High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression

Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot

High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression

ADRIEN PROST-BOUCLE, ALBAN BOURGE, and FRÉDÉRIC PÉTROT, Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, France

Although performing inference with artificial neural networks (ANN) was until quite recently considered as essentially compute intensive, the emergence of deep neural networks coupled with the evolution of the integration technology transformed inference into a memory bound problem. This ascertainment being established, many works have lately focused on minimizing memory accesses, either by enforcing and exploiting sparsity on weights or by using few bits for representing activations and weights, so as to be able to use ANNs inference in embedded devices. In this work, we detail an architecture dedicated to inference using ternary $\{-1, 0, 1\}$ weights and activations. This architecture is configurable at design time to provide throughput vs power trade-offs to choose from. It is also generic in the sense that it uses information drawn for the target technologies (memory geometries and cost, number of available cuts, etc) to adapt at best to the FPGA resources. This allows to achieve up to $5.2k$ fps per Watt for classification on a VC709 board using approximately half of the resources of the FPGA.

Additional Key Words and Phrases: Ternary CNN, low power inference, hardware acceleration, FPGA

ACM Reference Format:
Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot. 2018. High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression. *ACM Trans. Reconfig. Technol. Syst.* 11, 3, Article 15 (December 2018), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rebirth of artificial neural networks a decade ago, as analyzed in [32], has triggered a lot of works in the hardware acceleration of inference. As compared to previous generation of artificial neural networks (ANN), deep learning approaches, with the adequate structure and training, perform extremely well in terms of accuracy for many applications. This however comes at the cost of ANNs with tens of layers, thousands of neurons and tens to hundreds of millions of weights, such as, for example, VGG [28] or Inception v4 [31]. Using classical software implementation techniques, inference requires a lot of floating point computations. Nowadays, practical general purpose implementations of inference make use of some kind of hardware accelerator, typically a GPU that provides 16-bit floating point arithmetic, an FPGA working on 8 to 16-bit integers, or a specific DSP based hardware. Paraphrasing Ken Batcher famous quote: “A supercomputer is a device for turning compute-bound problems into I/O-bound problems.” [3], we can say that a modern ANN accelerator is a device for turning compute-bound problems into memory-bound problems. Indeed, according to [25], for a given 45nm technology, the energy per operation of an off-chip DRAM access

*Institute of Engineering Univ. Grenoble Alpes

This project is being funded in part by Grenoble Alpes Métropole through the Nano2017 Esprit project.
Authors' address: Adrien Prost-Boucle; Alban Bourge; Frédéric Pétrot, Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, Grenoble, 38000, France, <firstname>.<lastname>@univ-grenoble-alpes.fr.

is 3500× higher than that of a 16-bit addition, and around 60 to 80 times higher than an on-chip SRAM access. Concerning FPGA implementations, the off-chip DRAM vs on-chip SRAM power ratio should be of the same order of magnitude, while bandwidth, thanks to the numerous available memory cuts, should be much higher for SRAM than for DRAM. Given these numbers, it is clear that the main challenge is not the computations to be performed by the neurons, but the access to the pre-trained network weights from external memory. Common computing technologies usually used for desktop or cloud computing are unfortunately not suitable for embedded or IoT devices, where the demand for intelligent sensors is growing. These applications represent a wide range of uses, for which ANN accelerators are needed to reach low-power and low-cost objectives [5].

To limit power consumption, researchers have focused mainly on three alternative paths: (i) exploiting the sparsity of weights [12], (ii) limiting the number of weight values [7], or (iii) limiting the number of bits of the weights and activations [9]. Our approach falls in the third category, by using weights and activations that are balanced ternary values $\{-1, 0, 1\}$[†]. This choice comes from recent advances in training with ternary weights. TWN [17] and TTQ [38] produce a classification accuracy very close to the 32-bit floating point one on reference benchmarks, however using activations with larger bit width (4 bits and above). TNN [1] trains networks with ternary weights *and* activations, for all but the input layer, and still reaches good accuracy compared to the state of the art. The immense advantage of using binary or ternary values is that, even for already quite large networks, all weights can fit within on-chip memory. This makes the memory bound problem less of a problem, the advantage of ternary over binary being that it achieves much higher accuracy. The up to 2× increase in weight memory is clearly at its disadvantage, but allows to trade accuracy versus area depending on the application.

In this paper, we present a neural network architecture making use of ternary weights and activations. The architecture is configurable at design time, so that by choosing the appropriate degree of parallelism of some key neuron layers, different throughput values can be achieved. This in turn increases power because more memories have to be accessed at the same time, leading to power versus throughput trade-offs to be decided upon depending on the application. This work is an extension of a previous work [26] in which we presented an initial TNN implementation. In addition to describing the architecture with much greater details, we now also introduce fine-grained optimization for ternary adder trees and our strategy to optimize memory usage. Indeed, memory handling being one of the main architectural challenge, the design construction relies on information about the size, geometry and granularity of the different memory primitives. This enables to assemble at best the memory of weights required by each layer.

In addition, as the naive implementation for each ternary weight uses 2 bits, we compress the weights so that we spare 16% or 20% of the memory compared to the naive implementation, at the cost of some logic necessary to compress and decompress. For comparison, the highest theoretical saving is $\approx 20.75\%$ (this comes from the Shannon limit of $\log_2(3) \approx 1.585$ bits of information per ternary digit – also called *trit*).

The main contributions of this work are thus:

- an optimized flexible architecture that takes benefit from the FPGA characteristics,
- a compression/decompression engine to mitigate memory usage by encoding a sequence of trits as bits,
- a comprehensive study of implementations with different degrees of parallelism and weight compression usage.

[†]"Perhaps the prettiest number system of all is the balanced ternary notation", according to Don Knuth [13]

2 RELATED WORKS

We do not focus on the training matters in this paper, as the theory behind it is clearly not related to architecture or integration technology. We shall however keep in mind when comparing architectures and designs that the accuracy of the results has to be taken into account, as does area, power, and throughput. Therefore we shortly introduce some results regarding ANN with highly quantized weights and activations. Accuracy *vs* hardware cost for fixed topologies is studied in [35], which concludes that 1-bit, 2-bit and 4-bit quantizations provide Pareto optimal solutions. This work also points-out that reaching high accuracy levels for highly quantized weights (and possibly activations) requires specific training procedures and potentially an increase in the number of neurons.

Fortunately, more theoretically inclined researchers have worked on binary and ternary ANN. Training with binary weights was first considered in [4], and then later by, among others, [8], that achieves, according to the authors, results close to what the original full-precision network reaches. The binarization of the activations makes the process more challenging, and solutions such as [27] obtain image classification results in the order of 10% under the state of the art.

To get higher accuracy while being resource-efficient, a training methodology for balanced ternary weights and activation bit-width between 1 and 8 bits has been proposed in [10]. Using MNIST and TIMIT datasets, they show that the accuracy drop due to quantization is low and that it decreases with the number of neurons. More recent works use ternary weights but keep activations with 4 bits and above, such as TWN [17] and TTQ [38]. They produce a classification accuracy very close to the reference floating-point training on MNIST [16] and CIFAR-10 [14] benchmarks. In this work, we use the approach detailed in [1] for training using ternary weights and activations. It is based on a two-stage teacher-student approach in which the teacher network is trained with stochastically firing ternary neurons, and the student network learns how to imitate the teacher's behavior using a layer-wise greedy algorithm. The student network's weights are ternarized versions of the teacher network's weights, while each neuron output is ternarized using two local thresholds.

Table 1 shows the results obtained on image classification benchmarks using the VGG-like architecture of [4] for its ternary and floating points implementations. This architecture, depicted in Figure 1 and detailed in Section 3.1, is often used in works on highly quantized ANN. Table 2 gives the quantization and accuracy for some actual realizations of binary and ternary NN, as given in the literature. Please note that training approaches for networks using values coded on few bits is a rapidly evolving research area, and therefore the accuracies reported here are the snapshot at the time of the articles' writing.

Table 1. Ternarization Performance - Error Rates (%) [1]

	CIFAR-10 [14]	SVHN [22]	GTRSB [30]
NN-64			
Float	13.11	2.40	0.96
Ternary	13.29	2.40	1.05
NN-128			
Float	10.48	2.27	0.76
Ternary	10.61	2.30	0.80

We now focus on the main challenges behind low-power, high-performance and high-accuracy FPGA implementations: architectural impact of data quantization and strategies to limit the number of accesses to external memory to fetch the neurons weights.

Table 2. Comparison with related works

Dataset	Authors	Plat. name	NN Arch.	Input quant.	Weight quant.	%Error
CIFAR-10	**This work**		NN-64	3 ch, 8 bits	2 bits	**13.29**
	This work		NN-128	3 ch, 8 bits	2 bits	**10.61**
	[34]	FINN	NN-64	24 bits	1 bit	19.90
	[18]	BCNN	NN-128	3 ch, 6 bits	1 bit	12.20
	[37]	BNN	NN-128	3 ch, 20 bits	1 bit	11.32
	[1]	TNN	NN-128	12 ch, 2 bits	2 bits	12.11
SVHN	**This work**		NN-64	3 ch, 8 bits	2 bits	**2.40**
	This work		NN-128	3 ch, 8 bits	2 bits	**2.30**
	[34]	FINN	NN-64	24 bits	1 bit	5.10
	[1]	TNN	NN-64	12 ch, 2 bits	2 bits	2.73
GTSRB	**This work**		NN-64	3 ch, 8 bits	2 bits	**1.05**
	This work		NN-128	3 ch, 8 bits	2 bits	**0.80**
	[1]	TNN	NN-128	12 ch, 2 bits	2 bits	0.98

A binary weight approach for CNN has been proposed in [6], later expanded to FINN [33]. Their goal is to target the highest reported throughput of a state-of-the-art network on a single FPGA chip, goal they currently achieve with their implementation on board ZC706. Their design, which is essentially a large pipeline accessing many bits of memory at once, can classify the datasets CIFAR10 and SVHN [22] at a throughput of 21 900 fps. This is a high throughput, but the training applied to FINN for this experiment allowed to reach only a limited accuracy.

