ISCAS 2020 Virtual, October 10-21
IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS
CAS IEEE CIRCUITS AND SYSTEMS SOCIETY

Training Progressively Binarizing Deep Networks Using FPGAs

Corey Lammie, Wei Xiang, and Mostafa Rahimi Azghadi

College of Science and Engineering, James Cook University, Queensland 4814, Australia

2020 IEEE International Symposium on Circuits and Systems
Virtual, October 10-21, 2020

Overview

- Background and related works
- Motivation
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- Conclusion and outlook
- Acknowledgements

Overview

- **Background and related works**
- Motivation
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- Conclusion and outlook
- Acknowledgements

Background and related works

- Training Neural Networks (NNs) is computationally expensive.
- *Quantization* and *approximation* strategies can be used to reduce computational complexity.

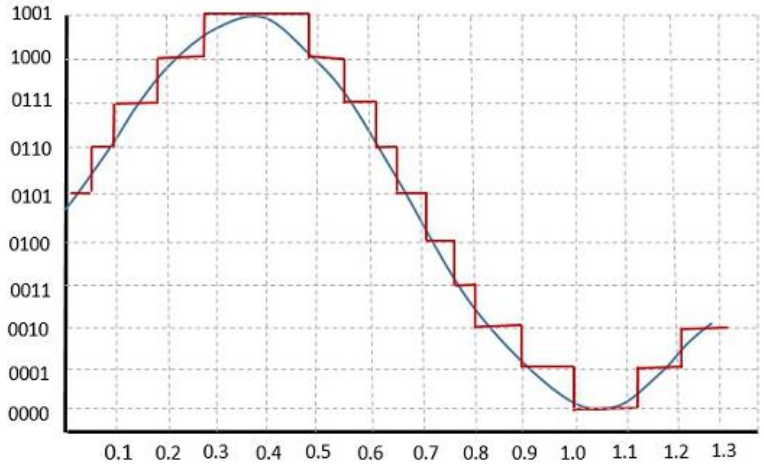

Fig. 1: Depiction of a quantized signal [1].

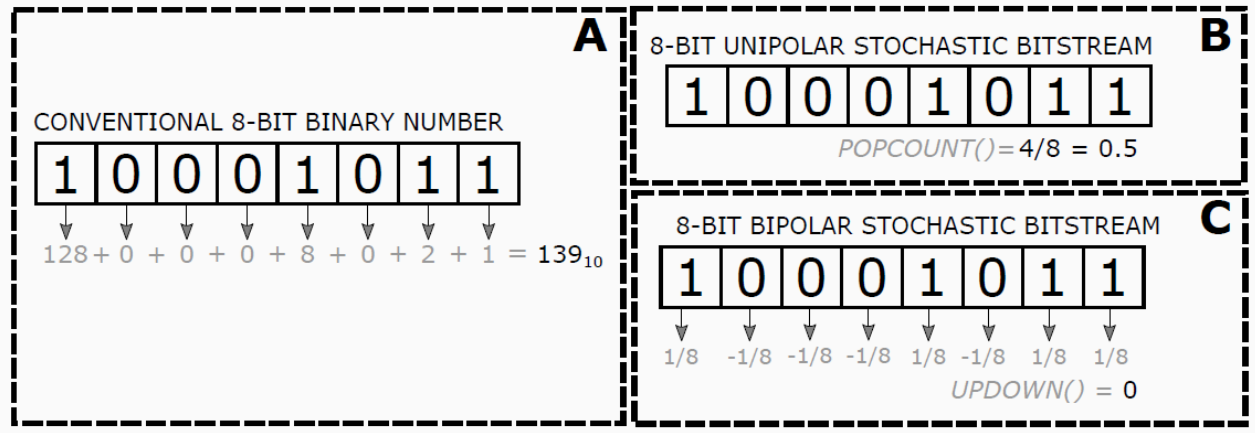

Fig. 2: Depiction of binary, unipolar, and bipolar stochastic bitstreams [2].

