FPGA-Based Training of Convolutional Neural Networks With a Reduced Precision Floating-Point Library

Roberto DiCecco, Lin Sun and Paul Chow
University of Toronto, Department of Electrical and Computer Engineering, Ontario, Canada
E-mail: dicecco1@eecg.toronto.edu, zac.sun@mail.utoronto.ca, pc@eecg.toronto.edu

***Abstract*—Convolutional Neural Networks (CNNs) have been shown to have high accuracy for classification tasks in numerous applications, which has resulted in their widespread adoption. However, the high accuracy of CNNs comes at the cost of high compute and bandwidth requirements for both classification and training. In this work we discuss an FPGA-based CNN training engine: FCTE, implemented using High-Level Synthesis (HLS), targeting the Xilinx Kintex Ultrascale XCKU115 device. Furthermore, we detail custom-precision floating-point (CPFP) cores for multiplication and addition implemented using HLS, which allows for reduced area utilization. We use these cores with our engine to train networks to demonstrate that an exponent width of 6 and mantissa width of 5 achieves accuracy comparable to single-precision floating-point for the MNIST and CIFAR-10 datasets. These results are achieved using round-to-zero for the CPFP multipliers and round-to-nearest for the CPFP adders, allowing for LUT savings of 32.6% for the multipliers and 21.7% for the adders when compared to half-precision floating-point, while using the same number of DSPs.**

I. Introduction

Convolutional Neural Networks (CNNs) have seen a significant rise in their usage. Recent FPGA works have mostly focused on the inference portion of CNNs, typically through model conversion from CNN frameworks to FPGA-based model-specific accelerators [1], [2]. While this allows for high-performance inference, it is less favorable for training as it requires running synthesis and place and route whenever a model is changed, which can take on the order of hours. Therefore, for training to be practical on FPGAs, there is a need for a more general architecture that can be used as model parameters are changed.

A primary consideration of a general architecture for training CNNs is the data type that should be used. Recently, there has been a push to use half-precision floating-point (fp16) for inference and training using GPUs [3]. However, prior work has shown that CNNs can be trained with fixed-point representations [4], [5]. The use of fixed-point and fp16 demonstrate the resilience of CNNs to low precision data types, which is beneficial for FPGAs as it can result in less area intensive designs. However, during training, fixed-point requires more fine-tuning than floating-point due to its lack of dynamic range. This can be alleviated by using dynamic fixed-point [5], or training with lower precision followed by high-precision [4], but both cases are more complex than floating-point for model exploration. With this taken into consideration, in this work we make the following contributions:

- We detail High-Level Synthesis (HLS) implementations of parameterizable custom-precision floating-point (CPFP) multiplication and addition.
- We present a FPGA CNN training engine (FCTE), with support integrated into FPGA Caffe [6]. FCTE is implemented using our CPFP cores with Xilinx SDAccel, targeting a Xilinx Kintex Ultrascale XCKU115 device.
- We demonstrate that we can reduce the exponent and mantissa widths to 6 and 5 respectively while achieving accuracy within 0.18% and 0.029% of single-precision floating-point (fp32) for the MNIST and CIFAR-10 datasets respectively. These results are achieved using round-to-zero (RTZ) for the multiplier and round-to-nearest (RTN) for the adder. To the best of our knowledge, this is the first work exploring the use of reduced-precision floating-point for training CNNs.
- Finally, all implementation details and libraries are available at: https://github.com/dicecco1/fpga_caffe and https://github.com/dicecco1/fpga_cpfp.

II. Background

CNNs are a type of feedforward network represented as a directed acylic graph where each node, or layer, consists of a set of computations applied to the input data [7]. The most common layers of CNNs are: convolution, pooling (max or average), activations (ReLU, Sigmoid, etc.), and fully connected (FC). Each layer has two sets of operations: forward and backward propagation. Forward propagation involves passing a set of input feature maps through each layer to produce the output of the network. In the case of inference the output may be used to classify an input that has not been seen by the model, while for training the output is used to compute the classification error. Backward propagation involves propagating the classification error through the network using the chain rule to update the model parameters and minimize a loss (objective) function.