The memory bottleneck issue is addressed in [19]. Activations and weights have high quantization (up to 32 bits) and all neuron weights are stored in off-chip DDR memory. They perform design space exploration with caching, scheduling and data arrangement techniques to achieve the highest throughput possible. Power efficiency reaches 18 GOP/s/W, which is high for this kind of workload, however this is very low compared to what binary or ternary networks achieve.

The work of [24] has identified that using only the FPGA internal memory could have an interest. Their NN are relatively simple MLPs on the MNIST benchmark, using 3-bit weights. Their overall architecture is very *ad-hoc* and, given the limited resources of their FPGA, they use a parallel/serial approach. They achieve 49k classifications per second at 100MHz.

The works of [36] focus on getting the maximum throughput from hard DSP cores of the Xilinx FPGAs. Using up to 24-bit precision and exploiting DSP cores at the maximum of their possible internal frequency (around 740 and 890 MHz depending on FPGA technology), they achieve unprecedented resource efficiency for this kind of workload. However they did not study the impact on power and did not address the memory bottleneck challenge.

The capability of Intel Arria10 FPGAs is studied in [23]. This FPGA technology, not released at that time, features hard DSP cores that are capable of 32-bit floating-point (FP32) operations. Their focus is then on performance and power figures compared to GPUs, when using FP32 activations with FP32 or ternary weights. They conclude that these FPGAs will provide better performance and energy efficiency than high-end GPUs.

Binary neural networks on the Intel Xeon+FPGA platform have been experimented in [20]. It is demonstrated that the integrated FPGA provides slightly better performance than a high-end discrete GPU. Competition between FPGA and GPU is fierce and new state of the art results require low-level implementation optimizations.

To the best of our knowledge, there is no previous work on the FPGA implementation of fully ternary CNN (*i.e.* where both weights and activations are ternary), but our own previous work presented in [26]. The current paper contains multiple improvements over our first architecture and presents a thorough analysis of different trade-offs, and comprehensive area and power results.

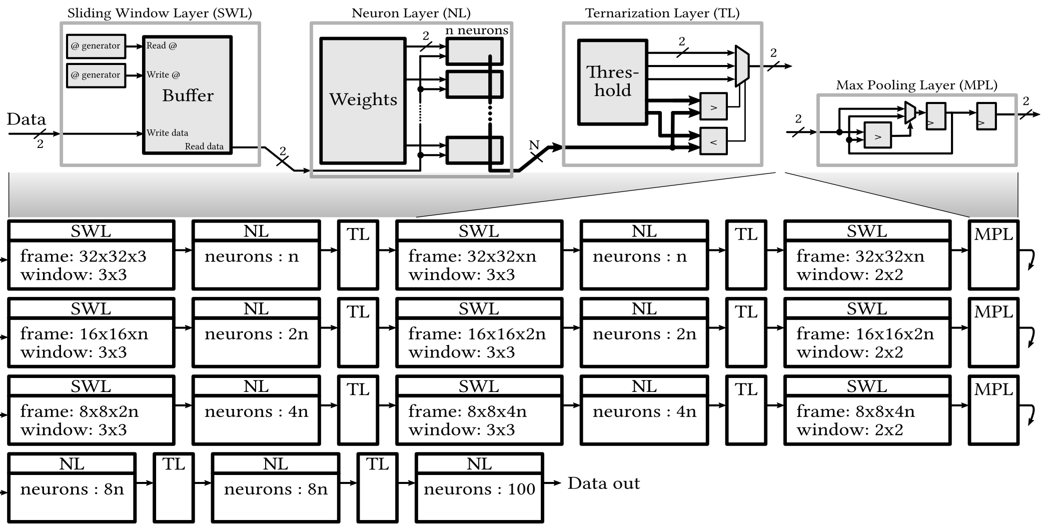


Fig. 1. CNN architecture overview

3 NEURAL NETWORK ARCHITECTURE

3.1 Overview

The ANN architecture we use for inference is a large-scale Convolutional Neural Network (CNN) pipeline similar to the VGG-like one introduced in [4]. The architecture of the ternary CNN is the following, in which we ternarized the weights and activations for the purpose of our design and implementation:

$$(2{\times}nCV_{3\times3}) - MP_{2\times2} - (2{\times}2nCV_{3\times3}) - MP_{2\times2} - (2{\times}4nCV_{3\times3}) - MP_{2\times2} - (2{\times}8nFC) - 100FC \quad (1)$$

where $mCV_{3\times3}$ represents a Convolution Layer (*CVL*) with m neurons, window size 3×3, step 1 and one pixel of padding at zero, $(2 \times mCV_{3\times3})$ is a pair of $mCV_{3\times3}$ layers in series, $MP_{2\times2}$ is max-pooling with window size 2×2, step 2 and no padding, and mFC is a fully-connected neuron layer with m neurons.

Figure 1 depicts the different layers that compose the VGG-like pipeline and the way they are connected. All layers are independent from each other: they have their own configuration registers and state machine, and image data is streamed between the stages through FIFO channels. For each layer type, we design a hardware block (implemented as hand-written VHDL with generics) that is reused all along the pipeline with different generic parameters. One can see the top of Figure 1 for a simplified schematic view of the implementation of each block type. Four main layer types are used: Sliding Window Layer (*SWL*), Neuron Layer (*NL*), Ternarization Layer (*TL*) and Max Pooling Layer (*MPL*). The *TL* exists because of the constraints introduced by the ternary activations: the result of a neuron is scalar and it has to be ternarized before being sent as input of the next *NL*. The pipeline begins with two *CVL*. A *CVL* is comprised of an *SWL*, an *NL* and a *TL*. These two *CVL* are followed by an *MPL*, two more *CVL*, another *MPL*, again two *CVL* and an *MPL*. It ends with three fully-connected *NL*. When functioning at full speed, meaning no input data starvation, the pipeline contains approximately the activations corresponding to two images. The input images are 32×32 pixels with three 8-bit color channels.

Throughout this paper we use two networks with different dimensions. They are named NN-64 and NN-128 and are built respectively with $n = \{64, 128\}$. Our network NN-128 actually has same architecture as the network originally used in [4] except for the number of output neurons that we increased from 10 to 100 to enable using datasets with up to 100 classes.

In the rest of the paper, we will focus only on the new features we have added as compared to our previous work, so we refer the reader to [26] for detailed internal implementation of layers *SWL*, *TL* and *MPL*.

We now present the parallelism approach we use for the layers (section 3.2), and detail the internals of the neuron layer (section 3.3).

3.2 Parallelization

In our baseline implementation, at most one activation value is transferred per clock cycle between two layers, through FIFO channels. This directly dictates the design throughput, in frames per second (fps). The limiting factor for throughput is the amount of activation values that each layer inputs and produces to process one image. For a given layer, both sides are often not balanced and, in the baseline implementation, this leads to some layers and FIFOs being stalled while other layers (or just one side of some layers) are busy.

To increase throughput, parallelism is introduced in the layers that are responsible for the bottleneck. The corresponding FIFOs are widened to allow transfer of more activation values per clock cycle. The RTL implementation of all layer types allows mostly-independent input/output degrees of parallelism for that purpose. The case of the neuron layer is detailed in Section 3.3.

To determine which layers (or sides of layers) are bottlenecks for throughput, the key notion is the *latency* of the layers. We define input and output latencies as the minimum number of clock cycles needed to transfer data on the corresponding layer side, assuming this is limited neither by the logic and bandwidth on other side, nor by the other layers and FIFOs. For the baseline implementation, *i.e.* where at most one activation value is transferred per clock cycle and per FIFO, these latencies are equal to the number of activation values transferred on input and output to process one frame. For example, for a fully-connected neuron layer with 256 neurons and input size of 1024 values per frame, input latency is 1024 cycles and output latency is 256 cycles. Similarly for each layer type and for a given frame size, these input and output latency values can be calculated statically.

Given these latencies, the creation of designs with increased throughput is done using the following process. Assume we want to create a design with throughput F times higher than the baseline implementation. We call this factor F *acceleration factor* in the rest of the paper. To determine which layers are bottlenecks, and to calculate the appropriate parallelism degrees for both input and output sides for these layers, the following method is used:

(1) calculate the highest latency L_H of both sides (input and output) of all layers in the baseline implementation,
(2) calculate the maximum latency L_M needed on each side of each layer to reach acceleration factor F: $L_M = \lfloor L_H/F \rfloor$,
(3) list the bottleneck layers, *i.e.* layers for which at least one side has baseline latency $L > L_M$,
(4) for each bottleneck layer and each side, calculate the minimum required parallelism degree P from the baseline latency L: $P \geq \lceil L/L_M \rceil$ (and select the lowest P made possible by the implementation of the layer).

Given the number of layers of different types in our designs, this process can be very tedious if done manually. To generate the different versions of our designs with different acceleration factors, we automated the process with a small dedicated program. This program implements a simple model of our neural networks that calculates the latency values and outputs the parallelism degrees for all layers using the aforementioned procedure.

