Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA

Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, Vincent Leroy

Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA

Adrien Prost-Boucle*, Alban Bourge*, Frédéric Pétrot*
Hande Alemdar†, Nicholas Caldwell†, Vincent Leroy†
*Univ. Grenoble Alpes, CNRS, Grenoble INP‡, TIMA, F-38000 Grenoble, France
†Univ. Grenoble Alpes, CNRS, Grenoble INP‡, LIG, F-38000 Grenoble, France
Email: *name.surname*@univ-grenoble-alpes.fr

***Abstract*—Thanks to their excellent performances on typical artificial intelligence problems, deep neural networks have drawn a lot of interest lately. However, this comes at the cost of large computational needs and high power consumption. Benefiting from high precision at acceptable hardware cost on these difficult problems is a challenge. To address it, we advocate the use of ternary neural networks (TNN) that, when properly trained, can reach results close to the state of the art using floating-point arithmetic. We present a highly versatile FPGA friendly architecture for TNN in which we can vary both the number of bits of the input data and the level of parallelism at synthesis time, allowing to trade throughput for hardware resources and power consumption. To demonstrate the efficiency of our proposal, we implement high-complexity convolutional neural networks on the Xilinx Virtex-7 VC709 FPGA board. While reaching a better accuracy than comparable designs, we can target either high throughput or low power. We measure a throughput up to 27 000 fps at ≈7 W or up to 8.36 TMAC/s at ≈13 W.**

I. Introduction

Artificial neural networks (ANN) have had a long and complicated history [1], but there is now a consensus that networks with many layers and many neurons per layer are achieving the best results on a broad range of artificial intelligence tasks. For the record, an ANN needs to be trained on many instances of a problem to determine synaptic weights (a.k.a learning) that are later used to solve a new instance of the same problem (a process called inference). Thanks to advances in integration technology and computer architecture, full software solutions to both learning and inference can be done at high performance on general purpose processors and graphical processing units. However, solving problems like clustering or classification has a lot of interest on systems ranging from the servers in datacenters to battery-powered devices, two kinds of systems in which power efficiency is key.

To achieve better results on ever increasing data sets, ANN have grown wider and deeper, leading to a large number of neurons. As a consequence a lot of floating-point multiplications are needed to realize the multiply-accumulate operations that compute the activations of the neurons. For instance, the implementation of ConvNet [2], a relatively classical convolutional neural network for synthetic vision, requires 435 million multiply-accumulate for VGA size images when using a 7×7 convolution kernel. Our goal in this paper is to demonstrate that it is possible to design deep neural networks (DNN) architectures that feature high throughput and low power while producing inference results that are close to the state of the art.

There are two strategies to lower power consumption: limit the amount of data to work on by using application-specific preprocessing and/or perform the computations with a low number of bits or a small set of values [3]. The extreme solution is to binarize all synaptic weights and activations, which eliminates multiplications once and for all, as proposed by [4], [5]. The loss of precision of these approaches is however quite high.

In this paper, we propose an FPGA architecture for ternary neural networks as a trade-off between inference accuracy, hardware resource utilization and power consumption.

II. Why Ternary Neural Networks?

There have been many recent works aiming at better utilizing the hardware resources to implement DNN. We can classify these works into two main categories.

The first one still uses floating-point arithmetic, but limits the number of possible values to a subset. Among representative works, [3] presents an ASIC architecture where they aim at limiting greatly the number and size of external accesses to memory. To that end, they prune the redundant connections and share weights by adequate training. As a consequence their design works on sparse matrices and uses small indexes to access arrays of weights. The approach proposed in [6] is somewhat different: at training time, is uses only specific combinations of activation and weight values. The pre-computed multiplication results are stored in lookup tables. They use ternary content addressable memory (TCAM) to ensure a fast and low-power search, but are therefore limited to ASIC.