Forward propagation of convolution layers involves convolving a set of input feature maps with O learned filters to produce O output feature maps. The backward pass of convolution involves three gradient computations: gradient of the loss function with respect to the input, filters, and bias.

The gradient with respect to the input and with respect to the filters both require convolution operations, though the inputs and input sizes differ from the forward pass. The gradient with respect to the input is a convolution between the backpropagated gradient of the loss from the previous layer and rotated filters. The gradient with respect to the filters is a convolution between the input and the gradient of the loss from the previous layer.

III. Related Work

Many FPGA works have focused on architectures for CNNs, with most focusing entirely on inference, using data types consisting of: fp16, fp32, and fixed-point [1], [2], [6]. Each of these works, use either Intel or Xilinx OpenCL solutions, with [2] achieving comparable performance (1.382 TFLOPs) to GPU implementations of AlexNet [8].

Several works have focused on reduced precision training of CNNs, with [4] using fixed-point with stochastic rounding, while [5] uses dynamic fixed-point. The work in [4] shows that 16-bit fixed-point can achieve accuracy that is degraded by 0.06% and 0.8% compared to fp32 for the MNIST and CIFAR-10 datasets. Similarly, the work in [5] shows that using low-precision multipliers with high-precision accumulators and dynamic fixed-point, they can achieve comparable accuracy to fp32. With dynamic fixed-point of 10 bits for propagations and 12 bits for weight updates, they achieve accuracy that is degraded by 0.08% and 0.77% of fp32 for the MNIST and CIFAR-10 datasets.

IV. High-Level Synthesis Custom-Precision Floating-Point

In the Vivado HLS flow, operators are mapped to Xilinx IP cores [9]. Fp16 and fp32 are supported in Vivado HLS through the Xilinx floating-point IP core [10]. However, in the case of applications where rounding does not have to be as strict or where standard floating-point precision is not required, these operators allow for limited flexibility. As discussed in Section III, prior works have shown that CNNs do not need high precision to obtain high accuracy during training, meaning that we can compromise on rounding and precision to improve area utilization.

To take advantage of these potential area savings we have implemented CPFP cores for: addition, multiplication, and several auxiliary operators, each written entirely in HLS such that they can be easily imported into SDAccel or HLS designs. These cores are capable of using custom exponent and mantissa widths and two rounding modes: RTZ or RTN. Currently the focus has been on low precision floating-point, with the range of mantissa fixed from 2 to 14 bits. In all cases, denormal numbers are treated as zero to reduce area and latency. In the case of overflows we clip the result at the largest possible value rather than setting it to Infinity. This ensures that invalid results cannot be input to the module or output, removing the need for all NaN and Infinity signalling logic.

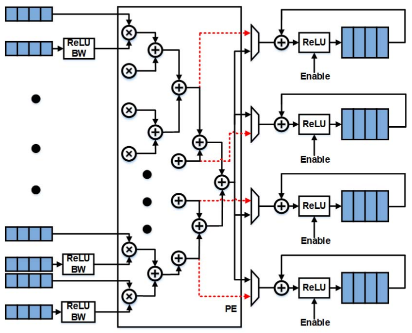

Fig. 1: PE structure, forward path in dashed red

CPFP multiplication requires an E-bit adder (with logic to handle the bias), where E is the exponent width, an (M+1)-bit multiplier, where M is the mantissa width, normalization logic, and rounding logic. The architecture we use is based on the floating-point multiplier described in [11]. With respect to FPGA resource utilization, the (M+1)-bit multiplier uses a single DSP, while the E-bit adder uses LUTs as the exponent is small enough to use the internal carry chains without timing penalties. CPFP addition is implemented as a dual path floating-point adder based on the adder described in [11]. The two paths are the close and far paths, where the close path corresponds to when the difference between exponents is zero or one and the operation is subtraction, and the far path is all other cases. The close path can result in trailing zeros, requiring both a leading one detector (LOD) and a barrel-shifter to renormalize the output. To allow for CPFP addition with low LUT utilization, each LOD has been customized for a given mantissa width.