Background and related works

Quantization strategies can include:

- Fixed-point and integer number representations;
- Binarized Neural Networks (BNNs);
- Ternary Neural Networks (TNNs).

Approximation strategies can include:

- Stochastic Computing (SC);
- Piecewise Linear (PWL) functions;
- COordinate Rotation DIgital Computer (CORDIC).

Background and related works

Quantization strategies can include:

- Fixed-point and integer number representations;
- **Binarized Neural Networks (BNNs);**
- Ternary Neural Networks (TNNs).

Approximation strategies can include:

- Stochastic Computing (SC);
- Piecewise Linear (PWL) functions;
- COordinate Rotation DIgital Computer (CORDIC).

Background and related works

- Binarized Neural Networks (BNNs) quantize network parameters to binary states:

$$\theta_b = \text{sign}(\theta) = \begin{cases} -1 \text{ if } \theta \leq 0 \\ +1 \text{ otherwize} \end{cases} \text{ or } \theta_b = \begin{cases} +1 \text{ with probability } \rho = \sigma(w) \\ -1 \text{ with probability } 1 - \rho \end{cases} \quad (1)$$

- During backward propagations, large parameters are clipped, and gradients are approximated, as the signum function is not continuously differentiable:

$$\frac{\partial c}{\partial \theta} = \frac{\partial c}{\partial \theta_b} 1_{|\theta| \leq t_{clip}} \quad (2)$$

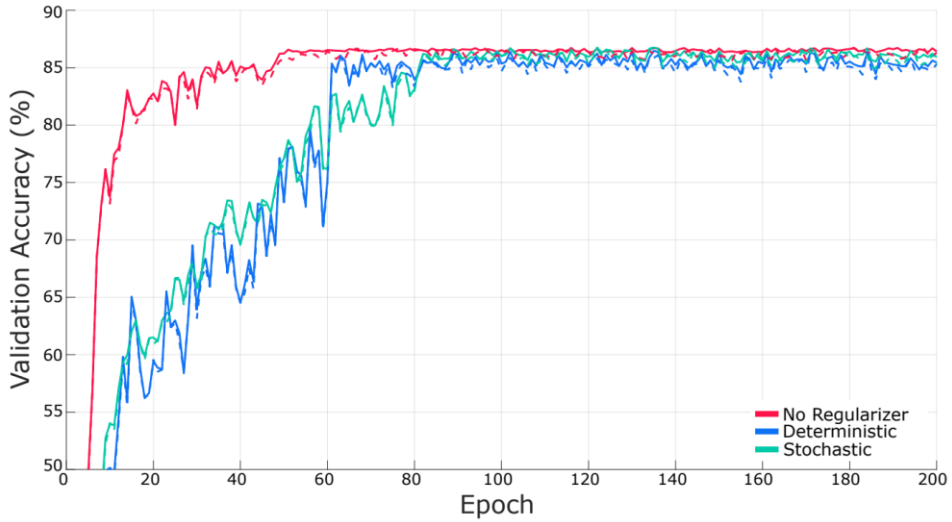

Fig. 3: The validation accuracy of full precision, deterministic, and stochastic BNNs [3] on the CIFAR-10 [4] dataset.

Overview

- Background and related works
- **Motivation**
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- Conclusion and outlook
- Acknowledgements

Motivation

- During training, conventional BNNs require two distinct sets of network parameters.
- Accuracy is commonly reduced, as gradients are approximated using a Straight Through Estimator (STE) [5].
- The training routines of conventional BNNs are inherently unstable.