The second category, the one we also follow, limits the number of bits for weight and/or activation values. The approach is not new, and for example [7] is an early paper studying the quality of the result as a function of the number of bit to code the weights. Using normal arithmetic, it is today admitted that using 6 or 7 bits does not significantly degrade the result of inference [8]. However, more extreme solutions have been advocated lately: binary [8], [9] (BNN) or ternary [10], [11] (TNN) encodings of the weights. Based on these training-

‡Institute of Engineering Univ. Grenoble Alpes

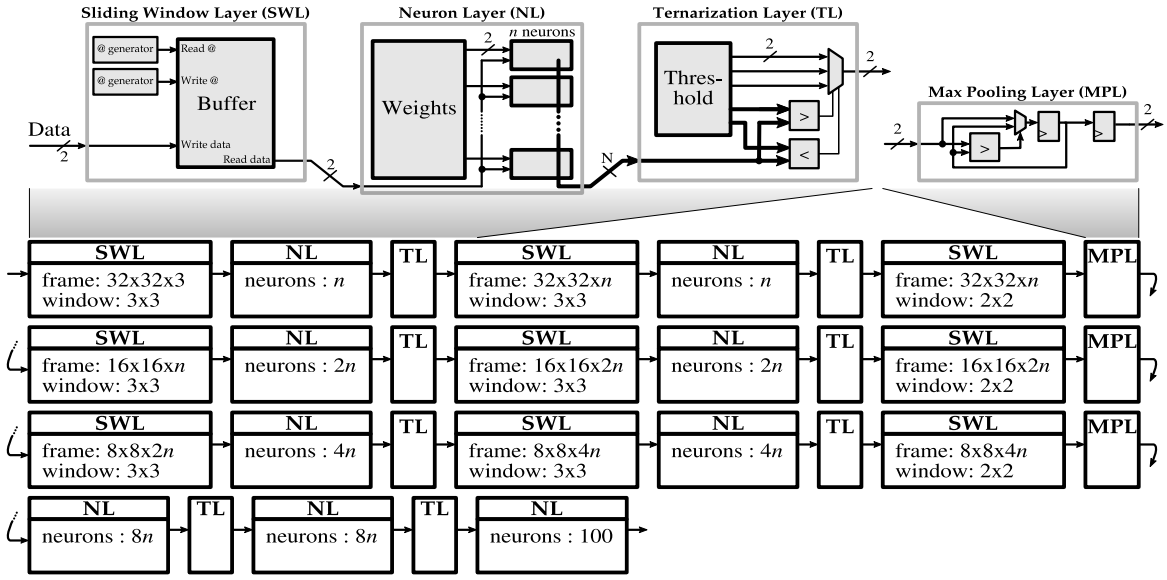

Fig. 1: CNN architecture overview

focused works, several hardware implementations have been proposed.

We first quickly review the most recent works focusing on binary weights. Andri *et al.* [12] implement a systolic array which processes each layer sequentially. Their ASIC implementation of BNN achieves state of the art area and energy efficiency, but because of the use of binary weights and activations, their error rate in applications is still fairly high. Umuroglu *et al.* [13] focus on high throughput FPGA implementations, and achieve highest reported throughput on a single FPGA chip. But again, binary trained networks are limited in use by the accuracy they achieve.

The very first work on TNN that we found is [14], a relatively imprecise short abstract from 1988 in which the authors study the adaptation of learning algorithms for ternary weights. Even though interesting from an historical perspective, the paper is quite lacunary. The first VLSI implementation of a TNN is reported in [15]. It also presents a training approach. However, the results are very difficult to interpret and to compare with the current technologies and state of the art. Since then, we have not found any detailed description of a hardware architecture for TNN while, according to [11], TNN can be fairly accurate when trained with the appropriate technique. To the best of our knowledge, [11] is the only recent work that makes reference to hardware implementations of TNN, FPGA and ASIC, but a) there is no detail whatsoever regarding the hardware architecture and its implementation, and b) they use ternary data obtained after a preprocessing step as primary input.

Given the accuracy achievable with TNN, we believe they are a sweet-spot between resource usage and precision, and that they have a place in applications for which power *vs.* accuracy trade-offs have to be made, for instance autonomous embedded devices or large-scale datacenters. The rest of this paper is dedicated to the presentation and evaluation of our TNN architecture and its FPGA implementation.