V. FPGA CNN Training Engine

Many architectures in recent works have a similar structure for their processing elements (PEs), consisting of a set of multipliers followed by an adder tree to implement a dot-product. The size of the dot-product is a function of two unroll factors, F_{fact} and C_{fact}, which correspond to how much of the convolution window and how many of the input feature maps are computed per cycle. F_{fact} larger than one may lead to under-utilization of the PE since the filter size is dependent on the layer. Given that the number of input feature maps is typically divisible by powers of two [8], [12], unrolling by C_{fact} leads to higher PE utilization. The PEs can then be replicated to increase throughput.

In the case of the backward pass, since the operation is also a convolution as discussed in Section II, a similar structure can be employed. The derivative with respect to the input is essentially the same as the forward pass therefore requiring no modifications. The derivative with respect to the weights requires a reduction across the height and width of the input and across the batch of images rather than across the convolution window and the input feature maps. Therefore, if an architecture for the forward pass is replicated such that it processes $H_{fact} \cdot W_{fact} \cdot N_{fact}$ inputs per cycle, where H_{fact},

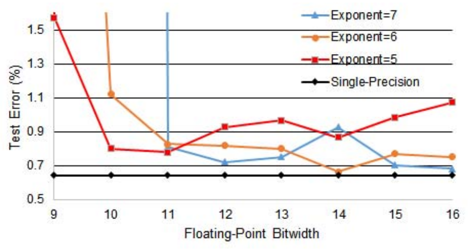

Fig. 2: MNIST Test error rate

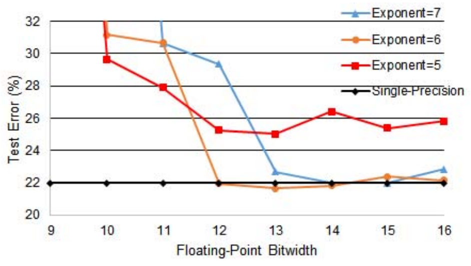

Fig. 3: CIFAR-10 Test error rate

W_{fact}, and N_{fact} are partial unroll factors for the height, width, and batch size; the backward pass requires the same number of multipliers, with an additional $H_{fact} \cdot W_{fact} \cdot N_{fact}$-to-1 adder tree.

To address these needs, we use a PE structure shown in Fig. 1. Each PE contains 16 multipliers and one 16-to-1 adder tree. The PEs are grouped into groups of four to allow for resource sharing between the forward and backward passes. This PE structure corresponds to setting N_{fact} to 16, while the number of PE groups corresponds to setting C_{fact} to four. The forward pass uses the first two stages of the adder tree as shown in dashed red in Fig. 1, while the derivative with respect to the weights uses the full adder tree. To achieve higher overall throughput, we replicate this structure across output feature maps, which requires a broadcast of the inputs to each PE group, while each PE group will have independent weights.

VI. Results

The following subsections detail the accuracy, resource utilization, and throughput of FCTE. The development board used is the Alpha-Data ADM-PCIE-8K5 card with a Xilinx XCKU115, connected to the host CPU through PCIe Gen 3. The host CPU is an Intel Xeon E5-2650 v4 running at 2.2 GHz. The Xilinx SDAccel version used to generate the results is 2016.3. For CPU-only results we use OpenBLAS [13] with 8 threads enabled. For GPU-based comparisons we use Caffe with cuDNN v6 [14] and an NVIDIA M60 GPU, using one of the two GPUs on board. This is the case as using both requires running separate batches for each GPU which modifies model convergence.