Overview

- Background and related works
- Motivation
- ***Implementation details*:**
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- Conclusion and outlook
- Acknowledgements

Implementation details

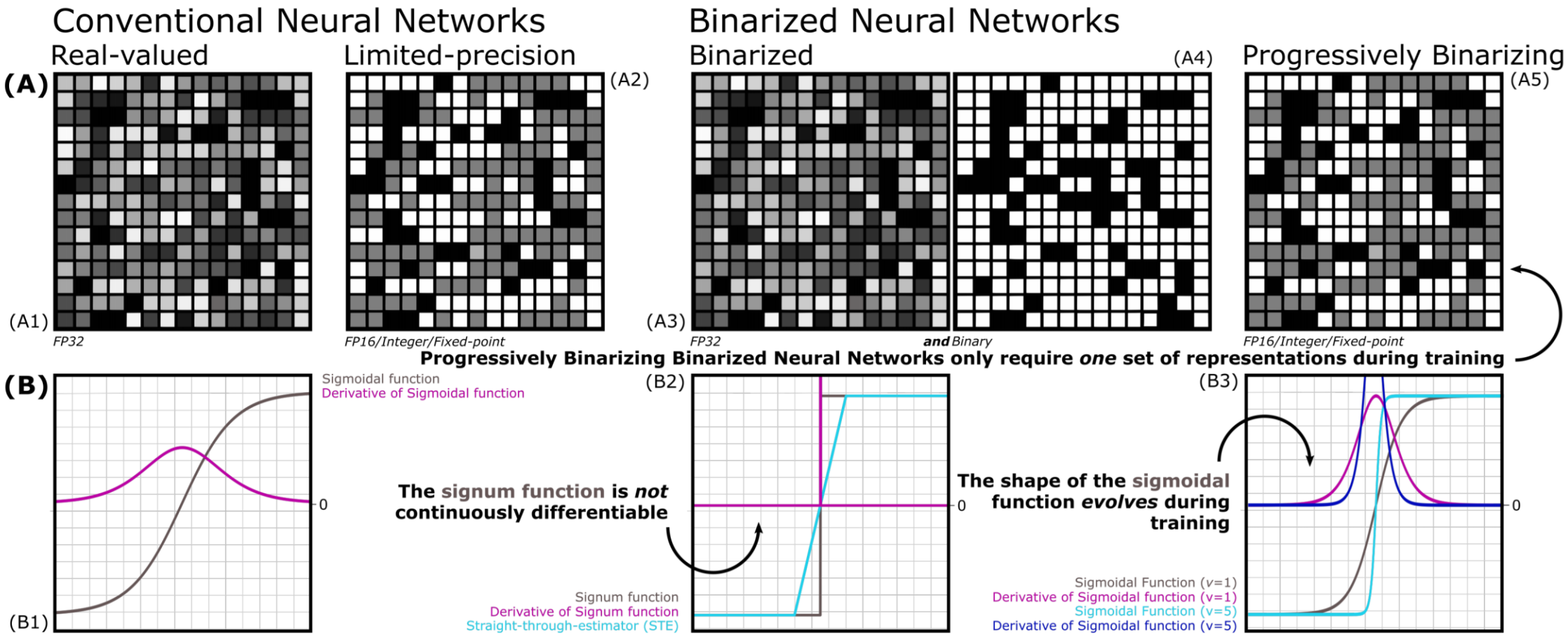

Fig. 4: Depiction of network parameter representations and activation functions required during training for conventional and BNNs.

Implementation details

- Progressively Binarizing Neural Networks (PBNNs) use a set of constrained real-valued parameters θ_l at each layer *l*, which are not directly learnable, but are a function of learnable parameters *P* at each layer.
- During training, as *v* increases, the shape of $\theta(P)$ better mimics that of the signum function.
- The final binary parameters can simply be obtained by passing $\theta(P)$ through the signum function.
- We employ a PWL function to approximate the hyperbolic tangent function:

$$\theta(P) = \begin{cases} -1 & if\ P < -1 \\ vP & if\ -1 \leq P \leq 1 \\ +1 & if\ P > 1 \end{cases} \quad (3)$$

