FINN-*R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks

MICHAELA BLOTT, THOMAS B. PREUßER, NICHOLAS J. FRASER, GIULIO GAMBARDELLA, KENNETH O'BRIEN, and YAMAN UMUROGLU,
Xilinx Research Labs, Ireland
MIRIAM LEESER, Northeastern University, U.S.
KEES VISSERS, Xilinx Research, U.S.

16

Convolutional Neural Networks have rapidly become the most successful machine-learning algorithm, enabling ubiquitous machine vision and intelligent decisions on even embedded computing systems. While the underlying arithmetic is structurally simple, compute and memory requirements are challenging. One of the promising opportunities is leveraging reduced-precision representations for inputs, activations, and model parameters. The resulting scalability in performance, power efficiency, and storage footprint provides interesting design compromises in exchange for a small reduction in accuracy. FPGAs are ideal for exploiting low-precision inference engines leveraging custom precisions to achieve the required numerical accuracy for a given application. In this article, we describe the second generation of the FINN framework, an end-to-end tool that enables design-space exploration and automates the creation of fully customized inference engines on FPGAs. Given a neural network description, the tool optimizes for given platforms, design targets, and a specific precision. We introduce formalizations of resource cost functions and performance predictions and elaborate on the optimization algorithms. Finally, we evaluate a selection of reduced precision neural networks ranging from CIFAR-10 classifiers to YOLO-based object detection on a range of platforms including PYNQ and AWS F1, demonstrating new unprecedented measured throughput at 50 TOp/s on AWS F1 and 5 TOp/s on embedded devices.

CCS Concepts: • **Hardware → Integrated circuits**; **Reconfigurable logic and FPGAs**; **Hardware accelerators**;

Additional Key Words and Phrases: Neural network, artificial intelligence, FPGA, quantized neural networks, convolutional neural networks, FINN, inference, hardware accellerator

This project has received funding from the European Union's Framework Programme for Research and Innovation Horizon 2020 (2014–2020) under the Marie Skłodowska-Curie Grant Agreement No. 751339. Miriam Leeser is supported in part by the National Science Foundation under Grant No. 1717213.

Authors' addresses: M. Blott, T. B. PreußEr, N. J. Fraser, G. Gambardella, K. O'brien, and Y. Umuroglu, Xilinx Research, 2020 Bianconi Ave, Citywest Business Campus, Saggart, Dublin 24 Ireland; emails: michaela.blott@xilinx.com, thomas.preusser@utexas.edu, {nfraser, giulio.gambardella, kenneth.obrien, yamanu}@xilinx.com; M. Leeser, 316 Dana Research Center, 360 Huntington Avenue, Boston, MA 02115, United states; email: mel@coe.neu.edu; K. Vissers, 2100 Logic Drive, San Jose, CA 95124-3400, United states; email: kees.vissers@xilinx.com.

1936-7406/2018/12-ART16 $15.00
https://doi.org/10.1145/3242897

ACM Reference format:
Michaela Blott, Thomas B. PreußEr, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-*R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 16 (December 2018), 23 pages.
https://doi.org/10.1145/3242897

1 INTRODUCTION

Deep Neural Networks (DNNs) achieve impressive results in computer vision, speech recognition, and many other applications. However, the enormous compute and storage requirements associated with their deployment encounter limitations in regards to cost, power budget, throughput, and latency. Energy costs and data transmission overheads of offloading to the cloud are unsuitable for low-latency, real-time, or safety-critical application requirements, such as speech recognition, augmented reality, drone control, or autonomous driving. Also in the cloud computing context itself, we face ever-increasing throughput requirements to process astronomical scales of data, bringing additional challenges in energy efficiency to minimize operating expenses. While cloud service latency is less critical compared to embedded scenarios, it still translates directly into customer experience for interactive applications. For instance, Jouppi et al. [27] quote a response time limit of 7ms for an interactive user experience in cloud-based services.

Neural networks using floating-point operations are the initial choice from the machine-learning community, but trained parameters can contain a lot of redundant information [21]. This can be exploited to reduce the computational cost of inference. Competitive levels of accuracy have been demonstrated using sparsification [36], singular value decomposition [14], pruning [18, 21, 65], or a combination of techniques [22, 25]. Another promising key approach for exploiting redundancy in neural networks is to use *quantization*. It has been demonstrated that moving from floating-point arithmetic to low-precision integer arithmetic only impacts the network accuracy lightly, especially if the network is retrained [57]. The quantization of neural networks down to 8-bit integers has been widely adopted and is well supported by designated and optimized software libraries, such as gemmlowp [16] and the ARM Compute Library [35]. In this article, we are interested in even further reduced precisions all the way down to the extreme case of Binary Neural Networks (BNNs) with binary representations for synapse weights, input and output activations, as these precisions offer the lowest hardware cost. We also leverage pruning techniques to furthermore reduce the number of operations in several networks, while having little impact on accuracy.

The key computational advantages obtained by using quantized arithmetic in inference are threefold: First, the quantized weights and activations have a significantly lower memory footprint and the working set of some quantized neural networks (QNNs) may entirely fit into on-chip memory. Fewer off-chip memory accesses also mean orders of magnitude less energy consumption. Furthermore, as on-chip memory can deliver much higher bandwidth, the utilization of the compute resources is increased for a higher performance. For example, binarization reduces the model size of VGG-16 [55] from 4.4Gbit to 138Mbit and that of YOLOv2 [53] from 1.6Gbit to 50Mbit. Second, replacing floating-point with fixed-point representations inherently reduces processing power by orders of magnitude [24]. While the reported study is for 45nm ASICs, FPGAs follow similar general trends. Finally, the hardware resource cost of quantized operators is significantly smaller than that of floating-point ones. This allows a much higher compute density with the same amount of resources and thereby an easier performance scaling, as the key computation in neural network inference is repeated multiply-accumulate (MAC) operations. As will be shown in Section 3, hardware complexity of an integer MAC basically grows proportional to the bitwidths of both factors.

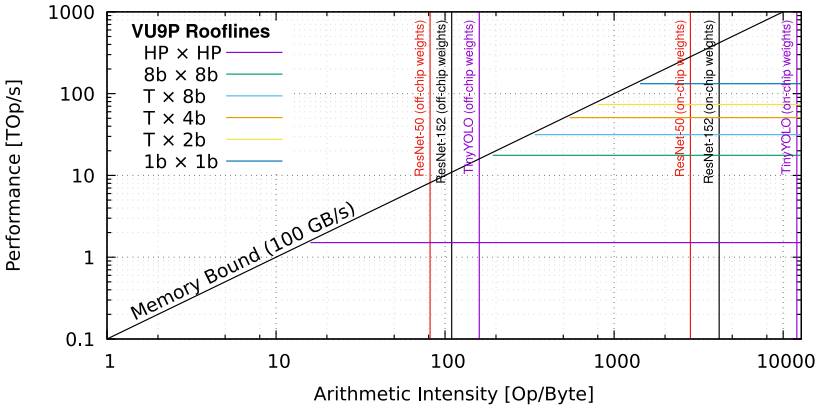


Fig. 1. AWS F1 Rooflines across a range of precisions.

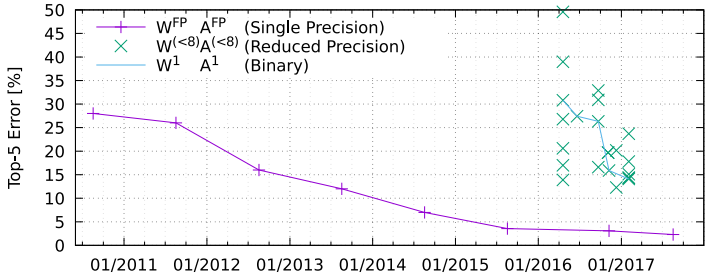


Fig. 2. Accuracy over time for ImageNet classification.

We use a roofline model, as introduced by Williams et al. [62], to exemplify the computational benefits of QNNs. Figure 1 shows the peak performance of an AWS F1 FPGA node for operations at different precisions.[1] The graphs illustrate that by going from half-precision floating point to reduced-precision fixed point, the peak compute performance increases by 87×. We also show how the accompanying reduction in model size enables staying in on-chip memory. This is illustrated by the arithmetic intensity of three different neural network topologies, specifically ResNet-50, ResNet-152, and Tiny YOLO, which are shown for the two cases of the half-precision floating-point and 1-bit operation. The arithmetic intensity for the 1-bit variants is greatly increased, and the implementation is no longer memory bound.

The reduction in precision has a slight impact on accuracy, which has been addressed by quantization-aware training techniques, by innovative numerical representations (e.g., MS-FP8 by Microsoft Research [8]) and by new quantization schemes (e.g., Half-wave Gaussian [9]). This is illustrated by Figure 2 showing the published top-5 error rate for the ImageNet classification by 32-bit floating-point networks [1] and corresponding reduced-precision networks [9, 21, 25, 50, 66, 72, 73].

The minimal reduction in accuracy combined with significant performance gains and power savings provides highly interesting design trade-offs. Figure 3 illustrates achievable compromises for the selection of CNNs given above, depicting the relationship between top-5 error rate and hardware cost for ImageNet classification. For this, we assume a target frame rate of 10,000 frames per second (fps), clock rate of 300MHz, and a hardware cost in lookup tables (LUTs) as derived by

[1]The following assumptions were applied: clock frequency: 400MHz, 90% DSP, and 70% LUT utilization; HLS overhead included by hardware cost functions as derived later.