Table 3. Neural network parallelization

NN	Acc.	Parallelism per layer (in/out)											
size	factor	*NL*1	*NL*2	*MPL*1	*NL*3	*NL*4	*MPL*2	*NL*5	*NL*6	*MPL*3	*NL*7	*NL*8	*NL*9
64	1	-	-	-	-	-	-	-	-	-	-	-	-
	2	-	2 / 1	-	-	-	-	-	-	-	-	-	-
	4	-	4 / 1	-	-	2 / 1	-	-	-	-	-	-	-
	8	-	8 / 1	-	2 / 1	4 / 1	-	-	2 / 1	-	-	-	-
	16	1 / 2	16 / 2	2 / 1	4 / 1	8 / 1	-	2 / 1	4 / 1	-	-	-	-
	32	2 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	4 / 1	8 / 1	-	-	-	-
	64	3 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	-	-	-
	128	9 / 16	192 / 16	16 / 4	32 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	-	-	-
	256	27 / 32	576 / 32	32 / 8	64 / 16	128 / 16	16 / 4	32 / 8	64 / 8	8 / 2	2/1	-	-
128	1	-	-	-	-	-	-	-	-	-	-	-	-
	2	-	2 / 1	-	-	-	-	-	-	-	-	-	-
	4	-	4 / 1	-	-	2 / 1	-	-	-	-	-	-	-
	8	-	8 / 1	-	2 / 1	4 / 1	-	-	2 / 1	-	-	-	-
	16	1 / 2	16 / 2	2 / 1	4 / 1	8 / 1	-	2 / 1	4 / 1	-	-	-	-
	32	1 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	4 / 1	8 / 1	-	-	-	-
	64	2 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	-	-	-
	128	3 / 16	128 / 16	16 / 4	32 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	-	-	-
	256	9 / 32	384 / 32	32 / 8	64 / 16	128 / 16	16 / 4	32 / 8	64 / 8	8 / 2	2/1	-	-

Table 3 gives the degrees of parallelism applied to the design layers in order to accelerate the baseline design. For clarity, parallelism degree numbers are only shown for bottleneck layers. For convenience, the considered acceleration factors are powers of 2 in range 1 to 256, where 1 indicates the baseline implementation with no acceleration and no parallelism. We give the full 1 to 256 range as illustration of scalability. Some of the highest-throughput designs actually require too many resources for our target FPGA.

The table shows that to get up to 8× more throughput than the baseline version, it is enough to add parallelism on the input side of only a small number of neuron layers. So the impact on the overall resource usage is expected to be low. For higher acceleration factors, more layers have to be parallelized, on input side as well as output side, so resource usage should increase more rapidly (see results in Section 5.2).

One achievement of the present work is the implementation necessary for acceleration factors 128 and 256. For this we had to implement special parallelism functionalities in our *SWL* layer. Previously, for a 3×3 window that has N channels, we could only generate values from the same window coordinate at each clock cycle, hence the output parallelism was limited to N (*e.g.* for NN-64, 3 for *SWL*0 and 64 for *SWL*1). Our new design allows to output values from several coordinates within the window, at each clock cycle. For NN-64 with acceleration factor 128 and NN-128 with acceleration factor 256, we output one entire window row at layers *SWL*0 and *SWL*1, so all values for each window position are output in only 3 clock cycles. And for NN-64 with acceleration factor 256, we output one entire window per clock cycle at these layers.

The case of NN-128 with acceleration 128 was challenging in a different way. Even though it did not require special layer implementation to achieve the required functionality, its overall resource usage was just too high for our FPGA and we considered several creative low-level optimizations and trade-offs to try to make it fit in our FPGA. These works are presented in Section 4.

3.3 Neuron Layer Implementation

One Neuron Layer (*NL*) is composed of neurons and a memory holding the ternary weights. At each clock cycle, one or more input activation values are broadcast to all neurons. Simultaneously, the weights are read from the memory and distributed to the appropriate neurons. All neurons then perform one or more multiply-accumulate operations on an internal register (accumulator).

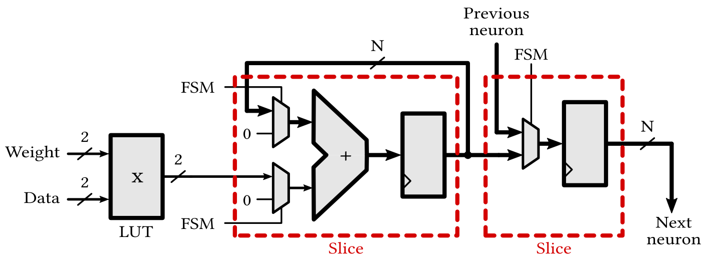


Fig. 2. Neuron implementation, with ternary inputs

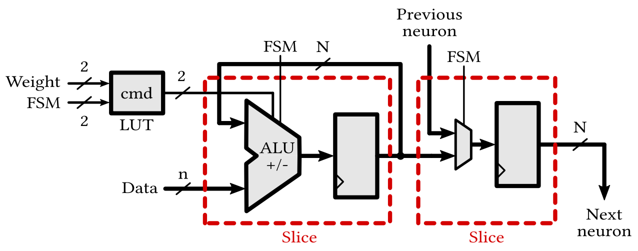


Fig. 3. Neuron implementation, with input data wider than ternary

To extract the values out of the neurons accumulators as well as to allow a compact placement in the FPGA, neurons are interconnected and form a scan chain, as proposed in [29]. This scan chain has its own registers, which enables to copy all accumulators values at once and to extract them while the accumulators accumulate the results of the computations on the next frame data.

The architectural interest of using ternary values is illustrated in Figure 2, which details the internal structure of the proposed neuron. The ternary multiplier requires two 4-input LUTs which fit into one unique 6-input LUT on a Xilinx FPGA, using the two outputs. Hence the neuron mainly consists of its two registers and associated ALUs and multiplexers. The ALUs and multiplexers are small enough to fit in the same *slice* with their associated registers. The neurons may use more than one slice in height, depending on the accumulator width that is required in the layer. For neuron layers whose input activations are not ternary (*e.g.* *NL*0 has 8-bit input), we use the alternative implementation represented in Figure 3. Instead of using one LUT to perform the ternary multiplication, it uses one LUT to generate a command for the ALU of the accumulator, which then performs addition or subtraction. For a same accumulator width, the resource usage is then similar in both implementations.

For resource efficiency, control signals are generated by a finite state machine (FSM) that is shared among all the neurons of a layer, in an SIMD fashion. Weight sparsity is intentionally not exploited. Indeed, compared to our very optimized FSM and neurons, the amount of per-neuron control needed to handle sparsity would come at an excessive cost in area and power. As a result, in the FPGA used in our experiments (433200 LUT and 3600 DSP cores) it is possible to implement 5 to 6× more 12-bit ternary neurons (19 LUT each with no parallelism) than neurons based on DSP cores.

Parallelism degrees for input and output of the *NL* (P_i and P_o) are independent. For input parallelism, each neuron performs several multiplications in parallel and sums these with an adder tree illustrated in Figure 4. Figure 5 illustrates how parallelism is applied with $P_i = 4$ and $P_o = 2$. On the input side, each neuron receives P_i weight and activation values, which are added up with

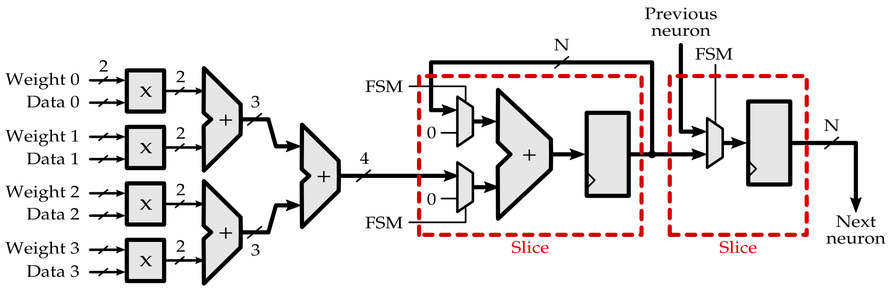


Fig. 4. Neuron implementation, with parallelism degree 4 for input

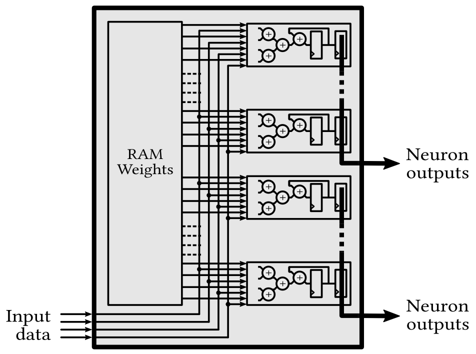


Fig. 5. Neuron layer implementation, with parallelism degrees 4/2 for input/output

the adder tree before the accumulator. On the output side, the scan chain is split into P_o smaller scan chains so several neuron results are output per clock cycle.

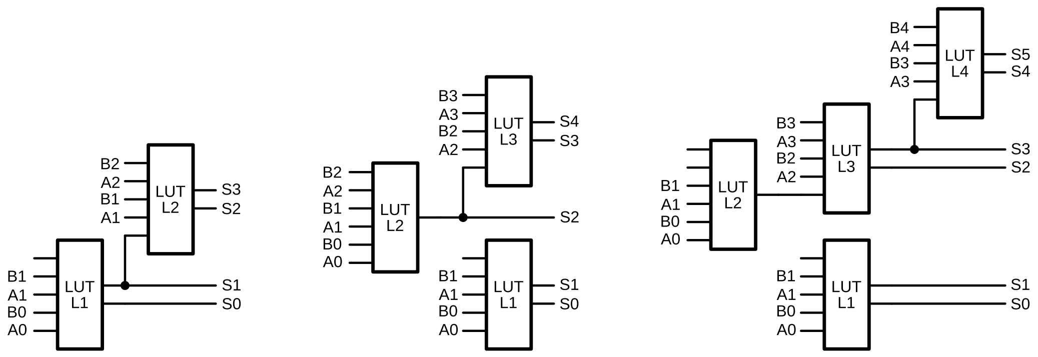
The weight memory is implemented either using RAM blocks or using the LUTRAM functionality of certain LUTs of the FPGA. For each neuron layer, the memory implementation can be forced using a generic parameters in our RTL implementation. We use a simpler rule compared to our previous works [26]: with W the number of weights per neurons, LUTRAM is used when $\lceil W/P_i \rceil \leq 64$, otherwise RAM blocks are used. This balances well the usage of LUTs for memory and for the neuron logic, while reserving RAM blocks for the deepest memories of the network.

