidth of up to 512 GBps and S-10 GX includes 5,760 DSP blocks, 933K ALMs and 240 Mbits of M20K and 4GB DDR3 with 16.9GB/s bandwidth. Identical design optimizations has been performed on both S-10 MX and S-10 GX design for fair comparison.

All parameters use 16-bit floating-point precision to reduce the memory footprint compared to 32-bit floating-point precision. Since the DSP units of Intel S-10 GX/MX FPGAs only support 32-bit floating-point precision, 16-bit (32-bit) to 32-bit (16-bit) floating-point precision converters are used before (after) DSP computation to utilize the existing DSP blocks in S-10 FPGAs. The latency was

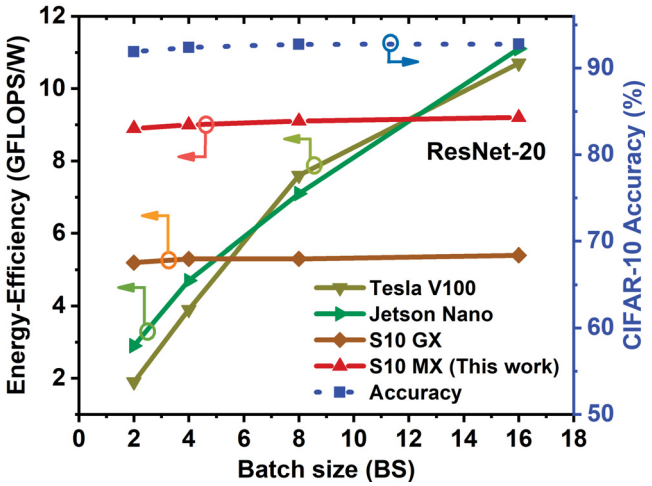

Figure 12: Energy and accuracy comparison of low-batch CNN training on Tesla V100 GPU, Jetson Nano, S-10 MX and GX devices.

measured using the functional simulation of the training accelerator. Using Intel Quartus 19.4 FPGA software, the accelerator was synthesized, placed/routed and the bitstream was uploaded to the FPGA board. Intel(R) Core (TM) i7-9800X CPU is used as the host system. For comprehensive comparison among FPGA, CPU, and GPU hardware for the same training tasks, we measured the power of each actual hardware system. FPGA board power consumption is measured using the Intel board test system (BTS) power monitor. The Intel BTS power monitor reported junction temperature of 47°C. Intel(R) Core (TM) i7-9800X CPU power is measured with the powerstat command using RAPL domains. To evaluate the performance of GPUs, we developed a floating point CIFAR-10 training model using PyTorch [36]. Tesla V100 GPU power measurements are done using CUDA nvidia-smi API and Jetson Nano power measurements are done using Nvidia tegrastat utility. To minimize measurement inaccuracy, 20 samples of power measurements are averaged over the duration of one epoch training.

4.2 Results and analysis

Table 2 shows the resource utilization of two CNN benchmarks (ResNet-20 and VGG-like CNN) for the CIFAR-10 dataset. All the training images in a given batch are processed sequentially. This greatly reduces the on-chip memory requirements as we only read the data required to process one training image at a time from HBM2. We achieved maximum operation frequency of 185 MHz for S-10 MX and GX implementations.

Table 2: Resource utilization for training tasks of ResNet-20 and VGG-like CNNs on Intel S-10 MX (1SM21BHU2F53E2VGS1) and S-10 GX(1SG280LU3F50E3VGS1) FPGA.

CNN	FPGA	DSP	ALM	M20Ks	Registers	Freq.
ResNet-20	S10-MX	1040 (26%)	239k (34%)	2558 (13.9M)	390k	185 MHz
VGG-like	S10-MX	1046 (26%)	221k (31%)	2998 (11.4M)	353k	185 MHz
ResNet-20	S10-GX	1043 (18%)	148k (16%)	1779 (14M)	385k	185 MHz
VGG-like	S10-GX	1044 (18%)	97k (10.4%)	1297 (11M)	167k	185 MHz