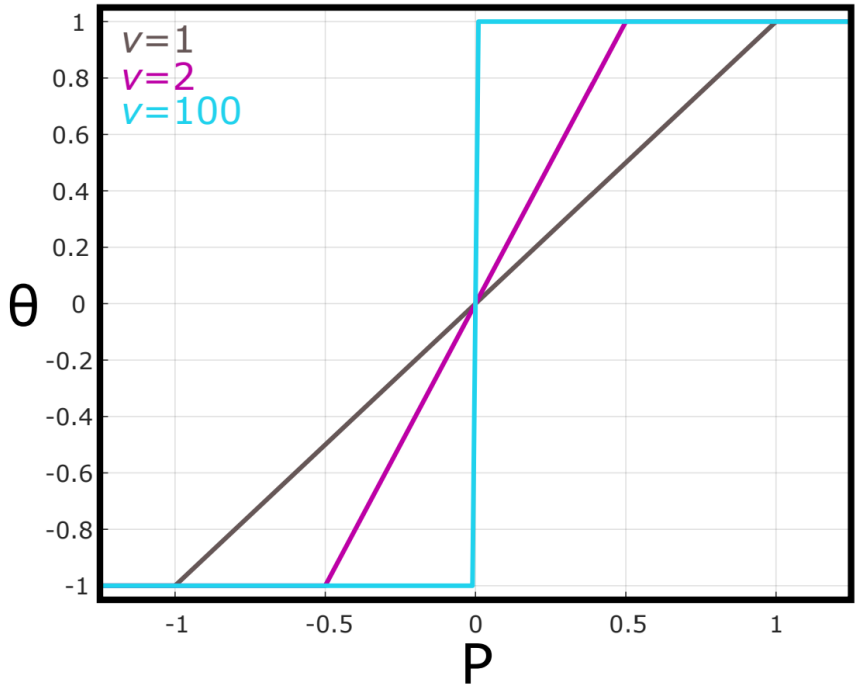

Fig. 5: Activation and binarization functions used for our progressive binarization training routine.

Implementation details

- The derivative of (3), when bounded, is constant and does not depend on *P*.

$$\theta(P) = \begin{cases} -1 & if\ P < -1 \\ vP & if\ -1 \leq P \leq 1 \\ +1 & if\ P > 1 \end{cases} \quad (3)$$

- As v ≫1, the sign of the output of Batch Normalization (BN) is reformulated [6] to reduce computation as per (4) and (5):

$$sign(a_l[n]) = XNOR(l > T, \gamma > 0) \quad (4)$$

$$T = \mu_r - \frac{\sigma_r \beta}{\gamma} \quad (5)$$

- Our training routine is presented in Algorithm 1.

Algorithm 1 The training rotuine adopted by all of our progressively-binarizing DNNs.

Input: Network hyperparameters (the learning rate schedule, η, scale parameter schedule, $\mathbf{v}$, batch size, $\Im$, gradient optimizer, loss function, $\mathbf{J}(\theta, \mathbf{y_i'}(a_{-1}), \mathbf{y_i})$, and the number of training epochs).

Output: Trained binary weights and biases, $\theta_\mathbf{b}$.

for each training epoch **do**

1. Forward Propagation

$\eta, v = \eta[\text{epoch}], \mathbf{v}[\text{epoch}]$

for each training batch **do**

for each layer **do**

Determine $\mathbf{a_l}[n] = \theta_\mathbf{l}[n-1](\mathbf{P_l}[n-1])$

end for

end for

2. Backward Propagation

Determine $\mathbf{J}(\theta, \mathbf{y_i'}(a_{-1}), \mathbf{y_i})$

for all other layers **do**

Determine $\frac{\partial \mathbf{J}}{\partial \mathbf{a_{l-1}}[n]}$ using $\frac{\partial \mathbf{J}}{\partial \mathbf{a_l}[n]}$ and $\theta_\mathbf{l}[n-1]$

end for

3. Parameter Optimization

for each layer **do**

Determine $\frac{\partial \mathbf{J}}{\partial \theta_\mathbf{l}[n-1]}$ using $\frac{\partial \mathbf{J}}{\partial \mathbf{a_l}[n]}$

Determine $\theta[n]$ using $\frac{\partial \mathbf{J}}{\partial \theta_\mathbf{l}[n-1]}$ and η

end for

end for

4. Determine the Trained Binary Parameters

$\theta_\mathbf{b} = []$

for each layer **do**

$\theta_\mathbf{b} = \text{concat}(\theta_\mathbf{b}, \text{sign}(\theta_\mathbf{l}))$

end for

Implementation details

Table 1: Adopted network architecture.