Note that NN-64 with acceleration factor 256 is a special situation because the input parallelism in layers $NL0$ and $NL1$ is high enough so the accumulation is performed in only one clock cycle. The one-word memory of weights is then actually implemented in registers instead of RAM-capable primitives.

Compared to our previous works where the neurons were separated into P_o groups, each having its own separate small memory of weights, we now store the weights in a fully packed way as shown in Figure 5. This limits memory space waste and is key for compression (see section 4.3).

4 OPTIMIZATIONS AND TRADE-OFFS

We present several netlist-level optimizations that we use for adder trees, including ternary adders for which only 3 of the possible 4 combinations are used. Although they are designed for Xilinx 6-input LUT architectures, similar tricks should be possible for other FPGA families and architectures.



(a) 3-bit adder in 2 LUTs (b) 4-bit adder in 3 LUTs (c) 5-bit adder in 4 LUTs

Fig. 6. Optimized implementations of 2-input adders

4.1 Adder tree optimizations

For the highest parallelism degrees, the adder trees in the neuron layers use most of the logic resources of our designs. The naive RTL implementation using the standard operator + generally results in N LUTs and a carry chain being used for two N-bit input values. Indeed this is generally the best solution for delay and an acceptable one for area. However in our designs, given the high ratio of resources used by these adders, it is worth considering more compact solutions.

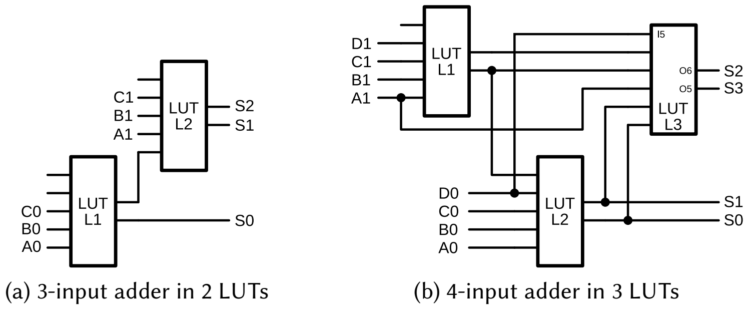
Figure 6 illustrates the template of our optimized 2-input adders for input widths of 3, 4 and 5 bits, with resource usage of respectively 2, 3 ad 4 LUTs and no carry chain. For each adder this is one LUT less than what synthesis produces. Given the low bit width this represents a notable saving between 20% and 33% for this part of the designs. We consider only relatively small bit widths because most neuron layers have balanced ternary activations, so most of the adder tree area is in the low bit-width part of the tree.

The functionality of the different LUTs in the adders can be easily understood from Figure 6 for 3-bit and 4-bit inputs. For 5-bit inputs, the LUT L2 generates the third bit of the partial sum of $A[1-0]$ and $B[1-0]$. This enables the LUT L3 to generate the bits S2 and S3 of the final sum, followed by the LUT L4 to generate the last two bits of the final sum.

Similarly, for adder trees with balanced ternary inputs, there are optimization opportunities due to the reduced output range and the correlation between the two bits of each input value. Figure 7 shows the implementation of our balanced ternary adders for 3, 4 and 7 inputs.

The 3-input adder (Figure 7a) is very good for its compactness and the usage of the maximum of its 3-bit output range. It uses only 2 LUTs where synthesis generates 3 LUTs from behavioural RTL code. The LUT L1 generates the 2-bit partial sum of the bits index zero of the inputs. Then the LUT L2 generates the remaining two MSBs of the final sum from the MSB of this partial sum and from the bits index 1 of the inputs.

The 4-input adder (Figure7b) is used mostly for adder trees with exactly 4 inputs. It also has the remarkable property that the two outputs of one of its LUTs (L3) are used while actually using all of the 6 inputs, which contributes to its high resource efficiency and which is very uncommon. It uses only 3 LUTs where synthesis generates 6 LUTs from behavioural RTL code. The LUT L1 computes the partial sum of the bits index 1 of the inputs and generates only the two LSBs of the result. Then the LUT L2 computes the partial sum of the bits index zero of the inputs plus the LSB of the partial sum generated by L1, which generates the bits index 0 and 1 of the final sum. Finally, the LUT L3 receives both these partial sums plus two bits from the inputs. This enables to rebuild



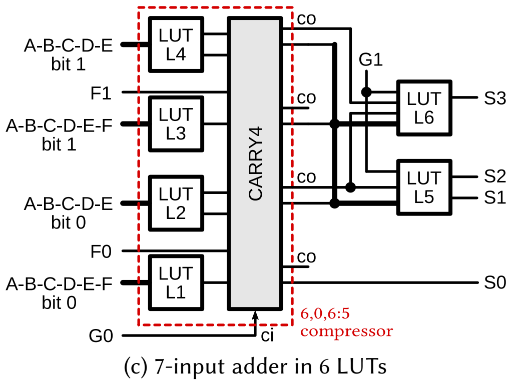


Fig. 7. Optimized implementations of balanced ternary adders

Table 4. LUT savings with optimized ternary adder tree

Number of inputs	4	8	16	32	64	128	192	256	384	576
Generic 2-bit radix-2 adder tree (LUT)	6	21	44	90	181	380	568	757	1289	1945
Optimized ternary adder (LUT)	4	9	21	44	90	184	274	371	555	839
Savings	33.3%	57.1%	52.3%	51.1%	50.3%	51.6%	51.8%	51.0%	56.9%	56.9%

the full 3-bit partial sums and to generate the two remaining bits of the final sum. Note that the assignment of input I5 and the order of the two outputs is critical.

The 7-input adder (Figure 7c) aggregates the highest number of inputs and uses the 4-bit output range at its maximum, which minimizes the rest of the adder tree. For adder trees with large number of inputs, the best implementation is with radix 7, *i.e.* inputs are initially added by groups of 7 with this adder. This adder is built around one *6,0,6:5* compressor [15], a structure that is normally used in compressor trees. Our solution uses the outputs of the compressor component (plus other carry-out outputs from its carry chain) as a signature, which has no arithmetic meaning but which is enough to rebuild the correct arithmetic sum with only two additional LUTs (L5 and L6). Overall, this adder uses only 6 LUTs where synthesis generates 13 LUTs from behavioural RTL code.

We quantify the interest of using this optimized ternary adder tree compared to a traditional radix-2 adder tree for 2-bit inputs in Tables 4 and 5. Table 4 focuses on savings on a single adder tree while Table 5 gives the gain for the entire NN-64 and NN-128 designs. Savings are significant

Table 5. Overall LUT savings (logic only) with optimized ternary adder tree

Acc. factor	4	8	16	32	64	128	256
Savings for NN-64	0.18%	1.38%	4.61%	10.9%	17.3%	24.1%	32.0%
Savings for NN-128	0.22%	1.71%	5.63%	12.9%	19.6%	25.7%	

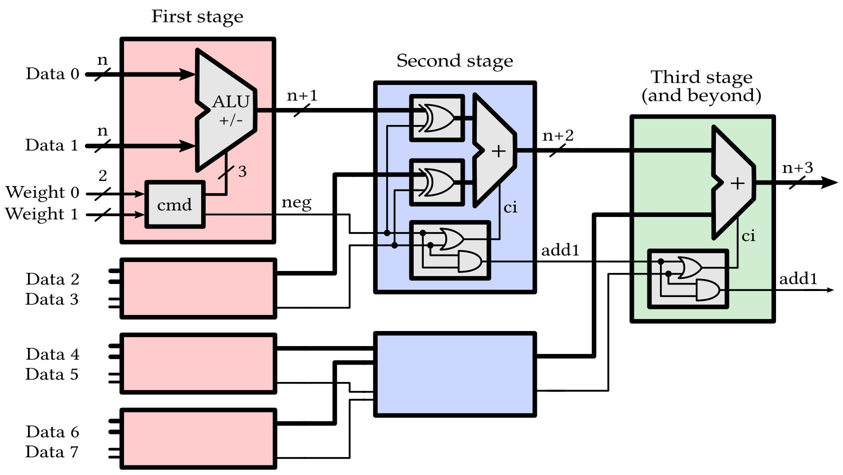


Fig. 8. Neuron implementation, with merged multiply-adder tree

for designs with high acceleration factors, and reach up to ≈ 32% for NN-64 with acceleration factor 256.

4.2 Merged multiplier-adder tree for ternary weight

The naive implementation of a parallel multiplier-adder tree is illustrated in previous Figure 4. In a first stage, the input data are multiplied with the balanced ternary weights. For balanced ternary input data the multiplier is just one LUT, which is very efficient. But for larger data the multiplier uses one ALU and for N-bit data this represents N LUTs per input. The cost of these multipliers is then high.