A. Accuracy

To evaluate the effectiveness of using CPFP cores we use FCTE to train models for MNIST and CIFAR-10. In each case we use RTZ for the CPFP multipliers and RTN for the CPFP adders. The CPFP cores are used for computing the forward pass and the gradients of the FC, ReLU, max pooling, and convolution layers. The updates to the weights are applied on the host side by converting the gradient to fp32, applying the update, and then converting back to CPFP. We take this approach for simplicity, as the weight updates have a low computational complexity.

The MNIST dataset [15] has 60,000 training and 10,000 test images, where each image is black and white with a size of 28×28. The model used is similar to LeNet-5 [15] and the model used in [4]: two 5×5 convolutional layers with ReLU activations, each followed by a max pooling layer with stride of 2 and a 2×2 window, with two FC layers followed by a 10-way softmax. The first convolutional layer has 32 output feature maps, the second has 64, and the first FC layer has 512 outputs, and we use a minibatch of 256. Fig. 2 shows the test error, which corresponds to the number of images that are incorrectly classified, for various exponent and mantissa widths with fp32 as a baseline. In this case an exponent width of 5 can achieve comparable accuracy (error rate within 1%) to fp32 with a mantissa width of 4. For exponent widths of 6 and 7, the mantissa width needs to be 3 or more to achieve comparable accuracy. For an exponent width of 4 we were unable to achieve comparable accuracy for any mantissa width.

The CIFAR-10 dataset has 50,000 training and 10,000 test images, where each image is 32×32 with three colour channels [16]. The model used is similar to [4], [8]: 3 5×5 convolution layers with ReLU activations, each followed by a max pooling layer with stride of 2 and a 3×3 window. The first two convolution layers have 32 output feature maps, while the last has 64. At the end of the network there is a FC layer with 10 outputs followed by a softmax layer. Similar to the MNIST model, we use a minibatch of 256. Fig. 3 shows the test error for various exponent and mantissa widths with fp32 as a baseline. Notably, an exponent width of 5 results in a degradation in accuracy compared to fp32. This can be attributed to a lack of denormal support in our CPFP cores, meaning that the lower valued loss gradients were rounded to zero due to low dynamic range. This was even more apparent for exponent widths of 4, where the lowest error rate was 90%. Compared to the MNIST model, higher precision is required: exponent widths of 6 and 7 require a mantissa width of 5 to achieve comparable accuracy to fp32.

B. Custom-Precision Floating-Point Utilization

Fig. 4 and 5 show the LUT utilization of our CPFP cores compared to the Xilinx fp16 and fp32 cores for both supported rounding modes. The fp16 and fp32 results were gathered with Vivado HLS 2017.1 with several configurations of DSP utilization, while the results of our CPFP cores were gathered using Vivado HLS 2016.3 due to a change in the way DSPs are inferred in 2017.1, which resulted in LUTs being utilized for small multipliers. To evaluate resource utilization for each core we run synthesis and place and route on two modules: one that multiplies two values together and one that adds two

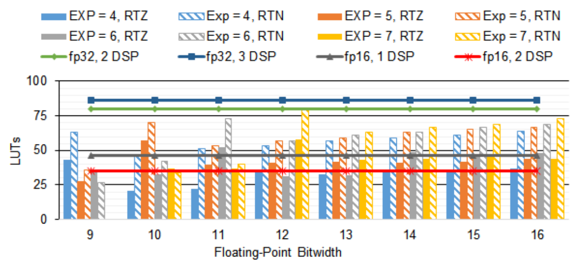

Fig. 4: LUT utilization for the CPFP multiplier with RTZ and RTN compared to fp32 and fp16

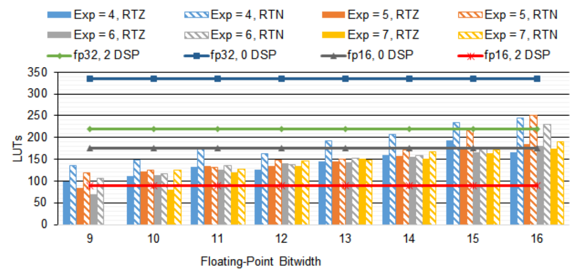

Fig. 5: LUT utilization for the CPFP adder with RTZ and RTN compared to fp32 and fp16

values together. For each run the target clock frequency is 250 MHz. We run two tests for each core: RTN enabled and RTZ enabled.