III. Proposed Architecture

We now detail our TNN architecture. We first give an overview of the architecture in terms of functional blocks, and then we describe each block thoroughly. We also detail how parallelism and area efficiency can be achieved by a proper pipeline design.

A. Overview

The large-scale ternary CNN pipeline VGG-like introduced in [8] is used as example throughout this paper. The architecture of our CNN is the following:

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

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 how the VGG-like pipeline is decomposed into layers connected in the form of a pipeline. All layers are independent from each other: they have their own state machine and image data is streamed through FIFO interfaces. For each layer type, we design a hardware block (hand-written VHDL) that is reused in the pipeline with different parameters. 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-only activations: the result of a neuron is a scalar and it must

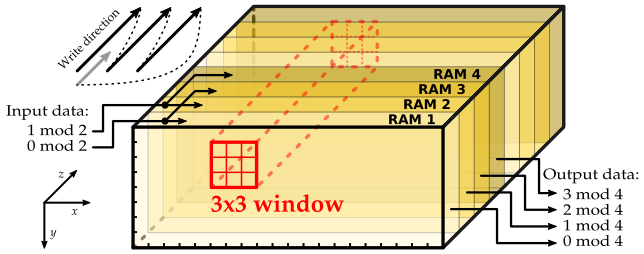

Fig. 2: Parallelism in Sliding Window Layer

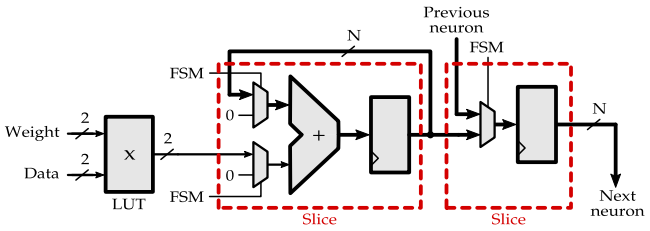

Fig. 3: Neuron implementation

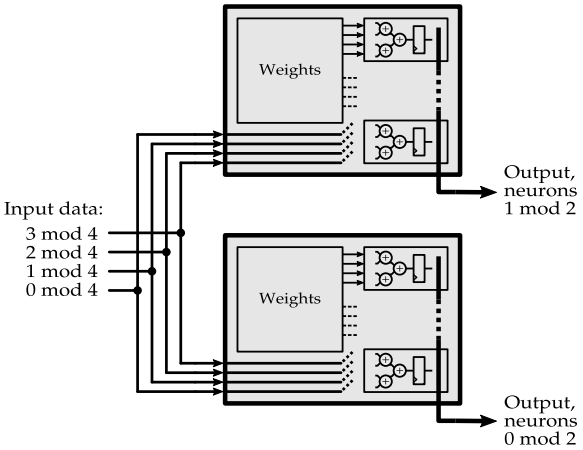

Fig. 4: Parallelism in Neuron Layer

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.

Throughout this paper, we use two networks with different dimensions, NN-64 and NN-128, respectively with $n = \{64, 128\}$. For instance according to Equation 1, the fifth NL has $2n$ neurons, hence the second SWL frame size has dimension $z = 128$ for NN-64 and $z = 256$ for NN-128. Our network NN-128 actually has same architecture than the network used in [8] except we increased the number of output neurons from 10 to 100 to enable using datasets with up to 100 classes.

Data channels between two blocks are implemented as small FIFOs (not shown for clarity) to compensate for the pipeline depth of the blocks and simplify their control flow. In our baseline implementation, each of these FIFOs transfers at most one activation value per clock cycle. This directly dictates the design throughput, in frames per second (fps). To increase throughput, parallelism is introduced in the layers that are responsible for the bottleneck. The corresponding FIFOs are widened and more activation values are transferred per clock cycle. How parallelism is implemented depends on the layer type and is explained in the following sections.