For S-10 GX (with DDR3) implementation, Fig. 9(a) and Fig. 10(a) show the latency breakdown of the proposed accelerator for three training phases (FP, BP, and WU) of the last training image of a batch (involving actual weight updates) for ResNet-20 CNN and VGG-like CNN, respectively. The latency breakdown includes the reading of input pixels and weights from off-chip memory (Inpx/wt rd), computing the convolution outputs (MAC), writing the output pixels (oupx wr), wt gradients (wt grads) and new weights (wts) back to HBM. In the overall training time, the off-chip DDR3 memory consumes 47% of latency and logic consumes 53%. For memory-bound CNNs, even with high hardware parallelism, the low bandwidth of DDR3 memory will limit the performance [28]. This critical memory bandwidth bottleneck can be addressed using HBM2.

Fig. 9(b) and Fig. 10(b) provide the latency breakdown of the proposed FPGA accelerator implemented on the S-10 MX device using HBM2. WU phase consumes longer latency than FP/BP phases, as it involves weight gradient computation, gradient accumulation and computation of new weights. The high off-chip memory bandwidth provided by HBM2 significantly reduces the latency consumed to read/write the activations and weights from/to the off-chip memory. As a result, the logic latency dominates the total latency, compared to S-10 GX implementation with DDR3 in all three phases of training. Further latency improvement could be achieved by increasing the number of parallel MAC arrays or by increasing the operating frequency. Using the proposed channel allocation scheme and HBM2 for the S-10 MX implementation, we achieved ~4X reduction in off-chip memory latency and ~1.5X reduction in system-level CNN training time, compared to those of the S-10 GX implementation with DDR3.

Fig. 11a and Fig. 11c provide the low-batch training throughput of two CNN benchmarks (ResNet-20 and VGG-like CNN) on Intel i7-9800X CPU, Jetson Nano embedded platform, Tesla V100 GPU, S-10 GX FPGA and S-10 MX FPGA. The overall training time of GPUs significantly increases with lower batch sizes. The proposed FPGA training accelerator achieves better throughput compared all other hardware platforms on both the benchmarks for small batch sizes of 2 and 4. Tesla V100 provides better throughput for higher batch sizes (8 and 16) but at the cost of high power consumption. The power consumption of all hardware platforms for different batch sizes are shown in Fig. 11b and Fig. 11d. The FPGA power consumption is low because of less utilization (~30%) of FPGA resources, operating frequency of 185MHz and junction temperature of 47°C reported by Intel BTS tool. Compared to S-10 MX FPGA, Jetson Nano consumes less power (~5W) but suffers from long training latency. For ResNet-20/VGG-like CNNs, our FPGA implementation of CNN training using S-10 MX with HBM2 is ~4.5-9.7X more energy-efficient compared to Tesla V100 GPU, ~3-7X more energy-efficient compared to low-power Jetson Nano embedded platform, and ~1.7X more energy-efficient compared to implementation on S-10 GX with DDR3. Our proposed S-10 MX design with HBM2 addresses the critical memory bottleneck problem for CNN

training and the custom architecture enables efficient low-batch training.

Fig. 12 shows the overall energy-efficiency and accuracy comparison of Tesla V100 GPU, Jetson Nano, S-10 MX, S-10 GX devices for ResNet-20 training across different batch sizes. It can be seen that the low-batch training accuracy has minimal degradation compared to high-batch training accuracy [8]. At the same frequency and MAC array size, S-10 MX design provides 1.7X improvement in energy-efficiency compared to S-10 GX design by greatly reducing the off-chip communication latency.

5 CONCLUSION

This paper presents a flexible CNN training accelerator on FPGA using HBM2, which performs end-to-end training of modern CNNs involving residual connections and stride-2 convolutions. The FPGA accelerator is implemented on Intel S-10 MX (with HBM2) and S-10 GX (with DDR3) devices, demonstrating system-level benefits of HBM2 over conventional DDR3 off-chip memory. The proposed accelerator achieves 4.5-9.7X energy-efficiency improvement compared to Tesla V100 GPU and 7-11X improvement in throughput compared to that of Intel i7-9800X CPU for low-batch training tasks of ResNet-20/VGG-like CNNs.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[2] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[3] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6243–6252, 2017.