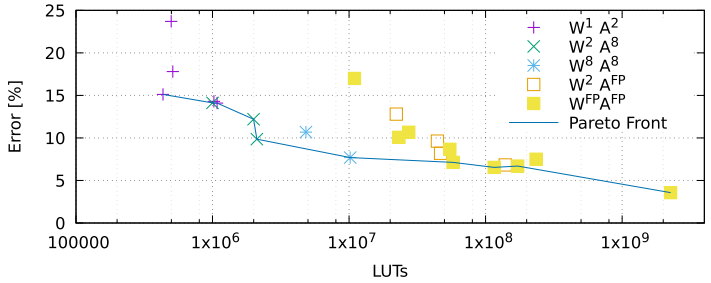


Fig. 3. Accuracy vs. hardware cost with different precisions for ImageNet.

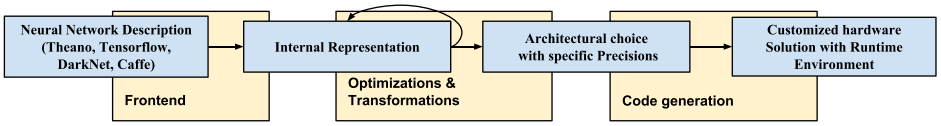


Fig. 4. FINN-*R* framework overview.

the microbenchmarks in Section 3 with HLS. The interesting points within this design spectrum are the ones on the Pareto frontier. It is clear that, for example, for a maximum error of 10%, the most cost-efficient implementation leverages a 2b/8b representation. Similarly, these graphs illustrate that Pareto optimal trade-offs are often reduced-precision networks, for instance when the hardware cost or energy budgets are fixed. This is the case in many applications, be it a maximum price target for an embedded device or the PCIe power budget of 75W.

The key question is, given a set of design constraints and a specific machine-learning task, how the best possible trade-off within the vast design space can be identified. This question entails two parts: One is concerned with deriving the most implementation-friendly neural network, and the second part looks at the hardware implementation itself. Deriving systematic accuracy results is a time-intensive process given typical neural network training times. But even the potential computational benefits cannot be easily understood, because hardware implementations are time-intensive, the deployment space is complex, architectural choices are myriad and the prediction of power, performance and latency is complex. To find an optimal implementation given a set of design constraints requires a framework that provides insights and estimates given a set of design choices and automates the customization of the hardware implementation.

To address this challenge, we implemented FINN-*R*, the second version of the original tool [58], which supports more architectural choices as well as mixed and variable precisions beyond binary. FINN-*R* uses a quantization-aware intermediate representation to enable QNN-specific optimizations and has a modular frontend/transform/backend structure for flexibility as is shown in Figure 4. The focus of this article is on the architecture of inference accelerators, their optimization and automated generation towards different design targets. While we discuss some techniques used towards the quantization of neural networks during training, this is currently still a highly active research area and well beyond the scope of this article. Our contributions are as follows:

- Review and summary of the state of the art in reduced-precision neural networks, accuracy, frameworks, and reconfigurable hardware accelerators.
- Microbenchmarks demonstrating performance-cost tradeoffs of different compute architectures and substrates for a broad range of precisions.
- Cost models for different architectures and precisions.
- A modular quantization-aware end-to-end framework for QNN exploration and implementation.

Table 1. Latest Accuracy of Several QNNs [4, 9, 73] on ImageNet Dataset

Network	Float top-1 (top-5)	QNN top-1 (top-5)
GoogLeNet	71.4% (90.5%)	63.0% (84.9%)
VGG-like	69.8% (89.3%)	64.1% (85.6%)
ResNet-50	79.26 (94.75%)	64.6% (85.9%)

- Experimental results on four state-of-art QNN implementations on three different platforms yielding unprecedented measured peak performance.

This article is structured as follows: Section 2 reviews the state of the art in reduced precision neural networks. We describe the hardware architecture choices in greater detail in Section 3, including the microbenchmark results, which are used to derive hardware cost estimation functions, while Section 4 contains the details on the FINN-*R* framework. Section 5 presents experimental results, demonstrating the benefits of reduced precision and validating the FINN-*R* workflow and its flexibility with results measured for a range of neural networks, platforms, and precisions. Finally, Section 6 concludes the article and provides an outlook to future work.

2 BACKGROUND

QNN accuracy, hardware inference accelerators, and frameworks to automate the accelerator generation are fast-moving fields of research. The following sections capture the respective most recent and relevant state of the art at the time of writing.

2.1 Quantized Deep Neural Networks—Accuracy

On smaller image classification benchmarks such as MNIST, SVHN, and CIFAR-10, QNNs have been demonstrated [13, 72] to achieve nearly state-of-the-art accuracy. Kim and Smaragdis [29] consider full binarization (where weights, inputs and outputs are binarized) with a predetermined portion of the synapses having zero weight, and all other synapses with a weight of one. They report 98.7% accuracy with fully connected networks on the MNIST dataset and observe that only XNOR and bitcount operations are necessary for computing with such neural networks. XNOR-Net by Rastegari et al. [50] applies convolutional BNNs on the ImageNet dataset with topologies inspired by AlexNet, ResNet, and GoogLeNet, reporting top-1 accuracies of up to 51.2% for full binarization and 65.5% for partial binarization (where only part of the components are binarized). DoReFa-Net by Zhou et al. [72] explores reduced precision during the forward pass as well as the backward pass. Their results include configurations with partial and full binarization on the SVHN and ImageNet datasets, including best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarization. For the more challenging ImageNet benchmark, there is a noticable accuracy drop when using QNNs compared to their floating-point equivalents; however, there is significant evidence that increasing network layer size can compensate for this drop in accuracy as shown by by Fraser et al. [20], Sung et al. [57], Zagoruyko et al. [66], Mishra et al. [39], and Kim et al. [29].

Furthermore, new quantization schemes show promising results. For instance, Cai et al. [9] proposed Half-wave Gaussian Quantization (HWGQ) to take advantage of the Gaussian-like distribution of batch-normalized activations, demonstrating QNNs with binary weights and 2-bit activations with less than 5% top-5 accuracy drop compared to floating-point DNNs on the challenging ImageNet dataset, as summarized in Table 1. To the best of our knowledge, currently the lowest error rates for ImageNet classification have been achieved using ternarization [4, 73].

While different numerical representations are worth investigation, our current focus is on quantized values with fixed integer representations below 8 bits. We use *integer* to also refer to

fixed-point numbers as we can absorb fixed-point scaling factors into thresholds. The following notation W^xA^y is used across this article to represent a layer with x-bit weights and y-bit activations. The QNN networks within this article have most layers heavily quantized but may contain higher-precision or even floating-point layers. As pointed out before, the discussion on how to train for reduced precision is referenced in the experimental section for each QNN.

2.2 Accelerators and Architectures

A great deal of prior work on mapping neural networks to hardware exists for both FPGAs and ASICs. We refer the reader to the work by Misra and Saha [40] for a comprehensive survey. We cover a recent and representative set of works here, roughly dividing them into three categories based on their basic architecture: (1) a single processing engine [5, 11, 12, 27, 28, 34, 41, 46, 68], usually in the form of a systolic array, which processes each layer sequentially; (2) a streaming architecture [4, 49, 60], consisting of one processing engine per network layer where inputs are streamed through the dataflow architecture and all layers are computed in parallel; (3) a vector processor [17] with instructions specific to accelerating the primitive operations of convolutions. Recent graphics processing units (GPUs) have been shown to deliver competitive results, as well as neurosynaptic processors, which implement many digital neurons and their interconnecting weights [15]. However, this study focuses on FPGA-based accelerators only. For a detailed performance and efficiency comparison, we refer the reader to Section 5 and particularly Table 5.

Single Processing Engines: Zhang et al. [68] describe a systolic array style architecture, using theoretical roofline models to design accelerators optimized for the execution of each layer. Ovtcharov et al. [46] implement a similar style architecture achieving a 3× speedup. Eyeriss by Chen et al. [12] use 16-bit fixed point rather than floating point, and combine several different data reuse strategies, which provide 2.5× better energy efficiency over other methods. YodaNN by Andri et al. [5] have a similar design as Zhang et al. [68] but explore binary weights for fixed sized windows. Moss et al. [41] also implement a systolic array style processor, but specifically for BNNs, allowing for very high throughput, up to 40.8 TOp/s. Some alternative approaches to single accelerator designs are: Stripes [28], which implements a bit-serial processor capable of handling multiple precisions on a single compute array. The authors experiment with precision from 3 to 13 bits. FP-BNN, another implementation of a single processing engine for BNNs, utilizing an XNOR-popcount datapath. Interestingly, the authors implement batch-normalization and scaling in floating point for some networks, resulting in higher DSP usage than perhaps is required.

Streaming architectures: Venieris and Bouganis [60] proposed a synchronous dataflow (SDF) model for mapping CNNs to FPGAs, which is a similar approach to our dataflow variant. Their designs achieve up to 1.62× the performance density of hand tuned designs. Alemdar et al. [4] implement fully connected ternary-weight neural networks with streaming and report up to 255K frames per second on the MNIST dataset, but concentrate on the training aspect for those networks. Baskin et al. [7] similarly map multibit CNNs in streaming fashion onto FPGAs and show superior performance over GPUs. Prost et al. [49] design ternary networks in a dataflow, achieving notably high accuracies and performance, likely due to ternarization and a hand optimized RTL design.

Vector processors: Farabet et al. [17] describe a programmable ConvNet Processor (CNP), which is a RISC vector processor with specific macro-instructions for CNNs including 2D convolutions, 2D spatial pooling, dot product and an elementwise non-linear mapping function. The authors also created a tool to compile a network description into host code, which is used to call the CNP.