Fig. 4, shows that with RTN, our core almost always has higher LUT utilization than the equivalent single DSP fp16 core. However, with RTZ our core has comparable LUT utilization to the fp16 cores whether 2 or 1 DSP is in use. An exponent width of 6 and mantissa width of 5 as discussed in Section VI-A results in a LUT utilization improvement of 32.6% compared to fp16 with 1 DSP in use. Fig. 5, shows that our adder core with RTN has lower LUT utilization when the floating-point bitwidth is 12 or less compared to the fp16 core using the same number of DSPs (fp16, 0 DSP). An exponent width of 6 and mantissa width of 5, results in a LUT utilization improvement of 21.7%. With RTZ the LUT utilization is lower than the fp16 core for floating-point bitwidths below 14.

C. Throughput

To evaluate throughput we consider FCTE using CPFP cores with an exponent width of 6 and mantissa width of 5. The number of PEs was scaled until the clock period began to significantly degrade, corresponding to 48 PEs running at 222 MHz with LUT utilization of 36.6%, DSP utilization of 16%, BRAM utilization of 28% and FF utilization of 23.4%. The forward-backward throughput for the MNIST dataset was 926 images per second, while for the CPU it was 1,209, and for the GPU it was 16,753. For the CIFAR-10 dataset, the forward-backward throughput was 522 images per second, while for the CPU it was 411, and for the GPU it was 10,640. This shows that while the GPU has significantly higher throughput overall, the CPU and FPGA have comparable throughput, with the FPGA beating the CPU for CIFAR-10 and the CPU beating the FPGA for MNIST.

VII. Conclusion

In this paper we discuss the use of CPFP addition and multiplication for training CNNs. We introduce a CNN training engine with comparable throughput to CPUs. This engine is used to show that an exponent width of 6 and mantissa width of 5 is sufficient to achieve comparable accuracy to fp32 using models targeting the MNIST and CIFAR-10 datasets. We show that with these reductions in precision we can reduce our multiplier and adder LUT utilization by 32.6% and 21.7% respectively compared to fp16 with the same number of DSP blocks in use.

Acknowledgment

The authors would like to thank Xilinx, CMC and emSYSCAN and NSERC for the funding and resources provided for this work.

References

[1] C. Zhang et al. Caffeine: Towards uniform representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[2] U. Aydonat et al. An OpenCL™Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’17, pages 55–64, New York, NY, USA, 2017. ACM.

[3] NVIDIA. NVIDIA Tesla V100 GPU Architecture. Technical report, 06 2017.

[4] S. Gupta et al. Deep learning with limited numerical precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML’15, pages 1737–1746. JMLR.org, 2015.

[5] M. Courbariaux, Y. Bengio, and J.-P. David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[6] R. DiCecco et al. Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268, Dec 2016.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[9] Xilinx Inc. Vivado Design Suite User Guide, High-Level Synthesis, April 2017.

[10] Xilinx Inc. LogiCORE IP Floating-Point Operator v7.0 Product Guide, April 2014.

[11] M. D. Ercegovac and T. Lang. {CHAPTER} 8 - Floating-Point Representation, Algorithms, and Implementations . In M. D. Ercegovac and T. Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 396 – 487. Morgan Kaufmann, San Francisco, 2004.

[12] C. Szegedy et al. Going Deeper with Convolutions. *CoRR*, abs/1409.4842, 2014.

[13] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 684–691, Dec 2012.

[14] S. Chetlur et al. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014.

[15] Y. Lecun et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[16] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, 04 2009.