B. Sliding Window Layer

The Sliding Window Layer (SWL) is used for feeding either an NL or an MPL. To do so, it is highly configurable, partly at synthesis time and partly at runtime. Basically, the SWL acts as a buffer that stores enough data for the next layer to process in the order required by the following layer. To save memory resources, a SWL stores only a fraction of a frame and works as a ping-pong buffer. Both input and output sides can be parallelized to increase throughput. The output parallelism wanted defines the number of RAM blocks that are used to read data in parallel.

Figure 2 gives an example for an SWL configured with dimensions $20 \times 8 \times 8$ and window dimensions 3×3. Here, output parallelism is $P_o = 4$ using RAM1 to RAM4 and input parallelism is $P_i = 2$. Only 2 clock cycles are necessary to read an entire z-dimension of the window. Window size and step within all three directions can be set at run time. Input data is written in the following fashion: z, then x and finally y dimension. One should note that a P_i up to 4 could have been achieved thanks to the 4 RAM blocks that can be written at the same cycle.

C. Neuron Layer

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 multiply-accumulate operation on an internal register.

To extract the values out of the neuron accumulators as well as allow a compact placement in the FPGA, neurons are interconnected and form a scan chain, as proposed in [16]. This scan chain has its own registers, which enables to copy accumulator values and to extract them while accumulators perform the computations on the next frame data.

The architectural interest of using ternary values is illustrated Figure 3, which details the internal structure of the proposed neuron. The ternary multiplier requires two LUT4 which fit into one unique LUT6 on a Xilinx FPGA. Hence the neuron mainly consists of its two registers and associated ALUs and multiplexers. The ALUs and multiplexers are small enough so that they 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 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. As an example in the FPGA used in our experiments (433200 LUT6 and 3600 DSP cores), it is

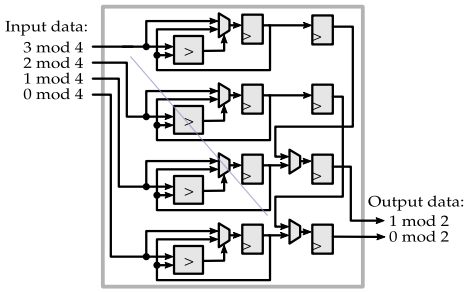

Fig. 5: Parallelism in Max Pooling Layer

possible to implement 5 to 6× more 12-bit ternary neurons (19 LUT6 each) than neurons based on DSP cores. Weight sparsity is intentionally not exploited. Indeed, compared to our very opimized FSM and neurons, the amount of per-neuron control needed to handle sparsity would come at an excessive cost in area and power.

Parallelism levels for input and output of the NL (P_i and P_o) are independent. Figure 4 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 a small adder tree before the accumulator. On the output side, all neurons are separated into P_o groups according to their index modulo P_o, each group having its own scan chain.

The weight memories are implemented either using RAM blocks or using the LUTRAM functionality of certain LUTs of the FPGA. For each neuron layer, the memory implementation is selected according to an arbitrary heuristic about the number of weights per neuron (W): LUTRAM is used when $W \leq 64$ or when $W \leq 128$ and $P_i \geq 4$ or when $W \leq 256$ and $P_i \geq 16$, 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.

D. Ternarization Layer

The Ternarization Layer (TL) is used to convert to ternary the scalar values produced by an NL. It acts as activation function as is often used in the literature. It is composed of one memory storing threshold values, two comparators and a multiplexer. There are two threshold values for each neuron of the previous neuron layer. Ternarization is performed the following way: if the result of a neuron is less than the first threshold, then the output is -1; if it is higher than the second threshold then the output is $+1$; and between the two thresholds the output is 0. Specifications of this step are closely linked to our training methodology, which is described in [11].

Parallelism level P is obtained by instantiating the ternarization block P times while sharing the same FSM. Instance index i handles data index i modulo P. Input and output parallelism levels of this layer are identical. In particular, this parallelism level is identical to the output parallelism of the previous NL.