Accelerator frameworks: Numerous new frameworks, including the original FINN tool [58], have been proposed that take a graph-based description of neural networks (such as Caffe's prototxt format) to operational hardware implementations, whereby some of those leverage a fixed hardware architecture, and others customize the hardware accelerator to achieve better throughput, latency

or power reduction. To the best of our knowledge FINN-*R* is the only tool that supports arbitrary precision in weights, input and output activations, plus the flexibility in the backend, which includes two hardware architectures to support a spectrum of design goals, as well as multiple target platforms. *DNNWeaver* [54] is a tool that generates bitstream+host code implementing CNNs on several FPGA platforms on the basis of a Caffe prototxt description. The generated coprocessor implements multiple processing engines depending on available resources. External memory is used for weights and for intermediate feature maps while the arithmetic supports 16-bit fixed or floating point. Similarly, *fpgaConvNet* is not quantization-aware. It creates customized FPGA dataflow architectures using reconfiguration when designs do not fit [60]. *CaffePresso* focuses on embedded systems with 20W power budgets including the Xilinx ZC706 (FPGA), NVIDIA Jetson TX1 (GPU), TI Keystone II (DSP), and Adapteva Parallella (RISC+NoC). The tool is currently limited to low-complexity classifiers that operate on small image maps and few class labels. Combining auto-tuning of the implementation parameters, and platform-specific constraints deliver optimized solutions for each input ConvNet specification. *GUINNESS* [43] is a GUI-based tool flow for training BNNs on GPUs and deploying them using Xilinx devices through SDSoC. Similarly, *HADDOC2* [3], the synthesis tool described by Wei et al. [61] and the compiler proposed by Ma et al. [37] transform CNN descriptions to synthesizable hardware. Finally, *Minerva* [51] proposed a five-stage SW-HW co-design work flow of an inference engine, however, it is constrained to fully connected (FC) layers only. Unlike FINN-*R*, Minerva includes a training space exploration, which is used to explore accuracy/resource trade-offs with a layer-wise direct quantization scheme, synapse pruning using thresholding and SRAM fault mitigation.

3 INFERENCE ACCELERATOR ARCHITECTURE

In this section, we investigate the various possible architectural choices when mapping inference onto programmable logic. The first subsection takes an in-depth look at reduced precision operational costs for both LUT- and DSP-based implementations. This includes systematic benchmarking results for a broad choice of precisions, which we use as the basis for our *operation cost function*. In the second subsection, we discuss the implementation of individual layers, including their related *layer cost function*. The third subsection elaborates on the supported choices for a complete inference accelerator and their associated impact on the resource requirements yielding the *accelerator cost function*. The cost functions are essential to obtain the optimal levels of parallelism for the architecture and performance predictions.

3.1 Microbenchmarks and Operation Cost Function

For a thorough understanding of the resource requirements of the omnipresent dot product computation, we have designed and implemented a set of microbenchmarks that perform multiple multiply-accumulate operations that can be customized in the bit widths of both operand vectors and the number of products summed up in a single step. We have implemented these microbenchmarks both (a) in VHDL for traditional RTL synthesis and (b) in C++ targeting the HLS flow. For FINN-*R*, we have chosen the latter for a number of reasons, namely design productivity, portability to different platforms, built-in optimizations for pipelining, design-space exploration and automated flow control. However, this comes at the expense of some resource overhead. The RTL results allow us to estimate the overhead for the higher-level design entry and establish a reference for potential future gains.

Combinations of different dot product sizes, which is defined as the size of the input vectors, $N \in \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128\}$ and operand bit widths $W \times A \in \{1, 2, 3\} \times \{1, 2, 3\}$ were measured, resulting in a three-dimensional parameter space. An elementary multiply decomposes into (a) the generation of a skewed bit matrix produced from $W \cdot A$ AND operations and (b) its

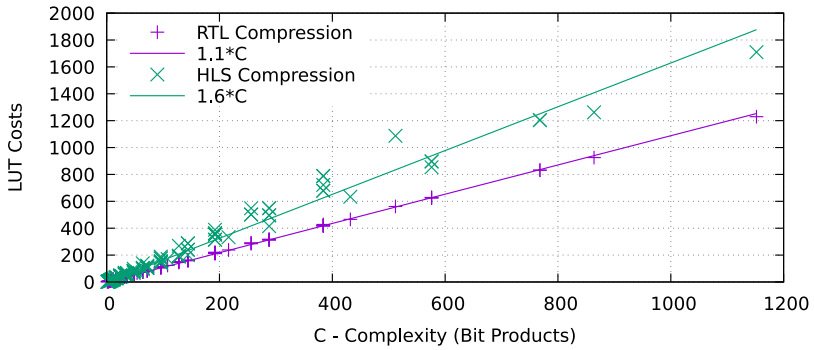


Fig. 5. LUT costs of dot product computation.

summation to yield the value of the product. The structural combinational complexity of both steps is determined by the number of bits produced and added up, respectively. The LUT cost can thus be expected to grow proportionally to this product as well, give or take some jitter caused by functional fragmentation due to the required mapping to physical 6-input LUTs.

Scaling the size N of the dot product to longer vectors can be expected to scale the structural complexity of the computation accordingly. Note that N bit matrices of size $W \cdot A$ must be produced. The following summation would ideally operate on all the stacked and merged matrices together without producing the individual partial results. This additive reduction would have $N \cdot W \cdot A$ inputs suggesting a proportional structural complexity in terms of LUTs.

The described implementation of the multi-MAC operation is effectively enforced in our RTL implementation by employing the generic matrix summation approach by Preußer [48]. The HLS flow through Vivado HLS and Vivado implements a similar slightly less efficient adder tree after the generation of the bit matrices. Both implementations can be expected to induce structural LUT costs that are roughly proportional to the product $C = N \cdot W \cdot A$, which we use to describe the *complexity* of the dot product operation. The coefficients corresponding to the two implementation choices can be determined empirically and is an immediate measure of their relative efficiencies.

The results of our microbenchmark experiments are shown in Figure 5. The anticipated linear dependency on the complexity measured by the number of overall bit matrix elements is confirmed to be extremely close for the RTL implementation while experiencing a somewhat greater variation about the fitted center line for HLS synthesis. HLS typically but unnecessarily reduces the partial products completely to a conventional binary number before feeding them into the adder tree. This creates an overhead that grows with the size N of the dot product and accounts for the observed variation. Comparing the measured coefficients, the HLS implementation is found to currently induce a 45% resource overhead over the far more complex VHDL implementation.

The use of binary ($\{-1, 1\}$) and ternary ($\{-1, 0, 1\}$) weights is very popular for reduced-precision neural networks. So, it is interesting how these specific range types behave in comparison to the neighboring conventional 1-bit or 2-bit integer types when used in dot products with activations of various precisions. Factoring out W from our previous complexity measure, we can predict linear dependencies of the LUT costs on the remaining $C' = N \cdot A$ product. Using HLS only in these experiments, we find the greatest cost increase going from conventional 1-bit weights, which are rarely used, to binary weights. This is due to the fact that already the negative multiple of A now required in the computation mandates the allocation of an extra sign bit. This can only be mitigated in the trivial case of 1-bit activations using the approach practiced by XNOR-Net or traditional binarized FINN [58]. Otherwise, an increase in LUT costs of 35% is experienced. The additional increases of 20% for each further step going to ternary and 2-bit precisions are somewhat smaller.

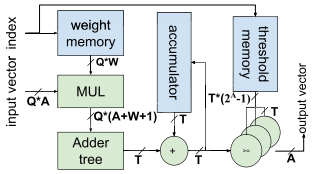
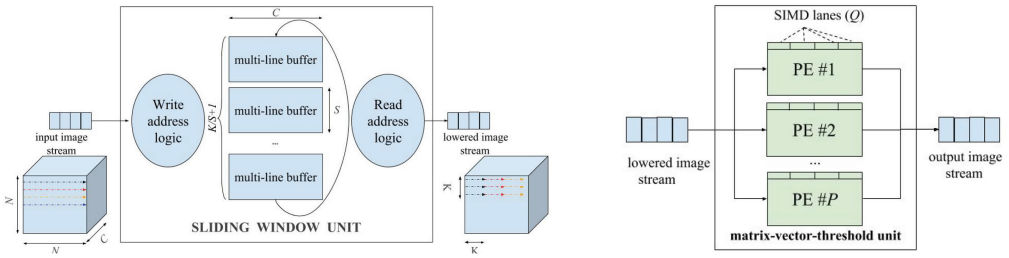


Fig. 6. Processing element (PE) as basic compute component.



(a) Sliding Window Unit (SWU) (b) Matrix-Vector-Threshold Unit (MVU)

Fig. 7. SWU and MVU block diagram.

3.2 Layers

The prinicipal elements that compose a typical convolutional layer are the *matrix-vector threshold unit* (MVU) and the *sliding window unit* (SWU). MVUs handle the compute aspects: For convolutional layers, the convolutions themselves can be *lowered* to matrix-matrix multiplications, which is well understood [10]. These can then be mapped in a streaming fashion onto the MVU. The corresponding weights from the convolution filters are packed into a filter matrix, while a sliding window is moved across input images to form an image matrix. These matrices are then multiplied to generate the output images.

We refer to the principal compute component of convolutional or fully connected layers as a *processing element* (PE). Its structure is illustrated in Figure 6. A PE performs Q parallel multiplications, which corresponds to the SIMD value. It then reduces them in an adder tree for their subsequent accumulation towards the currently computed dot product. Finally, threshold comparisons are used to derive the output values from the accumulation results. An array of P parallel PEs comprises a MVU. A third degree of concurrency is introduced that supports computation of multiple output pixels sharing the same weights within the same channel in parallel, referred to as M. This enables performance scaling with increased BRAM utilization. The choices of the parameters P, Q, and M determine the degree of a layer's computational parallelism. They are the key parameters for trading off resource versus performance of any layer's computation.