Layer	Output Shape	Binarized
Convolutional, $f = 128, k = 3, s = 1, p = 1$	$(128 \times 32 \times 32)$	✓
Convolutional, $f = 128, k = 3, s = 1, p = 1$	$(128 \times 32 \times 32)$	✓
Max Pooling, $k = 2, p = 2$	$(128 \times 16 \times 16)$	✓
Convolutional, $f = 128, k = 3, s = 1, p = 1$	$(128 \times 16 \times 16)$	✓
Convolutional, $f = 256, k = 3, s = 1, p = 1$	$(256 \times 16 \times 16)$	✓
Max Pooling, $k = 2, p = 2$	$(256 \times 8 \times 8)$	✓
Convolutional, $f = 256, k = 3, s = 1, p = 1$	$(256 \times 8 \times 8)$	✓
Convolutional, $f = 512, k = 3, s = 1, p = 1$	$(512 \times 8 \times 8)$	✓
Max Pooling, $k = 2, p = 2$	$(512 \times 4 \times 4)$	✓
Fully Connected, $N = 1024$	(1024)	✓
Fully Connected, $N = 1024$	(1024)	✓
Fully Connected, $N = 10$	(10)	

Implementation details

- We trained all networks for 50 epochs with a batch size $\Im = 8$.
- The initial learning rate was $\eta = 1e-3$, which was decayed by an order of magnitude every 20 training epochs.
- During training, each network's scale parameter, *v*, was increased logarithmically, from 1 to 1000.
- (4) was used to determine the output of all batch normalization layers when $v \geq 500$.

$$sign(a_l[n]) = XNOR(l > T, \gamma > 0) \quad (4)$$

- Adam and Cross Entropy (CE) were used.

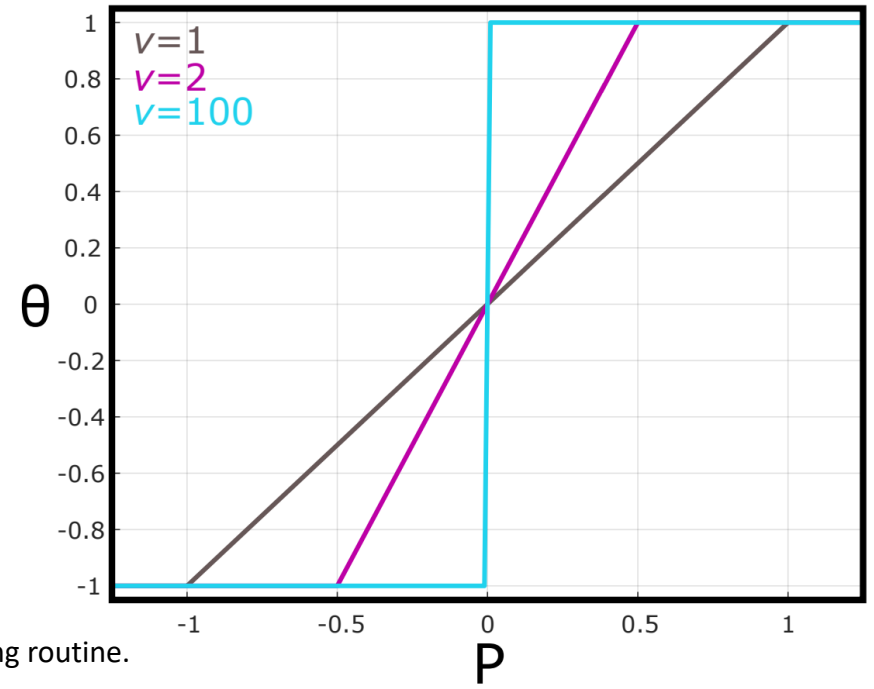

Fig. 5: Activation and binarization functions used for our progressive binarization training routine.