E. Max Pooling Layer

The Max Pooling Layer (MPL) is used to find the maximum activation within a window. The window values are sent by

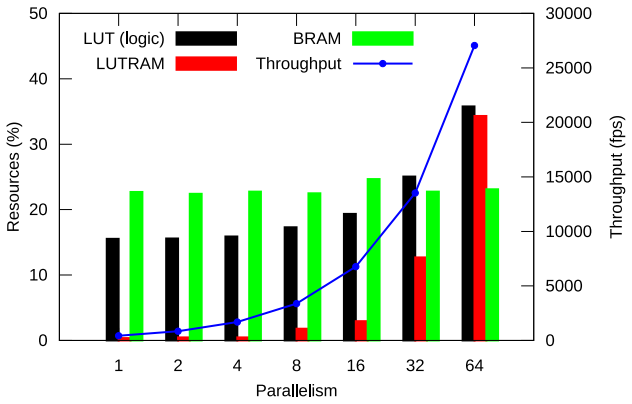

Fig. 6: NN-64: Area versus parallelism level

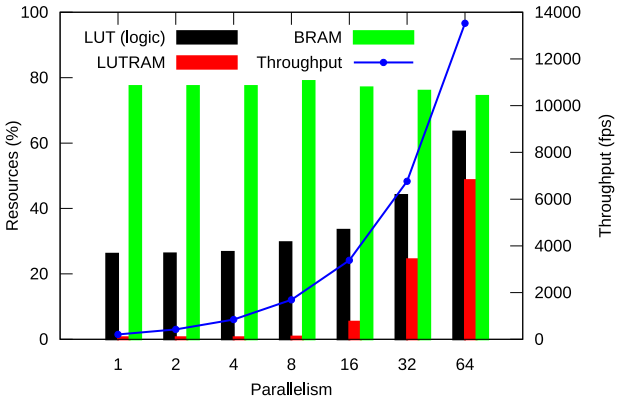

Fig. 7: NN-128: Area versus parallelism level

an SWL. Like other layers, both the input and the output can be parallelized. Figure 5 depicts an MPL with $P_i = 4$ and an $P_o = 2$. This parallelism configuration is for illustration only. Actually, is not particularly well suited to the typical case of a 2×2 sliding window feeding the MPL (see Figure 1 for the SWL configuration). There are 4 data items (2×2 window) at each 2-bit input ($0 \bmod 4$ to $3 \bmod 4$) and the number of cycles to empty the scan chain is 2 thanks to the output parallelism. Hence in this configuration, the scan chain is stalled half of the time. This output parallelism value is best used when the P_i reaches 8.

IV. Experiments and Results

In this section, we first describe our experimental setup. Then we present some characteristics of our TNN namely area *vs.* throughput and power consumption.

A. Experimental Setup

Experiments are performed on a VC709 FPGA board directly plugged in a PCI-Express slot of a workstation. This board is equipped with the Xilinx FPGA XC7VX690T. We highlight that the on-board 8 GB RAM is unused because only on-chip memory is used in our designs.