The SWU is the unit that generates the image representation required for a convolution lowered to a matrix multiplication (Figure 7(a)). It generates the same vectors as those in Reference [10] but with interleaved channels [58] to simplify memory accesses and to avoid the need for transposition between layers. This exhibits significantly lower latency compared to full image buffers and reduces buffer size requirements. Only as many consecutive rows as the height of the convolutional kernel must be kept available. For elasticity reasons, an extra row is used to collect new incoming image data.

3.2.1 Layer Cost Model. As can be seen from Figure 8, a layer is composed of different elements. Convolutional layers are composed of SWU, MVU as well as weight and threshold memories (WM

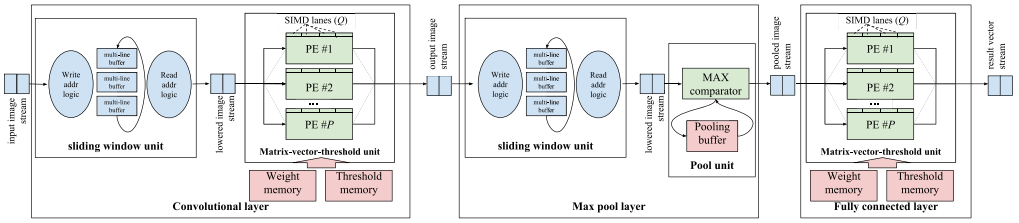


Fig. 8. Possible layer compositions.

Table 2. Layer Parameters

Fully Connected Layers	D, D'	input & output vector size
Convolutional Layers	N, C	input feature map width and channels
	$K \times K, S$	kernel dimension and stride
	C'	output feature map channels
MVU Dimensioning	M	parallel vectors processed by MVU
	P, Q	number of output & input channels computed in parallel
	A, W	bit width (precision) of activations / weights

& TM). Maxpool layers contain a SWU and a maxpool unit, and fully connected layers require only MVUs. Thus, the layer cost is a sum of the basic components as shown in Equation (1). Note that the logic cost relating to WM (LUT_{WM}) and the BRAM cost of MVUs ($BRAM_{\text{MVU}}$) is neglible:

$$BRAM_{\text{CNV}} = BRAM_{\text{SWU}} + BRAM_{\text{WM}}; \quad LUT_{\text{CNV}} = LUT_{\text{SWU}} + LUT_{\text{MVU}}; \tag{1}$$

$$BRAM_{\text{FC}} = BRAM_{\text{WM}}; \quad LUT_{\text{FC}} = LUT_{\text{MVU}}; \tag{2}$$

$$BRAM_{\text{MP}} = BRAM_{\text{SWU}} + BRAM_{\text{MP}}; \quad LUT_{\text{MP}} = LUT_{\text{SWU}} + LUT_{\text{MP}}. \tag{3}$$

In the following paragraphs, we derive BRAM, LUT, and DSP costs for all components (SWU, MVU, WM, MP) separately.

SWU Cost. The sliding window unit's hardware cost is dominated by the BRAM requirements, which can be directly derived from the implemented memory layout, as given by the parameters in Table 2. The line buffer occupies as many BRAM modules as specified by Equation (4):

$$\text{BRAM}_{\text{swu}} = M \cdot \left(\left\lceil \frac{K}{S} \right\rceil + 1 \right) \cdot \left\{ \left\lceil \frac{S \cdot N}{512} \right\rceil \times \left\lceil \frac{C \cdot A}{36} \right\rceil \right\}. \tag{4}$$

The multi-vector count scales linearly at the highest level of the equation. Otherwise, independent stripes of memory are used for each set of rows that can be released independently once the whole width of a line has been processed. An additional memory stripe is used as assembly buffer for the new image data coming in. This accounts for the first parenthesized factor. The remaining two factors capture the depth and the width of the memory stripes, which are potentially fragmented due to the depth and word width of the built-in BRAM modules. There is also a constant overhead in logic resources. This varies depending on the type of accelerator architecture. For a full feed-forward dataflow, each SWU requires 426 LUTs and 0 DSPs as the exact dimensions are known at compile-time and the parameters can be baked into the architecture. For a multilayer offload that executes many different layers on top of the same hardware components, the parameterization happens at run-time, therefore the overhead is larger with 1050 LUTs and 15 DSPs, respectively.

WM Cost. For convolutional layers, Equation (5) captures the number of BRAM modules needed to implement the weight memory of a convolutional layer. Its overall size is determined by the

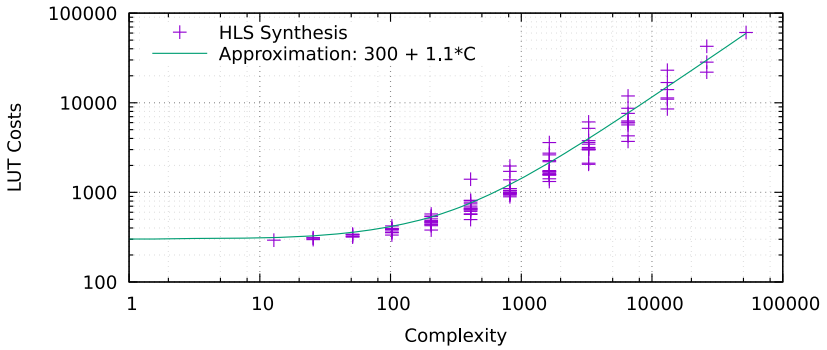


Fig. 9. Empirical fit of the LUT cost model for the MVU.

product of the squared kernel dimension and the numbers of input as well as output feature map channels. This memory volume is split into separate memories, one for each processing element. The parallel access of Q weights determines the word width used by the implementation. Again, memory depth and word size may be fragmented by the physical dimensions of the available BRAM modules:

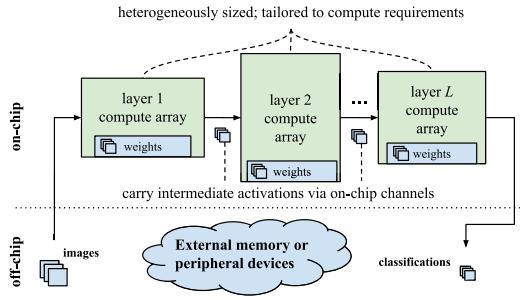
$$\text{BRAM}_{\text{WM}} = P \cdot \left\{ \left\lceil \frac{\omega}{512} \right\rceil \times \left\lceil \frac{Q \cdot W}{36} \right\rceil \right\} \quad \text{with} \quad \omega = \begin{cases} \frac{K^2 \cdot C \cdot C'}{Q \cdot P} & \text{for convolutional layers} \\ \frac{D \cdot D'}{Q \cdot P} & \text{for fully connected layers} \end{cases}. \tag{5}$$

MVU Cost. The computational concurrency of a convolutional layer is controlled by (a) the number P of PEs concurrently working on distinct output channels, (b) Q, the SIMD of input channels processed within one clock cycle, and (c) M, the multi-vector count capturing the concurrent duplication of this compute structure across multiple output pixels for convolutional layers. These parameters allow to scale the performance of a layer implementation in a wide range but also affect the hardware costs directly. Generating a network implementation, FINN-*R* must be aware of these costs to be able to scale the individual layer implementations towards a balanced performance within the resource limits of the targeted device.

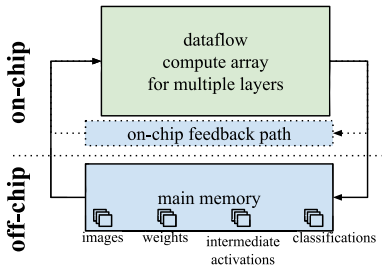
The hardware cost of the MVU can be modeled as an essentially constant control part and the dot product arithmetic. The latter scales both with the duplication into parallel PEs and with parallel multi-vector processing. The model for the internal costs of the individual arithmetic blocks can be taken from the results of the microbenchmarks in Section 3.1. Combining both control and arithmetic, we derive the following model: $LUTs = c_0 + c_1 \cdot M \cdot (P \cdot Q)\,(W \cdot A)$. Recall that $M = 1$ for fully connected layers as they cannot share weights across multiple kernel applications.

To determine the two parameters of this model and to validate its fitness, we again performed HLS synthesis experiments with the parallelization parameters $M = 1$, $P \in \{2, 4, 8, 16, 32, 64\}$ and $Q \in \{2, 4, 8, 16, 32, 64\}$. The complete product scaled by c_1 is taken as the single-figure complexity measure. The obtained empirical fit of the LUT model against the synthesis results is depicted in Figure 9. The anticipated behavior is confirmed; however, the prediction may be off by up to 30% of the later synthesis result for individual experiments.

The cost of the threshold operations depends strongly on the precision of the output activation function as the number of thresholds to be stored and compared with grows exponentially with this precision. For small precisions like the ones FINN-*R* is targeting, these costs practically disappear within the remaining MVU costs. Going for precisions significantly above 4 bits will quickly render the associated costs expensive or simply make the thresholding approach infeasible altogether.



(a) Dataflow Architecture with Per-Layer Tailored Compute Arrays, On-Chip Weights and Activations



(b) Multilayer Offload Architecture with Maximally-Sized Homogeneous Compute Arrays for Different Precisions

Fig. 10. Possible backend architectures.

MP Cost. The BRAM and LUT requirements for the actual compute of the max pooling layers is very little. The block is basically implementing C parallel comparators, one for each channel whereby each sequentially compares two A-bit words holding onto the maximum of its pooling window. The total computational LUT costs are roughly equivalent to the product of A and C.