For large data width (*e.g.* larger than 4 bits) we propose another approach illustrated in Figure 8. The multiplication operations are distributed inside the stages of the adder tree. The first stage is an ALU with 3-bit command signal, which is the maximum to fit the 5 available inputs of the LUTs. The 8 cases where the two weights are not both −1 are fully handled by this stage and other stages behave like a normal adder tree. In the case where both weights are −1, a simple addition is performed (as if both weights were +1) and a *neg* signal is sent to the second stage. This *neg* signal tells the second stage to negate the data, *i.e.* to invert it (hence the *xor* operation) and to add 1 (using carry input). But the second stage can't fully handle when both its inputs have to be negated because it can add 1 only once; so in this case the need to add the second 1 is propagated to the next stage, and through all other stages if necessary, until it reaches the neuron accumulator where it is finally handled as carry input.

On average, compared to the adder tree alone, this implementation adds 2 LUTs in the first stage and 1 LUT in all other stages, regardless of the input data width. For large numbers of inputs, it can be computed that the average cost is around 1.5 LUT per input. The saving compared to the naive implementation (with a multiplier by a ternary weight, followed by an adder tree) are given

Table 6. LUT savings with merged multiplier-adder tree

Inputs	2 bits	4 bits	8 bits	12 bits	16 bits	24 bits	32 bits	Infinity
2	33.3%	50.0%	58.3%	61.1%	62.5%	63.9%	64.6%	66.7%
3	27.3%	42.9%	51.2%	54.1%	55.6%	57.0%	57.8%	60.0%
4	20.0%	37.9%	47.4%	50.6%	52.2%	53.8%	54.7%	57.1%
8	14.7%	32.8%	42.7%	46.2%	48.0%	49.7%	50.6%	53.3%
16	12.3%	30.4%	40.5%	44.1%	46.0%	47.8%	48.7%	51.6%
32	11.2%	29.1%	39.4%	43.1%	45.0%	46.9%	47.8%	50.8%
64	10.6%	28.5%	38.9%	42.6%	44.5%	46.4%	47.4%	50.4%
128	10.3%	28.1%	38.6%	42.3%	44.2%	46.2%	47.2%	50.2%
Infinity	10.0%	27.8%	38.2%	42.0%	43.9%	45.9%	46.9%	50.0%

in Table 6, for several input data widths. With LUT saving up to 66.7%, the impact on this part of the designs is strong. Note that optimized adders presented in Section 4.1 are not considered in this specific study.

These results assume that a multiplier by a ternary value is a N-LUT ALU, for N-bit input values. This is the case for *NL*0 with its 8-bit input width. However, *NL*0 has relatively low input parallelism compared to other neuron layers, so it represents only a small part of our designs. For NN-64 and acceleration factors 32, 64, 128 and 256 (see Table 3), the overall LUT saving is respectively 0.88%, 1.03%, 1.8% and 2.8%. Similarly for NN-128 and acceleration factors 64 and 128, overall LUT saving is respectively 0.80% and 0.77%.

Unfortunately this optimization is not interesting for layers other than *NL*0. Indeed other layers have ternary input, and a ternary multiplier fits in just one LUT. So compared to the adder tree alone, the overhead of 1.5 LUTs on average is higher than the 1-LUT multipliers. The appropriate solution for ternary input is with separate multiplier and with adder tree based on low-area adder implementations presented in Section 4.1.

4.3 Compression of weights

The naive way to encode one balanced ternary value (trit) is to use 2 bits. However, given that 2 bits can represent 4 values and one trit uses only 3, this code is sub-optimal. In particular for the large memories of ternary weights in neuron layers, there is waste of memory bits.

There is a solution to this problem: compression. As can be seen in our neuron layer architecture depicted in Figure 5, when a high degree of parallelism is used the memory word is very large as a lot of weights have to be accessed simultaneously per clock cycle. This makes most common compression approaches impractical, in particular those that have non-uniform throughput and/or that require control to access memory. Because of that, the decompression approach we chose presents control neither on data flow nor on address calculation path, which enables to use pipelining everywhere and is key to sustain high throughputs. The idea is to manipulate several trits together and to devise a more compact code for that group. For example, 3 trits represent $3^3 = 27$ combinations, so only 5 bits are enough to represent these instead of 6 bits with naive 2-bit encoding per trit. In the general case, for a group of T trits, we want to know the optimal number of bits b required to store these. The formula is:

$$b = \left\lceil \log_2 \left(3^T\right) \right\rceil = \lceil T \times \log_2(3) \rceil \tag{2}$$

For large values of T, the limit in average number of bits per trit is:

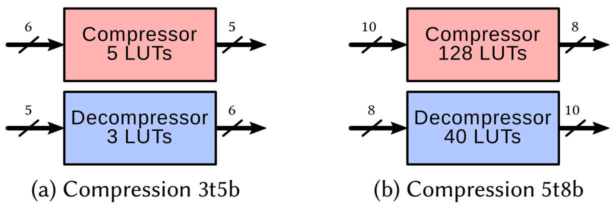
$$b/T \approx \log_2(3) \approx 1.585 \text{ bits} \tag{3}$$

This represents a maximum theoretical saving of $\approx 20.75\%$ compared to the naive 2-bit encoding. However, it can be shown that $\log_2(3)$ is irrational, so there is no exact solution for any value of T

and this limit is not actually reachable. Only a few values for T lead to interesting configurations. We identified two of these that are relevant for FPGA implementation:

- 3 trits / 5 bits (saving 16%),
- 5 trits / 8 bits (saving 20%).

The next configuration that brings better saving is with 17 trits and 27 bits for a saving of 20.59%. However with codes of this size, it would be impractical to build encoder and decoder (we call these compressor and decompressor) in FPGA. The saving brought by the 5-trit configuration is already close enough to the limit for our goals, so we consider only the 3-trit and 5-trit configurations.



(a) Compression 3t5b (b) Compression 5t8b

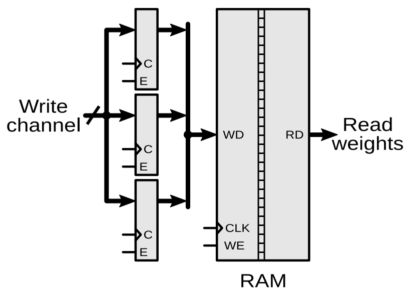
Fig. 9. Compressors and decompressors of balanced ternary values

The compressor and decompressor for these two situations are illustrated in Figure 9. The structure of both components is a standard look-up table. The 5-trit configuration brings the best compression ratio, however it is much more resource-hungry than the 3-trit configuration.

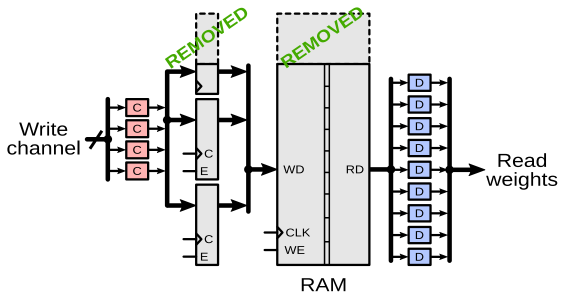
The insertion of compressors and decompressors around the memory of weights inside neuron layers is represented in Figure 10. Assuming that n is the number of neurons and p the degree of parallelism, the size of the word (drawn vertically) is $2 \times n \times p$ for the uncompressed scheme, visible in Figure 10a. The impact on hardware resources is illustrated in Figure 10b. For the sake of concreteness, Figure 10c details the compression and decompression paths when parallelism is involved. The thin lines out of the memory are unique bits, while the thick lines entering the neurons are trits encoded on two bits. We use 3t5b compression, in which 3 weights (*i.e.* 3 trits) are compressed into 5 bits. In that situation, the memory word contains $\lceil \frac{n \times p}{3} \rceil \times 5$ bits (padded with zeros if needed). On the write side of the memory, a shift register-like structure enables to build a full-width memory word using only a small number of compressors (3 in the example).

We however have to be careful: the gain obtained by reducing the word size should not be offset by the resources needed for the compression/decompression process.

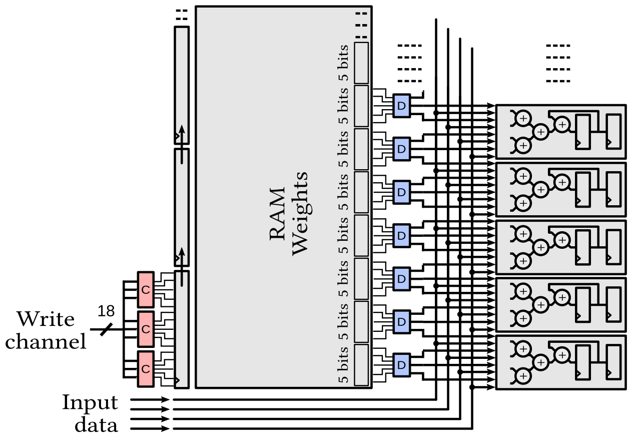
Regarding compression, there are two possible approaches. One approach consists in performing compression in software and thus writing pre-compressed weights inside the *NL* memories. This would leave only the decompression part to implement in hardware, but would make the pre-compressed data specific to a special implementation of the *NL*. Indeed, the values to upload in the weight memories would depend on the nature of the compression (3t5b or 5t8b) and other details such as the low-level implementation of the memories of weights (LUTRAM or BRAM), etc. To avoid this source of hardware/software discrepancy, we favor another approach which consists of implementing the compressors in hardware. This makes the presence of compression completely transparent to the software application. The resource usage of the compressors can be kept very low anyway by choosing a narrow write channel and hence a correspondingly low number of compressors. The shift register on the write side is filled in several clock cycles from the write channel, then it is written to memory in one cycle, and so on for each word of the memory. This allows trading compression resources for *NL* configuration time (which for many applications is not critical).