TABLE I: Neural network parallelization

NN size	Par. level	Parallelism per layer (in/out) $NL1$	$NL2$	$MPL1$	$NL3$	$NL4$	$MPL2$	$NL5$	$NL6$	$MPL3$	FPGA usage LUT (logic)	LUTRAM	BRAM 18k	Throughput (fps) Theory	Measure
64	1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	67300 (15.5%)	606 (0.34%)	667 (22.7%)	423.9	422.5
	2	1 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	67523 (15.6%)	786 (0.45%)	659 (22.4%)	847.7	845.1
	4	1 / 1	4 / 1	1 / 1	1 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1	68830 (15.9%)	794 (0.46%)	669 (22.8%)	1695.4	1690.3
	8	1 / 1	8 / 1	1 / 1	2 / 1	4 / 1	1 / 1	1 / 1	2 / 1	1 / 1	74940 (17.3%)	3138 (1.80%)	661 (22.5%)	3390.8	3381.2
	16	1 / 2	16 / 2	2 / 1	4 / 1	8 / 1	1 / 1	2 / 1	4 / 1	1 / 1	83789 (19.3%)	5118 (2.94%)	725 (24.7%)	6781.7	6763.3
	32	2 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	4 / 1	8 / 1	1 / 1	108474 (25.0%)	22110 (12.7%)	669 (22.8%)	13563.4	13525.3
	64	3 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	154929 (35.8%)	59762 (34.3%)	679 (23.1%)	27126.7	27042.9
128	1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	113179 (26.1%)	1046 (0.60%)	2276 (77.4%)	211.9	211.3
	2	1 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	113650 (26.2%)	1050 (0.60%)	2276 (77.4%)	423.9	422.6
	4	1 / 1	4 / 1	1 / 1	1 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1	115778 (26.7%)	1058 (0.61%)	2276 (77.4%)	847.7	845.3
	8	1 / 1	8 / 1	1 / 1	2 / 1	4 / 1	1 / 1	1 / 1	2 / 1	1 / 1	128499 (29.7%)	1434 (0.82%)	2321 (78.9%)	1695.4	1690.8
	16	1 / 2	16 / 2	2 / 1	4 / 1	8 / 1	1 / 1	2 / 1	4 / 1	1 / 1	144983 (33.4%)	9362 (5.37%)	2264 (77.0%)	3390.8	3382.1
	32	1 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	4 / 1	8 / 1	1 / 1	191192 (44.1%)	42582 (24.4%)	2234 (76.0%)	6781.7	6764.0
	64	2 / 8	64 / 8	8 / 2	16 / 4	32 / 4	4 / 1	8 / 2	16 / 2	2 / 1	275042 (63.4%)	84810 (48.6%)	2187 (74.4%)	13563.4	13525.8

The RIFFA framework [17] is used as PCI-Express communication interface with the computer. All designs run at 250 MHz clock frequency, which is the frequency generated by the embedded PCI-Express endpoint. Power measurements are performed with on-board PMBus, an I^2C bus dedicated to that purpose. We added a custom and independent UART-to-I^2C bridge to our designs to read power values without interfering with PCI-Express data transfers.

The networks used are NN-64 and NN-128. Experiments are conducted with well-known datasets CIFAR10 [18], GTSRB [19] and SVHN [20]. In all datasets, all images have a size of 32×32 pixels on 3 color channels. As also performed in the related works, we pre-process the images before sending them to the FPGA: Global Contrast Normalization followed by LeCun LCN is used for datasets GTSRB and SVHN, and normalization and ZCA whitening is used for dataset CIFAR10. We use 8-bit quantization per pixel and per color channel.

Demonstration materials (bitstreams and communication software) are available at the team webpage[1]. It allows to reproduce the paper results.

B. Area and throughput

In our base design, all layers of the network receive and transmit at most one ternary value per clock cycle. In particular, inside neuron layers, all neurons perform in parallel one multiply-accumulate operation per clock cycle. To increase the design throughput (in frames/second), we parallelize the layers that are the bottleneck of the architecture. Table I presents the parallelism levels applied to neurons and to max pooling layers. Corresponding parallelism levels on ternarization and window layers result directly. Layers are named $NL1$ to $NL9$ for Neuron Layers and $MPL1$ to $MPL3$ for Max Pooling Layers. The unit used is the number of values transferred per clock cycle in the input and output ports of these layers. The input and output of the pipeline are not bottlenecks and are not parallelized.

We highlight that gaining a $2\times$ speedup does not necessarily require $2\times$ more hardware resources. This is illustrated in Figures 6 and 7 (resources not on the same scale for clarity). Indeed, all layers have different execution times, and only the most demanding layers are parallelized, which may be only a small fraction of the design resources. Similarly, inside the neurons themselves, only the size of the adder tree increases, not the entire neuron.

Without parallelism, all weight memory banks are implemented within dedicated block RAM (BRAM) resources of the FPGA. Adding parallelism increases the amount of data that these memory banks have to produce at each clock cycle. Even though the storage needs (in bits) does not increase, the BRAM requirements increase to implement the required output width. To avoid BRAM shortage, the LUTRAM resources are used when parallelism is high enough and frame size is low enough in the neuron layers.