3.3 Full Inference Accelerator Architecture: Dataflow and Multilayer Offload

FINN-*R* supports two key choices for the architecture of the accelerator. The first is a custom-tailored strictly feed-forward dataflow implementation as described in the original FINN paper [58] and illustrated in Figure 10(a). The second offloads a part of the dataflow pipeline, which represents a significant proportion of the compute load and allows the feature maps to iterate over it multiple times through a loopback path as is shown in Figure 10(b).

The customized *Dataflow Architecture (DF)* differentiates itself from many other accelerators in that it is customized for a specific NN topology and for different precisions in activations and weights for each individual layer avoiding "one-size-fits-all" inefficiencies and reap more of the benefits of reconfigurable computing. One streaming compute engine is instantiated per layer, with resources tailored to fit each layer's compute requirements and the user-defined frame rate. This is accomplished by adjusting their P, Q and M, as introduced in Section 3.2, according to the algorithm in Section 4.4. As the datafow architecture is fully rate balanced, each layer produces and consumes data in the same order with the aim of minimizing buffer requirements in between layers and latency. An engine starts to compute as soon as the previous engine starts to produce output and as such introduces another level of concurrency between layers.

The *Multilayer Offload Architecture (MO)* is beneficial when the minimal footprint of the fully unrolled DF architecture exceeds the target device capabilities or when the fragmentation overhead is unattractive. The main application context are large networks under hard resource constraints.

Comparing these two architectures, we observe the following: The cost of the DF implementation is the sum across all implemented layers, while the cost of the MO architecture is defined by the maximum across the scheduled layers and provides as such better scalability towards really deep CNNs. From a throughput point of view, which is dominated by the total amount of compute resources instantiated and their utilization, we expect both architectural choices to be equivalent. For DF, we experience a certain amount of fragmentation as will be explained in the next section, while for MO architectures, the utilization is determined by how well a layer can be scheduled onto the offloaded compute resources. However, we expect that the reduction of buffering between layers should bring significant latency benefits for DF vs MO, but this still remains to

be confirmed with experiments. Note that these two choices represent the two endpoints within a large design space of potential architectures that could be worth a more thorough investigation. For now, the chosen architectures provide the flexibility to build accelerators that scale both to extreme performance and to minimal footprints, even for very large networks.

3.4 Full Accelerator Cost Model

The full accelerator cost encompasses a constant part given by the formal *shell* resource overhead, and the dynamic aspect for the actual accelerator architecture, which is described above. Depending on the chosen target platform, a different base infrastructure or *shell* is created, which handles all memory interfaces, DMA engines, network connections and host interface. FINN-*R* currently supports a number of different platforms, whereby within the article we focus on PYNQ-Z1, AWS F1, and a Zynq UltraScale+ platform, called Ultra96. More detailed platform descriptions will be provided in Section 5. The shells are substantially different: Ultra96 and Pynq-Z1 move the images and the results through the coherently shared memory between ARM and FPGA fabric (PS memory), and, as memory controllers are hardened inside the SOC, the corresponding soft logic requirements are very small (8 BRAM18s, 2.6 kLUTs). For AWS F1, the SDK-based design entry is chosen, which moves the data between host and FPGA card via FPGA-attached DRAM and requires soft memory controllers as well as PCIe interface with DMA engine, all joined with an AXI interconnect. The overall overhead amounts to 1090 BRAM18s and 297 kLUTs. The total hardware cost for the different platforms can be computed as the sum of the specific shell overhead and the chosen accelerator architecture.

4 FINN-*R*

FINN-*R* is trying to answer the following question: Given a set of design constraints and a specific neural network, what is the best possible hardware implementation that can be achieved? For this, FINN-*R* provides insights and estimates and automates the customization of the hardware implementation. The tool is to be used interactively to explore a given CNN in terms of high-level concepts of the target platform, architecture, and precisions to achieve specific design goals and satisfy given constraints. Given the choices, the tool then customizes the hardware accelerator, either DF or MO style, to meet the constraints. We currently support resource footprint and throughput constraints. Latency, power estimation, and automated design-space exploration are left as future work.

The key functionality in the tool for the MO is the generation of the runtime schedule that sequences the compute onto the hardware engines, and for DF, the calculation of the folding factors that generate a balanced dataflow, whereby the whole architecture gets incrementally unfolded until design targets are met. As shown in Figure 4, FINN-*R* has a modular structure inspired by LLVM compiler infrastructure, with frontends, passes, and backends, and a quantization-aware intermediate representation (IR) of QNNs. The *frontend* is responsible for interfacing with a selection of training frameworks such as Caffe, DarkNet, and Tensorflow and translating trained QNNs into the IR. The IR is used as the basis of the performance estimation tool. FINN-*R* supports a number of *transformations* that help generate more efficient representations. Finally, the *backend* contains a code generator that creates executable inference accelerators for a selection of platforms, including PYNQ-Z1, Ultra96, and AWS F1. The accelerators are composed of components defined in the *QNN library*. All of these components are described in further detail in the following subsections.

4.1 Frontends

The frontend stage is responsible for converting QNNs trained by a variety of frameworks to the FINN-*R* intermediate representation. As each framework exposes their QNN topologies through

custom formats, FINN-*R* must first perform a conversion to a common intermediate representation, to process the network. Currently, we support frontends for BinaryNet [13], Darknet [52], and Tensorpack [71]. In the case of BinaryNet or Tensorpack, FINN-*R* examines the dimensions of these frameworks' network data stored in .npy files. Finally, for Darknet, the network topology is extracted from the configuration files (.cfg files) of the network. As FINN-*R* follows a modular design, additional frontends supporting new QNN frameworks can be added as they emerge.

Once converted to the intermediate representation, FINN-*R* is aware of the network topology and the precision of the data types within each layer. At this frontend stage, the representation is device agnostic; however, we can provide useful statistics, such as the operation count and the memory footprint of the weights and activations of each layer. In addition to loading topological information, FINN-*R* may optionally load a set of trained weights. These weights are reordered and forwarded to subsequent FINN-*R* stages for processing and inclusion in the final deployed network.

4.2 The Intermediate Representation

As is common practice [2, 54, 60], FINN-*R* represents a QNN as a directed acyclic graph. Its nodes represent layers and edges carry outputs from one layer to become inputs to another. The key differentiator of FINN-*R*'s intermediate representation (IR) is its quantization-awareness. Each node is tagged with the quantization of its inputs, parameters (weights), and outputs to enable quantization-aware optimizations (such as the streamlining optimization described below) and the mapping to backend primitives optimized for quantized computation. Internally, the IR differentiates *backend-neutral* and *backend-specific* layers. When a QNN is initially imported into FINN-*R*, its abstract computational structure is solely represented by backend-neutral layers. This representation is made backend-specific for a hardware implementation by a series of transform and analysis passes. In the derived graph, the network is decomposed into concrete hardware building blocks, such as the SWU an the MVU.

FINN-R Transform and Analysis Passes. FINN-*R* employs *passes*, i.e., small subprograms that operate on the IR. Each pass consumes an IR graph and may (a) *transform* the QNN to output a modified graph or (b) *analyze* the QNN to produce metadata about its properties, or it may do both. A pass may be composed of smaller passes to facilitate reuse and modularity. We highlight some of the key passes implemented in FINN-*R* below.

Direct Quantization. The first and last layers of QNNs are often quantization-sensitive and left using floating-point arithmetic [9, 50, 72]. However, a modest direct quantization as to 8-bit fixed-point quantities has little or no impact on accuracy [27] but already leads to significant resource savings. FINN-*R*'s direct quantization pass applies this transformation to non-quantized layers converting its parameters to fixed-point values of the specified bit precision. For quantizations below 8 bits, retraining is highly recommended but is not part of this pass.

Streamlining. The original FINN paper described how batch normalization parameters can be absorbed into thresholds via simple mathematical manipulation. This can be further generalized into a *streamlining pass* that absorbs floating-point scaling factors into multi-level thresholds as explored by Umuroglu and Jahre [59]. This is done by collapsing scaling layers in front of a quantization layer into a single linear transform that is then merged entirely into the quantization by updating its thresholds. The mathematically equivalent output QNN eliminates the storage and compute overhead of the subsumed floating-point scaling factors. Finally, as the maximum operator commutes with monotone functions such as the quantization used in QNNs, maxpool layers may be moved behind a quantization layer. This decreases the required precision of the comparators in the maxpool layer, resulting in further resource savings.

FPGA Resource Analysis. Chooses and scales hardware operators to optimize the network performance within a given resource budget. The corresponding algorithm is detailed in Section 4.4, and is integrated into FINN-*R* as an analysis pass.

FPGA Dataflow Architecture Generation. Generates a synthesizable DF implementation from the IR after the concurrency annotation by the resource analysis pass. Each layer is converted to an equivalent FPGA layer representing a building block of the QNN library parametrized to the determined parallelism. Finally, the corresponding HLS code is generated.

FPGA Multilayer Offload Schedule Generation. Generates an execution schedule targeting a given MO implementation, which is the sequence of layers as specified in the IR graph.

4.3 Backends

Backends are responsible for consuming the IR graph and backend-specific information to create a deployment package, and/or providing performance/resource estimates for two previously introduced hardware architectures (DF and MO). The deployment package consists of parameter data for the QNN model, and the backend-specific code that executes the model, consisting of both runtime environment as well as an executable hardware design for targeting dataflow and multilayer offload architectures, as well as a selection of predefined platforms.

4.4 Controlling Performance and Resource Utilization

FINN-*R* exploits the concurrency potential in a given QNN to generate a solution, which is scaled to utilize the committed resources optimally by tuning the previously introduced concurrency parameters P (PE duplication), Q (SIMD scaling), and M (multi-vector parallelization). All of them allow us to accelerate the computation of the respective layer whose throughput grows proportionally. For a feasible schedule, we choose Q as a factor of C, P as a factor of C', and M as a factor of N'. Finally, A represents the compute complexity of each layer and M its cumulative parallelism.