Implementation details

- All our implementations are described using the heterogeneous OpenCL [7] framework.
- The Intel FPGA SDK for OpenCL Offline Compiler (IOC) was used, as part of the Intel FPGA SDK for OpenCL and Quartus Prime Design Suite 18.1.
- A Titan V GPU was used to execute OpenCL kernels and an AMD Ryzen 2700X @ 4.10 GHz Overclocked (OC) CPU was used to drive the host controller.

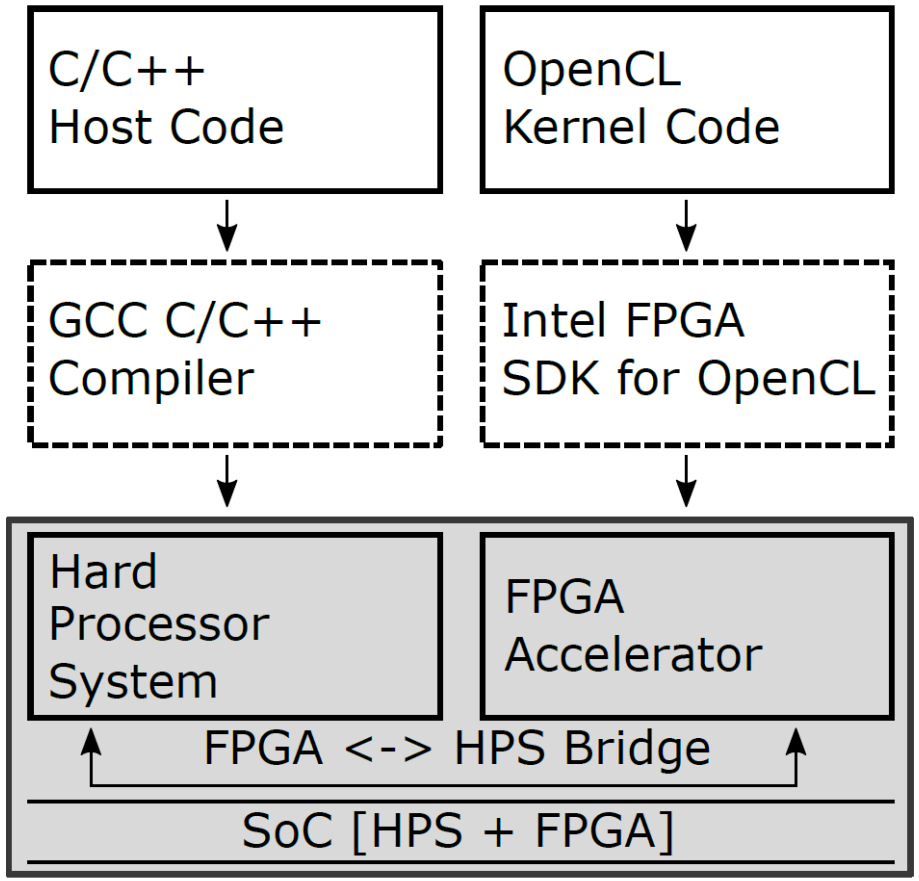

Fig. 6: Top level flow diagram for FPGA implementation [3].

Overview

- Background and related works
- Motivation
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- **Implementation results**
- Conclusion and outlook
- Acknowledgements

Implementation results

Table 2: Implementation results. [1]The mean and standard deviations reported for the Training Time per Epoch (s) metric are determined over 50 training epochs. [2]Similarly to conventional networks, the unbounded ReLU activation function was used. [3]The same test set accuracy was achieved for GPU and FPGA implementations.