(a) Without compression



(b) Hardware impact of compression



(c) 3t5b compression applied on a *NL* with input parallelism $P_i = 4$

Fig. 10. Compression/decompression of balanced ternary values applied to a memory of weights

On the read side, the full memory width has to be decompressed in one clock cycle, therefore a large number of decompressors have to be instantiated in each neuron layer of our designs. As opposed to the compression side, there is no leeway there, so decompressors are expected to have a noticeable impact on the overall FPGA resource usage. To save the largest amount of memory bits, it is best to implement compression on neuron layers needing the widest memory of weights. This means that a lot of decompressors have to be used, consequently using more resources. Hence there is a trade-off between memory saving and decompression resource overhead.

The decompressors are small look-up tables or ROM memories and appropriate FPGA implementations use LUT or BRAM primitives. Considering the amount of these two resources and their granularity in the FPGA, we chose LUT-only implementation for decompressors. The implementation of compressors has much lower criticality, we also chose LUT for simplicity. However, these LUT-based decompressors cost more (in LUTs) than it would save from the memory if this memory of weights is implemented in LUTRAM (which is actually one functionality of some LUTs). So this compression functionality is only pertinent for neuron layers whose memory of weights is implemented in BRAM. This is the case for fully-connected layers (with the largest memories) and some convolutional layers with low input parallelism degree.

5 EXPERIMENTS AND RESULTS

In this section, we describe our experimental setup and present area, throughput and power consumption results for all possible configurations, for the VC709 FPGA board. For reference, this board is equipped with the Xilinx FPGA XC7VX690T which features 433200 6-input LUTs including 174200 LUTRAM-capable LUTs, 866400 flip-flops (FF), 2940 18kb RAM blocks (BRAM18) and 3600 hard multiplier-ALU cores (DSP).

5.1 Experimental Setup

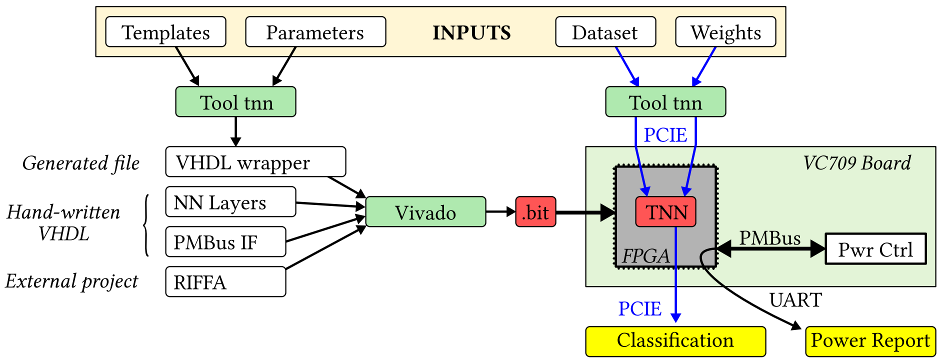


Fig. 11. FPGA demonstrator setup

The setup of the FPGA demonstrator used for our experiments is detailed in Figure 11. The PCI-Express framework RIFFA [11] is used with 2 channels for high-throughput data transfers between the PC and the FPGA. One channel is dedicated to read and write the configuration registers of our CNN IP. The other channel is for transfer of weights and thresholds into memory elements, at configuration time, and for bidirectional transfer of input images and classification results, at inference time. The designs also feature a custom and tiny UART-to-I2C bridge so the on-board PMBus-capable power supplies can be accessed from software with no interference with data transfers. The power consumption of the FPGA can thus be monitored in real-time on all voltage rails, although we focus on the core 1V rail.

Given the large number of design configurations considered, we use a custom tool, `tnn`, to automate the generation of designs. Given some user parameters such as layers, numbers of neurons, acceleration factor and compression type, the `tnn` tool produces the instantiation of the entire CNN in VHDL language, within a custom RTL wrapper (pre-equipped for the PCIe and UART data interfaces) and with the appropriate RTL-level per-layer generic parameters. All designs are synthesized with Xilinx Vivado 2015.3 tool suite, with a frequency constraint of 250 MHz.

The `tnn` tool also handles the way the configuration and inference tasks are performed at run-time. For configuration, it loads weights and thresholds data from disk files, converts this into the appropriate packed data stream that suit per-layer internal memory implementation, and sets the global configuration registers of the CNN IP. Once configured, the FPGA accelerator can be used for any number of inference tasks. At inference time, the `tnn` tool loads the image data from disk files, reorders pixel data for the input neuron layer, and sends this data to the accelerator through PCI-Express. While image data is being transferred, classification results are sent back to the software side through PCI-Express too.

The throughput of the accelerator is the number of images per second (fps) it can process. The total inference time is measured between the start of the transfer of a set of images to the accelerator

Table 7. Resource usage, latency, throughput, power

NN	Acc.	Resource usage				Latency	Throughput	Power	Efficiency
size	factor	LUT (logic)	LUTRAM	BRAM 18k	FF	(ms)	(fps)	(W)	(fps/W)
64	1	70872 (16.4%)	546 (0.31%)	586 (19.9%)	90511 (10.4%)	3.060	422.5	1.24	340.7
	2	71124 (16.4%)	550 (0.32%)	586 (19.9%)	91305 (10.5%)	1.838	845.1	1.43	591.0
	4	71707 (16.6%)	558 (0.32%)	595 (20.2%)	94126 (10.9%)	1.121	1690.3	1.59	1063.1
	8	73310 (16.9%)	582 (0.33%)	616 (21.0%)	103091 (11.9%)	0.634	3381.2	1.87	1808.1
	16	78149 (18.0%)	3458 (1.99%)	647 (22.0%)	119378 (13.8%)	0.360	6763.3	2.53	2673.2
	32	90231 (20.8%)	3962 (2.27%)	770 (26.2%)	149996 (17.3%)	0.233	13525.3	3.89	3476.9
	64	112533 (26.0%)	24098 (13.83%)	844 (28.7%)	195215 (22.5%)	0.167	27042.9	6.44	4199.2
	128	170555 (39.4%)	37402 (21.47%)	1410 (48.0%)	321352 (37.1%)	0.135	60257.8	11.53	5226.2
64	256*	302748 (69.9%)	106168 (60.9%)	2920 (96.7%)	640849 (74.0%)		≈ 122070		
128	1	116239 (26.8%)	638 (0.37%)	2203 (74.9%)	152153 (17.6%)	6.001	211.3	2.77	76.3
	2	116608 (26.9%)	642 (0.37%)	2203 (74.9%)	153561 (17.7%)	3.631	422.6	2.93	144.2
	4	114951 (26.5%)	650 (0.37%)	2203 (74.9%)	159116 (18.4%)	2.178	845.3	3.52	240.1
	8	117804 (27.2%)	674 (0.39%)	2179 (74.1%)	177404 (20.5%)	1.228	1690.8	4.29	394.1
	16	126688 (29.2%)	762 (0.44%)	2252 (76.6%)	213444 (24.6%)	0.724	3382.1	5.42	624.0
	32	148717 (34.3%)	12022 (6.90%)	2347 (79.8%)	267862 (30.9%)	0.424	6764.0	7.91	855.1
	64	193968 (44.8%)	13090 (7.51%)	2766 (94.1%)	377985 (43.6%)	0.212	13525.8	14.03	964.1
128	128*	276590 (63.9%)	91392 (52.5%)	2920 (99.3%)	551260 (63.6%)		≈ 27127		

* Designs too large for our device. Resource usage results are after logic synthesis, throughput is theoretical value.

and the end of the reception of all classification results. This time is then divided by the number of processed images, when processing a large number of images in one go. We measure latency the same way but with just one image. This measure includes parasitic PCI-Express and RIFFA latencies plus other small delays due to software threads, so the reported latency is higher than the latency of the FPGA IP alone, but it may still be interesting for readers as a reference as it is more representative of what would happen in an actual application.

Power measurement is performed while processing the dataset CIFAR10. It has already been shown in our previous works [26] that the power consumption of our designs is largely independent from the dataset being processed, and we have verified that this is still the case with the present optimizations.

5.2 Resources, Latency, Throughput, Power

Synthesis results are given in Table 7. The considered acceleration factors are powers of 2 in range 1 to 128, for both NN-64 and NN-128. Area results show LUT usage as logic and as LUTRAM, as well as BRAM and flip-flops (FF). DSP usage is not shown because it is rather constant in all designs: 95 to 99 DSP per design (2.64 to 2.75% of the FPGA).

Note that acceleration factor 128 for NN-64 actually presents a throughput improvement of 142× compared to the baseline design, and by 2.2× compared to acceleration factor 64. This is due to our new implementation for layers *SWL*0 and *SWL*1, which were bottleneck layers: their throughput improved 3× instead of just 2× compared to acceleration factor 64. We still refer to it by acceleration factor 128 for simplicity and for symmetry with NN-128.

Acceleration factor 256 for NN-64 leads to very high FPGA resource utilization ratio and Vivado failed to perform placement and routing. Given the overall LUT usage ratio (94.4%), even considering compression or using DSP cores would not lead to a feasible design. The case of NN-128 with acceleration factor 128 is similar except overall LUT usage ratio (84.9%) leaves some margin to try compression of weights (see details in Section 5.4), unfortunately with no success.