ALGORITHM 1: Data Flow Balancing by FINN-*R*

```
Input: A[0..L-1]
Data: M[0..L-1]
candidate := MO;                                   /* default to a multilayer offload */
M[0..L-1] := 1 ;                                   /* minimal dataflow compute */
/* Adopt dataflow with greater compute parallelism as long as feasible.          */
while feasible(M) do
    candidate := { DF, M };
    idx := max_index { A[.]/M[.] };
    M[idx] := next greater factor of C[i] · C'[i] · N'[i];
end
return candidate;
```

The minimal DF implementation chooses a scaling of 1 for all parameters and all layers. Its feasibility within the committed resources as determined by the cost functions of Section 3 decides whether a retreat to an MO architecture is necessary or the DF performance scaling can be pursued. The balanced scaling of the layers in a DF pipeline is the key capability of FINN-*R*. Having determined the compute requirements of all the layers, it systematically widens the most pressing bottlenecks as shown by Algorithm 1 until the resources are exhausted. The cumulative scaling factor determined for each layer is used to tile its computation into corresponding factors Q, P, and M.

Table 3. Platform Summary

Platform	Part #	Node	DDR	kLUTs	BRAMs
AWS F1	XCVU9P-FLGB2104-2-I	16nm	64GB	1,180	4,320
Ultra96	XCZU3EG-SBVA484-1	16nm	2GB	71	432
PYNQ-Z1 [63]	XC7Z020-1CLG400C	28nm	512MB	53.2	280

FINN-*R* estimates the performance of its generated implementation based on the chosen parallelism and the reported initialization interval of the building blocks. The layer compute time is evaluated as the quotient of its compute requirements and the attained concurrency. A throughput characterization is also performed for the other building blocks (SWU, MP) to identify when they become the bottleneck within the pipeline.

5 EVALUATION

We present and evaluate FINN-*R* on a range of platforms, with a selection of neural networks, for both dataflow and multilayer offload architectures, and compare it with state of the art.

5.1 Experimental Setup

A broad range of platforms have been targeted from embedded to datacenter scale. Their details are summarized in Table 3. PYNQ-Z1 is an open-source board and easily available, Ultra96 is a board based on the ultra96.org hardware specification using the newer Zynq Ultrascale+ architecture, and AWS F1 is a FaaS node in the public AWS cloud infrastructure. We considered four different types of neural network topologies to evaluate how well the performance prediction works for different computing patterns. The exact specifications can be found at GitHub [32] and descriptions are given in the following paragraphs.

MLP-4, CNV-6. These two topologies are identical to the ones used in FINN [58]. One is a classifying multilayer perceptron built from four large fully connected layers, and the other one implements a CIFAR-10, GTSRB, or SVHN classifier and is inspired by BinaryNet [13] and VGG-16 [55]. *Tincy YOLO* is a quantized adaptation of Tiny YOLO, which itself is a stripped-down version of YOLO directly provided by its authors [53]. Tiny YOLO is described completely in darknet [52]. Specifically, the differences between Tincy YOLO and Tiny YOLO are as follows: (1) the first and last convolution layers are converted to use 8-bit weights; (2) the other convolutional layers are converted to use 1-bit weights, 3-bit activations; (3) the activation function is removed from the last convolutional layer; (4) all other activation functions converted from leaky-ReLU to ReLU; (5) increase the size of the number of OFMs in the 2nd convolutional layer from 32 to 64; (6) a decrease in the number of OFMs of the 7th and 8th convolutional layers from 1024 to 512; (7) removal of the first maxpool layer; and (8) an increase of stride in the first convolutional layer from 1 to 2. The end result of these topological changes is a reduction in operations of 36%, while reducing the mean average precision (mAP) accuracy from 57.1% to 50.1%.

DoReFa-Net/PF. The DoReFa-Net/PF topology is based off the DoReFa-Net topology described by Zhou et al. [72]. DoReFa-Net itself, is a variant of AlexNet [30] with 1-bit weights, 2-bit activations used in all layers except the first and last, where floating-point weights are used. We further modify DoReFa-Net as follows: (1) the first and last layer are converted from floating-point weights to 8-bit weights; and (2) the first three layers are pruned by 30%, specifically, decreasing the number of OFMs in these layers from 96, 128 and 384 to 68, 90 and 272, respectively. The end result of the these topological changes is a reduction in operations of 56%, while improving the overall accuracy (Top-1) by 0.2%. For completeness, the Top-5 accuracies we were able to achieve with

Table 4. Dot-Product Workloads and Network Accuracies

	Bin. Ops/Frame	High Precision Ops/Frame	Total Ops/Frame	Parameters	Accuracy	Dataset
MLP-4	6.0 M [W^1A^1]	–	6.0 M	2.9 M	97.69 (Top-1 (%))	MNIST
MLP-4	–	6.0 M [$W^{FP}A^{FP}$]	6.0 M	2.9 M	98.74 (Top-1 (%))	MNIST
CNV-6	115.8 M [W^1A^1]	3.1 M [W^1A^8]	118.9 M	1.4 M	80.10 (Top-1 (%))	CIFAR-10
CNV-6	115.8 M [W^1A^1]	3.1 M [W^1A^8]	118.9 M	1.4 M	98.08 (Top-1 (%))	GTSRB
CNV-6	115.8 M [W^1A^1]	3.1 M [W^1A^8]	118.9 M	1.4 M	94.90 (Top-1 (%))	SVHN
Tincy YOLO	4385.9 M [W^1A^3]	59.0 M [W^8A^8]	4444.9 M	6.4 M	50.1 (mAP (%))	VOC 2007
Tiny YOLO	6780.5 M [W^1A^3]	194.9 M [W^8A^8]	6975.3 M	15.9 M	48.7 (mAP (%))	VOC 2007
Tiny YOLO	–	6975.3 M [$W^{FP}A^{FP}$]	6975.3 M	15.9 M	57.1 (mAP (%))	VOC 2007
DoReFa-Net/PF	2009.7 M [W^1A^2]	171.3 M [W^8A^8]	2181.0 M	60.2 M	50.3 (Top-1 (%))	ImageNet
DoReFa-Net	3626.3 M [W^1A^2]	250.1 M [$W^{FP}A^{FP}$]	3876.3 M	61.0 M	50.1 (Top-1 (%))	ImageNet
DoReFa-Net	–	3876.3 M [$W^{FP}A^{FP}$]	3876.3 M	61.0 M	55.9 (Top-1 (%))	ImageNet

DoReFa-Net/PF, DoReFa-Net [W^1A^2], and DoReFa-Net [$W^{FP}A^{FP}$] were 74.0%, 73.1%, and 77.4%, respectively.

In terms of training, a combination of the following techniques were used to train the networks described in Table 4: (1) the MLP-4 and CNV-6 networks were trained using the techniques described by Courbariaux et al. [13]; (2) DoReFa-Net/PF was trained using the techniques described by Zhou et al. [72], with the exception of the W^8A^8 layers, which are trained using the techniques described by Su et al. [56]; (3) the YOLO networks were trained using the methods described by Rastegari et al. [50] for the network weights, Zhou et al. [72] for the activations, and Su et al. [56] for the W^8A^8 layers. The pruning methods applied to realise Tincy YOLO and DoReFa-Net/PF are described by Faraone et al. [19].

The computational workloads of all these four QNN variants are indicated by Table 4. The stated figures sum up all the additions and multiplications within the dot products computed as an individual input frames passes through the convolutional layers and perceptrons of a topology. The very small MLP-4 network already asks for 6 million operations for each image. Being used for a digit recognition, which is purely based on shape, it is also the only network implementation that has been fully binarized starting from the very input.

All the other topologies feature layers of a higher 8-bit precision, particularly to interface naturally with their application scenarios. This precision allows to easily convey the color information of their visual inputs into their first layers. It is also the natural choice for reporting confidence levels of complex classification tasks. While the input and output layers, thus, pose intrinsic quantization bounds, the hidden layers can often be quantized significantly while practically maintaining accuracy. As shown in Table 4, the amount of computation that could be reduced this way is enormous and accounts for the vast majority of the compute in these networks.

For reference, we've also provided the accuracy and workloads of unpruned and/or unquantized versions of these networks if they are available. No floating-point results are provided for the CNV-6 topology, as this network was designed specifically as a BNN by Umuroglu et al. [58]. With regard to accuracy, we note that the difference in accuracy between MLP-6, Tincy YOLO and DoReFa-Net/PF and their unquantized, unpruned equivalents is 1.05%, 7.0%, and 5.6%, respectively. Whether these accuracies are still acceptable is dependent on the target application.