Training Routine	Total Kernel Power Usages (W)		Total Training Time (s)		Training Time per Epoch (s)[1]		Test Set Accuracy (%)[3]	
	FPGA	GPU	FPGA	GPU	FPGA	GPU	FPGA	GPU
8-bit Fixed Point								
Stochastic	8.06	133.9	1,592.73	2,613,67	31.85 ± 0.21	52.27 ± 0.37	85.91	85.91
Deterministic	7.95	133.0	1,523.17	2,497.72	30.46 ± 0.18	49.95 ± 0.31	85.56	85.56
Progressive	7.60	130.3	1,383.17	2,315.97	27.66 ± 0.17	46.31 ± 0.32	86.28	86.28
16-bit Fixed Point								
Stochastic	10.19	134.2	1,989.25	3,147.23	39.78 ± 0.17	62.94 ± 0.31	86.45	86.45
Deterministic	10.03	132.8	1,907.17	2,909.62	38.14 ± 0.19	58.19 ± 0.36	86.16	86.16
Progressive	9.27	130.5	1,729.32	2,685.22	34.58 ± 0.22	53.70 ± 0.34	86.94	86.94
FP32 Baseline								
Real-valued[2]	—	137.1	—	2,524.20	—	50.48 ± 0.35	—	86.77

Implementation results

Table 3: Comparison of device FPGA utilization for various binarization training approaches.

Training Routine	**Deterministic**	**Stochastic**	**Progressive**
Device	Intel FPGA OpenVINO		
Dataset	CIFAR-10		
	8-bit Fixed Point		
Flip Flops (%)	63.19	66.42	62.95
ALMs (%)	81.38	84.87	76.92
DSPs (%)	100.00	100.00	93.20
	16-bit Fixed Point		
Flip Flops (%)	96.06	98.43	91.96
ALMs (%)	90.40	94.31	85.54
DSPs (%)	100.00	100.00	100.00

Overview

- Background and related works
- Motivation
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- **Conclusion and outlook**
- Acknowledgements

Conclusion and outlook

- We proposed and implemented novel and scalable PBNNs on GPUs and FPGAs.
- We compared our approach to conventional BNNs and real-valued DNNs using GPUs and FPGAs and demonstrated notable reductions in power and resource utilizations for CIFAR-10.
- This was achieved through approximations and hardware optimizations, as well as using only one set of network parameters compared to conventional BNNs.
- We leave further hardware-level dissemination, upscaling, hyperparameter optimization, and tuning to future works.

Overview

- Background and related works
- Motivation
- *Implementation details*:
  - Progressive binarization training routine;
  - Network architecture and hyperparameters;
  - Hardware architecture.
- Implementation results
- Conclusion and outlook
- **Acknowledgements**

Acknowledgements

- The James Cook University Domestic Prestige Research Training Program Scholarship;
- My supervisor- Dr. Mostafa Rahimi Azghadi;
- My co-supervisor- Prof. Wei Xiang.

References

[1] R. Bhaduri, S. Bonnerjee, and S. Roy, “Onset detection: A new approach to QBH system,” arXiv e-prints, p. arXiv:1908.07409, Aug. 2019.

[2] C. Lammie and M. R. Azghadi, “Stochastic Computing for Low-Power and High-Speed Deep Learning on FPGA,” in IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan., May 2019.

[3] C. Lammie, W. Xiang, and M. R. Azghadi, “Accelerating Deterministic and Stochastic Binarized Neural Networks on FPGAs Using OpenCL,”

in IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Dallas, TX., 2019, pp. 626–629.

[4] A. Krizhevsky et al., “Learning Multiple Layers of Features from Tiny Images,” Citeseer, Tech. Rep., 2009.

[5] M. Courbariaux and Y. Bengio, “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” CoRR, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[6] F. Lahoud, R. Achanta, P. Márquez-Neila, and S. Süsstrunk, “Self-Binarizing Networks,” CoRR, vol. abs/1902.00730, 2019. [Online].

Available: http://arxiv.org/abs/1902.00730

[7] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” Computing in Science Engineering, vol. 12, no. 3, pp. 66–73, 2010.

Q&A