For the highest acceleration factors that fit our FPGA, our latency is below 200 μs, which even includes some PCI-Express related delays. This is extremely small for this kind of PCI-Express accelerated convolutional networks. To the best of our knowledge, our maximum throughput of

Table 8. Resource usage and power compared to [26]

NN size	Acc. factor	Resource usage LUT (logic)	LUTRAM	BRAM 18k	FF	Power
64	1	+5.31%	-9.90%	-12.14%	-8.11%	-30.25%
	2	+5.33%	-30.03%	-11.08%	-8.42%	-28.68%
	4	+4.18%	-29.72%	-11.06%	-8.02%	-22.61%
	8	-2.18%	-81.45%	-6.81%	-9.76%	-18.65%
	16	-6.73%	-32.43%	-10.76%	-7.07%	-13.25%
	32	-16.82%	-82.08%	+15.10%	-9.70%	-12.73%
	64	-27.36%	-59.68%	+24.30%	-22.93%	-11.21%
128	1	+2.70%	-39.01%	-3.21%	-22.03%	-16.88%
	2	+2.65%	-38.86%	-3.21%	-22.10%	-14.56%
	4	-0.71%	-38.56%	-3.21%	-21.20%	-13.57%
	8	-8.32%	-53.00%	-6.12%	-18.73%	-10.46%
	16	-12.62%	-91.86%	-0.53%	-13.59%	-5.83%
	32	-22.22%	-71.77%	+5.06%	-15.40%	-10.58%
	64	-29.48%	-84.57%	+26.47%	-19.34%	-2.25%

$60.2k$ fps and $13.5k$ fps, respectively for NN-64 and NN-128, is also the highest reported in the literature. Using a slightly larger FPGA would directly enable to synthesize the next acceleration factor for our two networks and hence to double these throughput results. The power efficiency of our designs is also improved compared to our previous works and it is now only 14% lower than the binary framework FINN [33], while still having much better accuracy [26].

5.3 Changes since previous works

Our new designs, as our previous works, are pure hardware accelerators for ternary CNN, therefore, classification accuracy is identical. However, our new designs include several implementation optimizations:

- use PCI-Express Gen 1 instead of Gen 2 and 64-bit Riffa channels instead of 128-bit (reduced need for LUT, FF and BRAM),
- remove a few pipeline stages (reduced need for FF),
- simplify the choice between LUTRAM and BRAM for implementation of memory of weights (reduced need for LUTRAM, increased need for BRAM),
- better packing of neuron weights into BRAM (reduced need for BRAM),
- use a large write register for neuron weight memories (slightly increased need for LUT and FF),
- use optimized adders (strongly reduced need for LUT),
- increase performance and power efficiency.

The overall impact of these changes is given in Table 8: LUT and FF usage is strongly reduced, BRAM usage is increased and power consumption is strongly reduced. For the same acceleration factor, throughput and latency have not changed, as well as DSP usage, so these are not present in the table. Power efficiency improvement is then identical to power consumption improvement.

Note that for low acceleration factors, 1 to 4, LUT usage increases slightly. This is due to the change of packing of weights in neuron layer memories and the impact on the addition of a large register and associated logic on the write side of these memories. This part only draws power when writing weights and not while running inference tasks, hence the strong power saving due to the other changes.

Power consumption changes notably with the chip temperature, which depends on room temperature, air flow, PC enclosure, etc. To make a fair comparison, we measured power for our new design, but also measured again power for our previous designs. Both measures being done in conditions as similar as possible, they can be compared.

5.4 Compression

All designs were synthesized with the two types of compression, 3 trits / 5 bits (noted 3t5b) and 5 trits / 8 bits (noted 5t8b). Only neuron layers with BRAM-based weight memory are affected by compression. For each type of compression, we consider two situations:

- only fully-connected layers have compression (identified with the FC-* prefix),
- all neuron layers with BRAM have compression (identified with the BRAM-* prefix).

We consider the "FC layers only" case because a significant amount of the overall design memories is in the FC layers (and in BRAM), so targeting these layers only should already bring significant memory saving. Additionally, the FC layers have no input parallelism in our experiments, so the overhead of the decompressor blocks should be minimal. On the other hand, non-FC layers can have high parallelism, so targeting all neuron layers should bring the highest memory savings but at the cost of a higher logic overhead.

Results on area and power compared to designs with no compression are given in Figure 12. Compression has no impact on LUTRAM and on DSP usage so it is not present in the results. Note that some designs for NN-128 with acceleration factors 64 and 128 exceed the FPGA size, so area results are after logic synthesis only, and no power result is present.

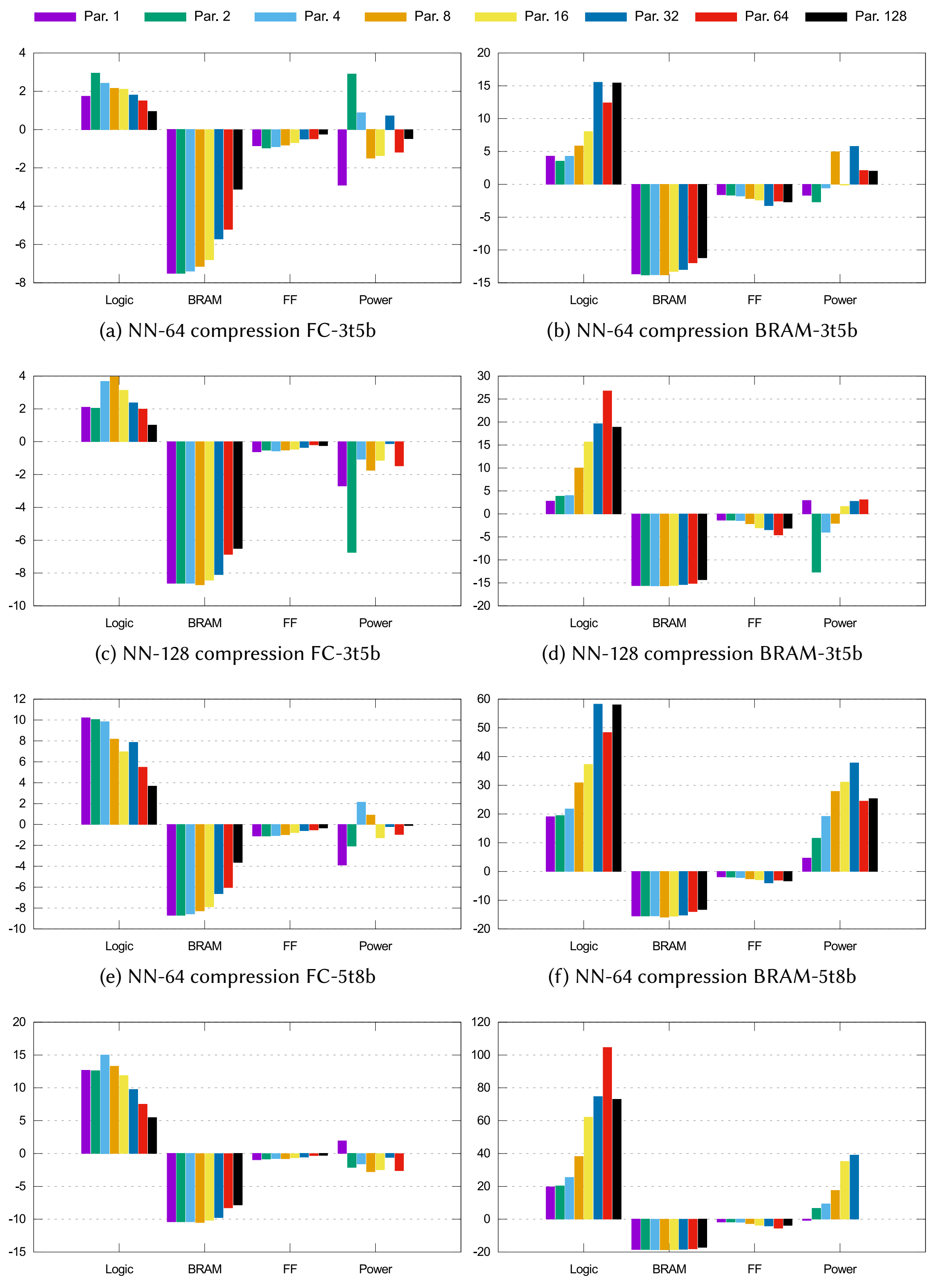
For FC-3t5b compression, BRAM saving in range 6 to 8% is obtained, as well as a moderate 2% increase in logic. For BRAM-3t5b compression, BRAM saving is higher, in range 12 to 15%, which is much closer to the theoretical maximum saving of 16%. The difference is due to the fact that some BRAM blocks are used in parts of the design not affected by compression, *e.g. SWL* and *TL* layers and Riffa interface. The logic overhead of compression BRAM-3t5b is much higher with a maximum of +27% and this can be correlated to a slight increase in power for NN-128 in Figure 12d. For other FC-3t5b designs, the impact on power is too low on average to differentiate signal from noise. Additionally FC layers stay idle for a significant part of the time, which explains why compression has a very low impact on power, even for compression type 5t8b.

Compression type 5t8b has a much higher overhead on logic utilization compared to compression 3t5b, even when targeting only FC layers. When targeting only FC layers, BRAM saving is in range 6 to 10% at the cost of a logic overhead in range 6 to 15%. The trade-off is much less interesting when targeting all BRAM layers, with BRAM saving in range 15 to 19% (for a theoretical maximum of 20%) and logic overhead in range up to +75%. The impact on power is obvious and strongly correlated to logic overhead: power is higher than non-compressed designs, with an increase of up to +40%.