5.2 Measured Results and Evaluation

Concrete implementations of the four described networks on the introduced platforms were dimensioned, synthesized and measured. All results are summarized in Table 5 together with prior

Table 5. FINN-*R* Implementations and Prior Work

CNN	Platform	Clock (MHz)	BRAM18	LUTs	Perf. (predicted) (GOp/s)(%)	(Power) (W)	Efficiency (GOp/s/W)	Precision (%)
FINN-*R* Results								
FINN-*R* MLP-4	**AWS F1** (DF)	232.9	1,652	337,753	50,776 (79%)	-	-	W^1A^1
FINN-*R* MLP-4	**Ultra96** (DF)	300	417	38,205	5,110 (75%)	11.8	433	W^1A^1
FINN-*R* MLP-4	**PYNQ-Z1** (DF)	100	220	25,358	974 (99%)	2.5	390	W^1A^1
FINN-*R* CNV-6	**AWS F1** (DF)	237	1,888	332,637	12,109 (98%)	-	-	W^1A^1
FINN-*R* CNV-6	**Ultra96** (DF)	300	283	41,733	2,318 (99%)	10.7	217	W^1A^1
FINN-*R* CNV-6	**PYNQ-Z1** (DF)	100	242	25,770	341 (99%)	2.25	152	W^1A^1
FINN-*R* Tincy YOLO	**AWS F1** (DF)	249.8	2,638	242,457	5,271 (93%)	-	-	W^1A^3
FINN-*R* Tincy YOLO	**Ultra96** (MO)	220	316	40,808	288 (68%)	9.7	30	W^1A^3
FINN-*R* Tincy YOLO	**PYNQ-Z1** (MO)	100	280	46,507	133 (66%)	2.5	53	W^1A^3
FINN-*R* DoReFa-Net/PF	**AWS F1** (DF)	155.8	1,332	476,970	11,431 (92%)	-	-	W^1A^2
FINN-*R* DoReFa-Net/PF	**Ultra96** (MO)	220	432	36,249	400 (70%)	10.2	39	W^1A^2
FINN-*R* DoReFa-Net/PF	**PYNQ-Z1** (MO)	100	278	35,657	129 (48%)	2.5	51	W^1A^2
Related Work - Reduced Precision								
FINN MLP-4 [58]	**ZC706**	200	396	82,988	9,086(-)	22.6	402	W^1A^1
FINN CNV-6 [58]	**ZC706**	200	186	46,253	2,466(-)	11.7	210	W^1A^1
FINN CNV-6	**ADM-PCIE-8K5**	125	1,659	365,963	14,814(-)	~41	361	W^1A^1
Alemdar et al. [4] CNV-X	**Kintex-7 160T**	-	-	-	~878(-)	-	-	W^2A^2
Prost-Boucle et al. [49] CNV-6	**VC709**				~3, 029(-)	6.64*	-	W^2A^2
Park and Sung [47] MLP-X	**ZC706**	172	-	-	~210(-)	~5	42	W^3A^8
Nakahara et al. [42] CNV-X	**Zedboard**	144	32†	18,325†	329(-)	2.3	143	W^1A^1
Yonekawa et al. [64] CNV-X	**ZCU102**	150	1,367†	~22, 095†	461(-)	22	21	W^1A^1
Zhao et al. [70] CNV-X	**Zedboard**	143	94†	~46, 900†	207.8(-)	4.7	44	W^1A^1
Jiao et al. [26] DoReFa-Net	**Zynq 7020**	200	106†	~44, 000†	410.2(-)	2.3	182	W^1A^2
Liang et al. [34] CNV-X	**MPC-X2000 DFE**	150	-	-	9,396(-)	26.2	359	$W^{1-8}A^{1-8}$
Related Work - Higher Precision	* no full board power, only as measured by PMBUS available rails; † includes shell overhead							
Wei et al. [61] AlexNet	**AWS F1**	230	1,682	545,000	1,884.2(-)	-	-	W^8A^{16}
Wei et al. [61] VGG16	**AWS F1**	240	1,690	504,000	2,260.9(-)	-	-	W^8A^{16}
Wei et al. [61] ResNet	**AWS F1**	230	1,644	496,000	1,875.3(-)	-	-	W^8A^{16}
Zhang et al. [67] VGG16	**VC709**	150	-	-	354(-)	26	14	$W^{16}A^{16}$
Aydonat et al. [6] AlexNet	**GX1150**	303	-	-	1,382(-)	~45	31	$W^{HP}A^{HP}$
Zhang et al. [69] VGG16	**GX1150**	385	-	-	1,790(-)	~37.46	48	$W^{16}A^{16}$
Hedge et al. [23] Caltech101	**ZC706**	180	-	-	11.5(-)	19	<1	$W^{16}A^{16}$
Ma et al. [38] VGG16	**GX1150**	150	-	-	645.25(-)	-	-	W^8A^{16}
Ma et al. [37] NiN	**GX1150**	200	-	-	588(-)	-	-	$W^{16}A^{16}$
Ma et al. [37] VGG-16	**GX1150**	200	-	-	720(-)	-	-	$W^{16}A^{16}$
Ma et al. [37] ResNet-50	**GX1150**	200	-	-	619(-)	-	-	$W^{16}A^{16}$
Ma et al. [37] ResNet-152	**GX1150**	200	-	-	710(-)	-	-	$W^{16}A^{16}$

Metrics not reported by prior work are indicated by -. Estimated values are denoted as ~.

art. The datapoints targeting PYNQ-Z1 can be reproduced by using the open-source releases [31, 33]. We constrain our comparison to only measured results and exclude extrapolated numbers ([39, 41, 44, 45]). For undisclosed CNN and MLP topologies in related work, we use the term CNV-X and MLP-X, respectively. Power and efficiency are measured in respect to board power. HP and FP

stand for 16-bit and 32-bit (half precision and full precision) floating point, respectively. Resources are only given for BRAM18s and LUTs on Xilinx devices where available. URAM is excluded as many of the utilized devices do not support them, and optimal selection of LUTRAM, BRAM, and URAM through the tool flow is not yet controlled. DSP counts are not relevant for the extreme QNNs apart from the 8 bit precision layers. All implementations are compared in regards to performance, power, efficiency (=performance/power) for a specific network on a specific platform. Where possible, we provide details on resource usage, and data types used.

There are a number of key observations that can be taken from the table: First, as expected, there is a direct correlation between precision and achievable peak performance and energy efficiency, clearly demonstrating the value of QNNs. Wei et al. [61] and a recent Xilinx demonstration at SC'2017 clearly show close to possible peak performance for higher precision implementations on AWS F1. These solutions are bottlenecked by DSP availability. We show here that QNNs can provide further performance scalability by leveraging the vast amount of LUTs. This can be easily seen when comparing peak performance and efficiency on the same platform as for example AWS F1 across a spectrum of precisions. Second, CNNs compared to MLPs achieve on the same platform lower peak performance. This is because the actual compute in form of multiply accumulate has a lower percentage. As shown with the cost functions, CNNs contain also SWUs and MPs and the overhead has clearly an impact on overall performance. Compared to the original FINN, we demonstrate that the performance can scale with the size of the FPGA. Many new network topologies and platforms and different precision implementations were added.

The best performance numbers are achieved for a fully binarized MLP on AWS F1, at currently 50TOp/s and 12TOp/s for a CNN. Note that these are initial results with much yet to gain by investing into timing closure for instance. The best efficiency can only be reported for the embedded platforms as for other platforms board-level power measurements are not available. The highest value was achieved for FINN-*R* on the MLP on the Ultra96 platform with 433GOp/s/W.

While DorefaNet is no longer state-of-the art in regards to accuracy, the network is still representative and its implementation demonstrates that image classification with large networks on large input sizes can be highly performant on FPGA-based implementations leveraging reduced precision. The implementation defaulted to 155.8MHz and achieved with that a measured 11.43 TOp/s performance. Timing closure at 300MHz would immediately translate into a performance of 22 TOp/s as the implementation is not memory bound. Furthermore, there is still scope to scale spatially.

Compared to other reduced precision implementations on Virtex and Kintex platforms, we are higher performing on the same network compared to Prost et al. [49] on larger platforms (efficiency cannot be compared as not the full board power is taken into consideration). We outperform Alemdar et al. [4] by over an order of magnitude. Another highlight amongst related work is shown by the FP-BNN project from Liang et al. [34], which reports up to 9.4 TOp/s overall performance on a server class platform across a selection of networks with mixed precision. In regards to ZC706, our solution provides a speed-up of 10× and higher on the basis of dataflow-based architectures. ZedBoard data points are best compared to Pynq-Z1, as they feature the same device. In comparison to our dataflow architecture, Nakahara et al. [42] and Zhao et al. [70] are up to 2× slower. The binary network accelerator by Yonekawa et al. [64] is hard to compare as it was done on a ZCU102, where the FPGA device is around 4× larger than the same device on the Ultra96. Even with the significantly smaller device, the CNV-based FINN-*R* implementation is 4× faster. Finally, Jiao et al. [26] report excellent performance for a large network on a small device with 410.2GOp/s. Due to the large size of this network, FINN-*R* cannot use a DF design, and MO achieves 129GOp/s. Overall, we believe that FINN-*R* delivers highly competitive performance and efficiency, with room for further improvements.

In regards to the framework, the broad spectrum of networks, platforms and implementations demonstrates the flexibility that the tool can provide through automated scaling. Furthermore, the performance estimations for DF are 75%–99% accurate and as such a good indication of achievable performance without having to go through a full implementation flow. Also note that the prediction is around the 50% mark of the original AWS F1 roofline, and as such more accurate. The performance estimations for the MO still differ from actual implementations. This is due to the more complex nature of the system, which involves significantly more external memory transfers for pulling weights and FM, which latencies are currently estimated using a peak-bandwidth model. In the case of DoReFa-Net with MO on PYNQ-Z1, the measured performances are largerly lower than expected (48%). This is due to memory re-organization performed in the host whenever a split layer is present to concatenate feature maps for the following layer. The execution time scales with number of available threads on the host. This non-ideal behaviour is overcome in the Ultra96 case, which features four cores against the two available on PYNQ-Z1.

6 CONCLUSIONS AND OUTLOOK

Quantization and reduced precision representations for inputs, activations, and weights provide a promising approach to scalability in performance, power efficiency, and storage footprint for CNNs. They create interesting design trade-offs in exchange for a small reduction in accuracy. Especially in conjunction with FPGAs, a broad spectrum of architectures for inference engines can be explored whereby the minimal precision to achieve precisely the numerical accuracy required for a particular application can be exploited.