[4] Yuxin Wu and Kaiming He. Group normalization. In *European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[5] Pavan Kumar Chundi, Peiye Liu, Sangsu Park, Seho Lee, and Mingoo Seok. FPGA-based Acceleration of Binary Neural Network Training with Minimized Off-Chip Memory Access. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[7] A Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront*, 2009.

[8] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2019.

[9] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In *Advances in neural information processing systems*, pages 1945–1953, 2017.

[10] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. *Xilinx Whitepaper*, 2017.

[11] Manish Deo, Jeffrey Schulz, and Lance Brown. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge. *Intel Whitepaper*, 2017.

[12] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jaesun Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017.

[13] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[14] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *IEEE/ACM Design Automation Conference (DAC)*, pages 1–6, 2017.

[15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[16] 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 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 26–35, 2016.

[17] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating cnn inference on fpgas: A survey. *arXiv preprint arXiv:1806.01683*, 2018.

[18] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of fpga-based neural network accelerator. *arXiv preprint arXiv:1712.08934*, 2017.

[19] Tong Geng, Tianqi Wang, Ang Li, Xi Jin, and Martin Herbordt. A Scalable Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Weight and Workload Balancing. *arXiv preprint arXiv:1901.01007*, 2019.

[20] Afzal Ahmad and Muhammad Adeel Pasha. Optimizing hardware accelerated general matrix-matrix multiplication for cnns on fpgas. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[21] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-CNN: An FPGA-based framework for training convolutional neural networks. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, 2016.

[22] Seungkyu Choi, Jaehyeong Sim, Myeonggu Kang, and Lee-Sup Kim. TrainWare: A memory optimized weight update architecture for on-device convolutional neural network training. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2018.

[23] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 255–265, 2020.

[24] Qiang Liu, Jia Liu, Ruoyu Sang, Jiajun Li, Tao Zhang, and Qijun Zhang. Fast neural network training on FPGA using quasi-newton optimization method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(8):1575–1579, 2018.

[25] Alexander Gomperts, Abhisek Ukil, and Franz Zurfluh. Development and implementation of parameterized FPGA-based general purpose neural networks for online applications. *IEEE Transactions on Industrial Informatics*, 7(1):78–89, 2011.

[26] Rafael Gadea Gironés, Rafael Gadea Gironés, Ricardo Colom Palero, Joaquín Cerdá Boluda, Joaquín Cerdá Boluda, and Angel Sebastia Cortés. Fpga implementation of a pipelined on-line backpropagation. *J. VLSI Signal Process. Syst.*, 40(2):189–213, June 2005.

[27] Shijie Zhou, Rajgopal Kannan, and Viktor K Prasanna. Accelerating Stochastic Gradient Descent Based Matrix Factorization on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1897 – 1911, 2020.

[28] Shreyas Kolala Venkataramanaiah, Yufei Ma, Shihui Yin, Eriko Nurvithadhi, Aravind Dasu, Yu Cao, and Jaesun Seo. Automatic Compiler Based FPGA Accelerator for CNN Training. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 166–172, 2019.

[29] Hiroki Nakahara, Youki Sada, Masayuki Shimoda, Kouki Sayama, Akira Jinguji, and Shimpei Sato. FPGA-Based Training Accelerator Utilizing Sparseness of Convolutional Neural Network. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 180–186, 2019.

[30] Cheng Luo, Man-Kit Sit, Hongxiang Fan, Shuanglong Liu, Wayne Luk, and Ce Guo. Towards efficient deep neural network training by fpga-based batch-level parallelism. *Journal of Semiconductors*, 41(2):022403, 2020.

[31] JEDEC Standard. High bandwidth memory (HBM) DRAM. *JESD235*, 2013.

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[33] Shihui Yin and Jaesun Seo. A 2.6 TOPS/W 16-bit Fixed-Point Convolutional Neural Network Learning Processor in 65nm CMOS. *IEEE Solid-State Circuits Letters*, 3:13–16, 2020.

[34] Intel. Intel Stratix 10 MX FPGA Development Kit. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html. Accessed: 2020-04-02.

[35] Intel. Intel Stratix 10 GX FPGA Development Kit. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-fpga.html. Accessed: 2020-04-02.

[36] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop*, 2017.