We note that the ratio of BRAM saved with FC-only compression decreases when acceleration factor increases. Indeed, BRAM usage in convolutional layers globally increases with parallelism even though some of these layers switch to LUTRAM storage. As BRAM usage is unchanged in FC layers, their relevance for compression decreases.

We tried to use compression FC-3t5b on NN-128 with acceleration factor 128 to reduce its overall BRAM usage. The BRAM requirement decreased to 92.9%, which seems acceptable given that 94.1% was used with success for acceleration factor 64. However the overall LUT usage increased from 84.9% to 85.5% and routing still failed. Other attempts at further reducing LUT usage by packing some adder tree stages into DSP cores, reducing FF usage by removing non-critical pipeline stages, and locking some configuration registers to their synthesis-time value, lead to 79.5% LUT, 48.7% FF and 52.5% DSP.

To conclude, overall compression FC-3t5b brings a good trade-off between BRAM saving and logic overhead, and this technique can be useful to help fit large designs in a given FPGA. Other



(a) NN-64 compression FC-3t5b

(b) NN-64 compression BRAM-3t5b

(c) NN-128 compression FC-3t5b

(d) NN-128 compression BRAM-3t5b

(e) NN-64 compression FC-5t8b

(f) NN-64 compression BRAM-5t8b

(g) NN-128 compression FC-5t8b

(h) NN-128 compression BRAM-5t8b

Fig. 12. Impact of compression, in percent

types of compression may bring higher BRAM saving, but generally at a much higher cost in logic resources, so they are to consider only in corner-case situations with strongly unbalanced resource usage.

6 DISCUSSION

Performance results in this section are given with TOP and TMAC units, where one operation (OP) is one multiplication *or* one addition, and one MAC is one multiplication *and* one addition. The MAC unit is also useful for comparison with implementations on different platforms such as CPU or GPU, for which the multiply-accumulate instruction is often the base of CNN computations.

We reached 18.7 TOP/s performance (9.33 TMAC/s) with our NN-64 network using less than half of our FPGA resources. Our next larger designs are NN-64 with acceleration factor 256 and NN-128 with acceleration factor 128. Although they theoretically fit our FPGA resources, automatic routing failed. Using a slightly larger FPGA would make routing succeed and performance reach 37.4 TOP/s (18.7 TMAC/s) and 33.4 TOP/s (16.7 TMAC/s), respectively.

It can be argued that the scalability of our approach (especially for NN-128) is constrained by the presence on-chip of all neuron weights. Although some on-chip memory could be saved by storing part or the totality of the weights in on-board DDR memory, the memory interfaces would certainly use a significant amount of logic resources and some on-chip memory too. This approach does not seem ideal as our designs are also constrained by logic resources.

Off-chip storage of weights would also create a throughput bottleneck. Our VC709 board is equipped with two 64-bit DDR interfaces that support up to 1600 MT/s, which means a theoretical maximum throughput of 204.8 Gb/s. However, the total throughput of the weight memories of our current designs is up to 18.7 Tb/s for NN-64 at 60.2k fps and 16.7 Tb/s for NN-128 at 13.5k fps, which exceeds by far (and by $\approx 90\times$) the DDR throughput. Storing only FC weights in off-chip memory has been proposed in related works, *e.g.* [5]. But the FC layers of our designs use up to 290 Gb/s for NN-64 and 258 Gb/s for NN-128, which also already exceed the DDR throughput. Furthermore, this would also defeat the low power objective of using ternary weights. This is why we deliberately kept all weights on-chip.

Recent works with binary networks have shown that replacing the largest FC layers by average pooling operations could save a lot of resources while having only a minimal impact on accuracy [21]. Similar impact on our ternary networks can be expected, although dedicated training would be required. This may be considered in future works.

Finally for embedded devices, there is a demand for extremely power-efficient solutions. Many ASIC implementations already exist, but very few using highly quantized weights and/or activations, such as the YodaNN [2] with a binary approach. The question of ASIC integration can also be raised for our ternary approach. We highlight that a ternary multiplier is 3 logic gates, and for non-ternary activations our architecture for multiplier-adder tree shown in Figure 4.2 is a good fit for ASIC. Regarding compression, the trade-off between memory and logic for area and power may be very different. On the one hand, SRAM cuts can be tailored for the exact per-layer requirements, whereas in FPGA there can be strong under-usage of memory blocks. On the other hand, a good ternary-binary code mapping would certainly reduce the decompression logic overhead (in area and power), which would broaden the interest of weight compression.

7 CONCLUSION

Our work brings significant improvements to the FPGA implementation of convolutional ternary neural networks. Using ternary weights and activations allows to embed all parameters of already large networks into the FPGA BRAM and LUTRAM (up to 14 millions of parameters for NN-128). Avoiding external DRAM accesses has two advantages: there is no need to access the very power hungry DRAM and the bandwidth obtained by accessing all on-chip memory blocks in parallel is much higher. This bandwidth can be exploited to parallelize the access to the weights, leading to many different throughput vs power trade-offs. Using 2 bits to encode a ternary value is not optimal. We therefore proposed, implemented and evaluated compression schemes that optimize BRAM usage and that can be useful to squeeze large networks in a given FPGA, or to reach desired trade-offs between area and power consumption.

Overall, we propose an optimized design that can be configured for either low power or high throughput, achieving a peak performance of 18.7 TOP/s (9.33 TMAC/s) and a peak efficiency of 5226 fps per Watt (equivalent to 1.62 TOP/s/W or 810 GMAC/s/W at 250 MHz on NN-64) using half the resources of a XC7VX690T Xilinx FPGA.

ACKNOWLEDGMENTS

The authors would like to thank Olivier Menut from ST Microelectronics for his valuable inputs and continuous support.

REFERENCES

[1] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. 2017. Ternary Neural Networks for Resource-Efficient AI Applications. In *30th International Joint Conference on Neural Networks*. 2547–2554. Training code available at https://github.com/slide-lig/tnn-train.

[2] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. 2017. YodaNN: An Architecture for Ultra-Low Power Binary-Weight CNN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[3] Ken Batcher. 1987. The architecture of tomorrow's massively parallel computer. (Sept. 1987). After dinner talk given at Goodyear Aerospace Corporation.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[5] Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, and Nalin Aggarwal. 2017. 14.1 A 2.9 TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. In *IEEE International Solid-State Circuits Conference*. IEEE, 238–239.

[6] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Scaling Binarized Neural Networks on Reconfigurable Logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. 25–30.

[7] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. 243–254.

[8] Lu Hou, Quanming Yao, and James T. Kwok. 2017. Loss-aware Binarization of Deep Networks. In *5th International Conference on Learning Representations*. C18.

[9] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv:1609.07061* (2016). Retrieved from https://arxiv.org/abs/1602.07061.

[10] Kyuyeon Hwang and Wonyong Sung. 2014. Fixed-point feedforward deep neural network design using weights -1, 0, and +1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. 1–6.

[11] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A reusable integration framework for FPGA accelerators. *ACM Transactions on Reconfigurable Technology and Systems* 8, 4 (Sept. 2015), 22:23.

[12] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2017. A novel zero weight/activation-aware hardware architecture of convolutional neural network. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1462–1467.

[13] Donald E. Knuth. 1997. Seminumerical Algorithms, vol. 2. In *The Art of Computer Programming*. Addison-Wesley, Reading.

[14] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. Toronto University.

[15] Martin Kumm and Peter Zipf. 2014. Pipelined compressor tree optimization using integer linear programming. In *24th International Conference on Field Programmable Logic and Applications*. 1–8.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[17] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv:1605.04711* (2016). Retrieved from https://arxiv.org/abs/1605.04711.

[18] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2017. A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 290–291.

[19] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. 2017. Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems* 10, 3 (July 2017), 17:1–17:23.

[20] Duncan J. M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H. W. Leong. 2017. High performance binary neural networks on the Xeon+FPGA platform. In *2017 27th International Conference on Field Programmable Logic and Applications*. 1–4.

[21] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. 2017. A fully connected layer elimination for a binarized convolutional neural network on an FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications*. 1–4.

[22] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

[23] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, 5–14.

[24] Jinhwan Park and Wonyong Sung. 2016. FPGA based implementation of deep neural networks using on-chip memory only. In *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 1011–1015.

[25] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. 2017. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test* 34, 2 (2017), 39–50.

[26] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. 2017. Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA. In *27th International Conference on Field Programmable Logic and Applications*. 1–7.

[27] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[28] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[29] Peter Škoda, Tomislav Lipić, Àgoston Srp, Branka Medved Rogina, Karolj Skala, and Ferenc Vajda. 2011. Implementation framework for Artificial Neural Networks on FPGA. In *2011 Proceedings of the 34th International Convention MIPRO*. 274–278.

[30] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. 2011. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. In *International Joint Conference on Neural Networks*.

[31] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. 2016. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *arXiv:1602.07261* (2016). Retrieved from https://arxiv.org/abs/1602.07261.

[32] Olivier Temam. 2010. The rebirth of neural networks. Keynote speach at the International Symposium on Computer Architecture.

[33] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74.

[34] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. 2016. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *CoRR* abs/1612.07119 (2016). http://arxiv.org/abs/1612.07119

[35] K. Vissers. 2017. A Framework for Reduced Precision Neural Networks on FPGA. In *17th International Forum on MPSoC*. slides available at http://www.mpsoc-forum.org/previous/2017/files/proceedings/Kees_Vissers.pdf.

[36] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. 2017. A high-throughput reconfigurable processing array for neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications*. 1–4.

[37] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 15–24. https://doi.org/10.1145/3020078.3021741

[38] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained ternary quantization. *arXiv:1612.01064v3* (2017). Retrieved from https://arxiv.org/abs/1612.01064v3.