In this article, we described FINN-*R*, an end-to-end framework that automates the exploration of customized hardware accelerators. FINN-*R* is the second generation of the FINN tool, extending the original with support for arbitrary precision and more flexibility in the end architecture and target platforms, including hardware cost estimatation for given devices. We evaluated the generated architectures on four different reduced precision neural networks, from small CIFAR-10 classifiers to a YOLO-based object detection on PASCAL VOC datasets on a range of platforms including PYNQ and AWS F1. The generated design end-points showcase the promise of QNNs, demonstrating 5TOp/s on embedded platforms as well as 50TOp/s on datacenter platforms. The broad selection of measured results validates the workflow and the flexibility of the framework.

The spectrum of possible design-space tradeoffs is vast, and while we hope to shed some light on what is possible, many variants remain unexplored. We are currently in the process of open-sourcing the framework, such that the broader community can help investigate the design space. The networks themselves as well as a subset of implementations can already be found on github [32]. From a research point of view, we are focusing on tightening up the performance and resource predictions, adding support for more networks, such as the promising residual networks and LSTM, and providing more automation within the framework itself, while a separate strand of research is working on improving accuracy through novel training techniques and quantization methods with different numerical representations.

REFERENCES

[1] ImageNet Large Scale Visual Recognition Challenge (ILSVRC). 2017. Retrieved from http://image-net.org/challenges/talks_2017/ILSVRC2017_overview.pdf.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR* abs/1603.04467.

[3] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry. 2017. Tactics to directly map CNN graphs on embedded FPGAs. *IEEE Embed. Syst. Lett.* (2017).

[4] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot. 2016. Ternary neural networks for resource-efficient AI applications. *CoRR* abs/1609.00222.

[5] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2016. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In *Proceedings of the ISVLSI*. IEEE, 236–241.
[6] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. Chiu. 2017. An OpenCL (TM) deep-learning accelerator on Arria 10. *CoRR* abs/1701.03534.
[7] C. Baskin, N. Liss, A. Mendelson, and E. Zheltonozhskii. 2017. Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. *arXiv preprint arXiv:1708.00052*.
[8] Doug Burger. 2017. Microsoft Unveils Project Brainwave for Real-Rime AI. Retrieved from https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.
[9] Z. Cai, X. He, J. Sun, and N. Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.
[10] K. Chellapilla, S. Puri, and P. Simard. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.
[11] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam. 2016. DianNao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.
[12] Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the ISCA*. IEEE, 367–379.
[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training neural networks with weights and activations constrained to +1 or −1. *CoRR* abs/1602.0 (2016).
[14] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*, Vol. 1. MIT Press, 1269–1277. http://dl.acm.org/citation.cfm?id=2968826.2968968.
[15] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, McKinstry, Timothy Melano, Davis R. Barch, Carmelo Di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proc. Natl. Acad. Sci.* 113, 41 (2016), 11441–11446. http://www.pnas.org/content/113/41/11441.
[16] Benoit Jacob et al. 2017. gemmlowp: A Small Self-Contained Low-Precision GEMM Library. Retrieved from https://github.com/google/gemmlowp.
[17] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. 2009. CNP: An FPGA-based processor for convolutional networks. In *Proceedings of the IEEE FPL*. IEEE, 32–37.
[18] J. Faraone, N. Fraser, G. Gambardella, P. H. W. Blott, and M. Leong. 2017. Compressing low precision deep neural networks using sparsity-induced regularization in ternary networks. In *Proceedings of the ICONIP*. Springer, 393–404.
[19] Julian Faraone, Giulio Gambardella, David Boland, Nicholas J. Fraser, Michaela Blott, and Philip H. W. Leong. 2018. Customizing low-precision deep neural networks For FPGAs.
[20] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. 2017. Scaling binarized neural networks on reconfigurable logic. In *Proceedings of the PARMA-DITAM*. 6. Retrieved from https://doi.org/10.1145/3029580.3029586.
[21] S. Han, H. Mao, and W. J. Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR* abs/1510.00149 (2015).
[22] S. Han, J. Pool, J. Tran, and W. J. Dally. 2015. Learning both weights and connections for efficient neural networks. *CoRR* abs/1506.02626 (2015).
[23] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre. 2016. CaffePresso: An optimized library for deep learning on embedded accelerator-based platforms. In *Proceedings of the CASES*.
[24] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *Proceedings of the ISSCC*. IEEE, 10–14.
[25] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and < 1 MB model size. *CoRR* abs/1602.07630 (2016).
[26] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. 2017. Accelerating low bit-width convolutional neural networks with embedded FPGA. In *Proceedings of the FPL*. IEEE, 1–4.
[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ISCA*. ACM, 1–12.
[28] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the MICRO*. IEEE, 1–12.
[29] Minje Kim and Paris Smaragdis. 2016. Bitwise neural networks. *CoRR* abs/1601.0 (2016).
[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the NIPS*. 1097–1105.
[31] Xilinx Research Labs. 2017. BNN-PYNQ. Retrieved from https://github.com/Xilinx/BNN-PYNQ.
[32] Xilinx Research Labs. 2017. FINN-R. Retrieved from https://github.com/XilinxDublinLabs/FINN-R.

[33] Xilinx Research Labs. 2018. QNN-MO-PYNQ. Retrieved from https://github.com/Xilinx/QNN-MO-PYNQ.
[34] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* 275 (2018), 1072–1086. http://www.sciencedirect.com/science/article/pii/S0925231217315655.
[35] ARM Limited. 2017. Compute Library. Retrieved from https://developer.arm.com/technologies/compute-library.
[36] B. Liu, M. Wang, H. Foroosh, M. F. Tappen, and M. Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*. 806–814. Retrieved from https://doi.org/10.1109/CVPR.2015.7298681
[37] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the FPL*. IEEE, 1–8.
[38] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the FPGA 2017*. ACM, 45–54.
[39] A. K. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr. 2017. WRPN: Wide reduced-precision networks. *CoRR* abs/1709.01134.
[40] J. Misra and I. Saha. 2010. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing* 74, 1–3 (2010), 239–255.
[41] D. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. Leong. 2017. High-performance binary neural networks on the Xeon+ FPGA platform. In *Proceedings of the FPL*. IEEE.
[42] H. Nakahara, T. Fujii, and S. Sato. 2017. A fully connected layer elimination for a binarized convolutional neural network on an FPGA. In *Proceedings of the FPL*. IEEE, 1–4.
[43] H. Nakahara, H. Yonekawa, T. Fujii, M. Shimoda, and S. Sato. 2017. A demonstration of the GUINNESS: A GUI -based neural network synthesizer for an FPGA. In *Proceedings of the FPL*. IEEE, 1–1.
[44] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. 2016. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the FPT*. 77–84.
[45] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, et al. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the FPGA*. ACM.
[46] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/CNN20Whitepaper.pdf.
[47] Jinhwan Park and Wonyong Sung. 2016. FPGA-based implementation of deep neural networks using on-chip memory only. In *Proceedings of the ICASSP*. IEEE, 1011–1015.
[48] Th. B. Preußer. 2017. Generic and universal parallel matrix summation with a flexible compression goal for xilinx FPGAs. In *Proceedings of the FPL*.
[49] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *Proceedings of the FPL*. IEEE.
[50] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. *CoRR* abs/1603.05279 (2016).
[51] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. 2016. Minerva: Enabling low-power, highly accurate deep neural network accelerators. In *Proceedings of the ISCA*. IEEE Press.
[52] J. Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. Retrieved from http://pjreddie.com/darknet/.
[53] J. Redmon and A. Farhadi. 2017. YOLO9000: Better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. 6517–6525.
[54] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh. 2016. DnnWeaver: From high-level deep network models to FPGA acceleration. In *Proceedings of the Workshop on Cognitive Architectures*.
[55] K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556.
[56] Jiang Su, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Gianluca Durelli, David B. Thomas, Philip H. W. Leong, and Peter Y. K. Cheung. 2018. Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic. In *Proceedings of the ARC*. ACM, to Appear.
[57] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. 2015. Resiliency of deep neural networks under quantization. abs/1511.0.
[58] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the FPGA*. ACM.
[59] Y. Umuroglu and M. Jahre. 2017. Streamlined deployment for quantized neural networks. arXiv preprint arXiv:1709.04060.

[60] S. I. Venieris and C. Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *Proceedings of the CCM*. IEEE, 40–47.
[61] X. Wei, Peng Yu, C. H. and, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the DAC*. ACM, 29.
[62] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
[63] Xilinx, Inc. 2017. *Zynq-7000 All Programmable SoC Data Sheet: Overview*. Xilinx, Inc.
[64] H. Yonekawa and H. Nakahara. 2017. On-chip memory-based binarized convolutional deep neural network applying batch normalization free technique on an FPGA. In *Proceedings of the IPDPSW*. IEEE, 98–105.
[65] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the ISCA*. ACM, 548–560.
[66] S. Zagoruyko and N. Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146*.
[67] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the ICCAD*. IEEE.
[68] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the FPGA*. ACM.
[69] J. Zhang and J. Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the FPGA*. 25–34.
[70] R. Zhao, W. Song, W. Zhang, T. Xing, J. Lin, M. Srivastava, R. Gupta, and Z. Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the FPGA*.
[71] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. 2017. Incremental network quantization: Towards lossless CNNs with low-precision weights. *CoRR* abs/1702.03044. Retrieved from http://arxiv.org/abs/1702.03044.
[72] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR* abs/1606.06160.
[73] C. Zhu, S. Han, H. Mao, and W. J. Dally. 2016. Trained ternary quantization. *CoRR* abs/1612.01064.

Received December 2017; revised April 2018; accepted July 2018