There are two limits to the achievable parallelism with our design. The first is due to hardware resources: NN-128 with parallelism level 128 does not fit in our FPGA. The second is due to our parallelization technique for the $SWLs$: the maximum parallelism level achievable is the dimension of the image in the z dimension, which is directly related to the number of neurons. This is why with our current design, the maximum parallelism level for NN-64 is 64. Otherwise, the available hardware resources would allow parallelism level 128, with corresponding throughput $54k$ fps.

C. Power consumption

We measure the power consumption using the core $1\ V$ power supply rail of the FPGA, since our designs fit entirely inside the FPGA. To confirm that this way of doing the measure is correct, we also monitored the global board power (all supply rails measured through PMBus) and observed that it is higher than the core $1\ V$ rail by a rather constant $4.5\ W$ for all designs.

Results are presented in Figure 8. The figures related to NN-64 and NN-128 form two very distinct groups. For each of the two NN sizes, the power consumption is approximately a linear function of the design throughput, and varies little between datasets. This is due to our FPGA implementation not exploiting dataset sparsity (zero-activations and zero-weights) to reduce design activity. When neuron weights are packed inside large RAM banks, it is not possible to inhibit RAM read for selected positions.

[1] http://tima.imag.fr/sls/research-projects/tnn-fpga-implementation/

TABLE II: Comparison with related works

Dataset	Authors	Plat. name	NN Arch.	Input quant.	Weight quant.	%err	fps	Power (W)	fps/W	Target
CIFAR10	**This work**		NN-64	3 ch, 8 bits	2 bits	**13.29**	**27043**	6.80	**3976**	VC709
	This work		NN-128	3 ch, 8 bits	2 bits	**10.61**	**13526**	13.64	**992**	VC709
	[13]	FINN	NN-64	24 bits	1 bit	19.90	21900	3.6	6080	ZC706
	[21]	BCNN	NN-128	3 ch, 6 bits	1 bit	12.20	6218	8.2	758	xc7vx690t
	[22]	BNN	NN-128	3 ch, 20 bits	1 bit	11.32	168	4.7	35.8	ZedBoard
	[11]	TNN	NN-128	12 ch, 2 bits	2 bits	12.11	1695	9.58	178	VC709
SVHN	**This work**		NN-64	3 ch, 8 bits	2 bits	**2.40**	**27043**	7.08	**3820**	VC709
	This work		NN-128	3 ch, 8 bits	2 bits	**2.30**	**13526**	13.70	**987**	VC709
	[13]	FINN	NN-64	24 bits	1 bit	5.10	21900	3.6	6080	ZC706
	[11]	TNN	NN-64	12 ch, 2 bits	2 bits	2.73	3390	4.8	709	VC709
GTSRB	**This work**		NN-64	3 ch, 8 bits	2 bits	**1.05**	**27043**	6.64	**4073**	VC709
	This work		NN-128	3 ch, 8 bits	2 bits	**0.80**	**13526**	12.57	**1076**	VC709
	[11]	TNN	NN-128	12 ch, 2 bits	2 bits	0.98	1695	9.58	178	VC709

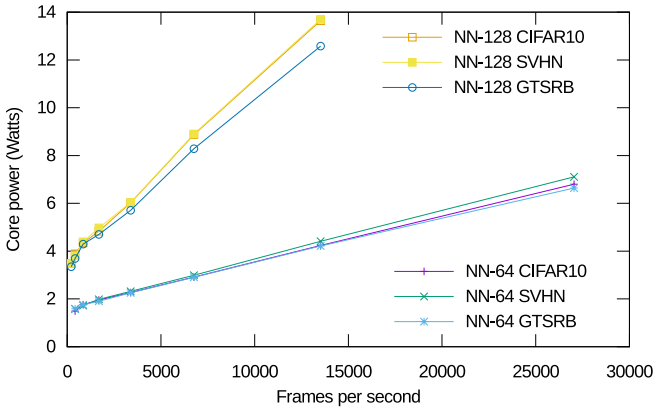

Fig. 8: Power versus framerate

For low throughput (less than $4000\,\text{fps}$), the NN-128 power is roughly $2\times$ the NN-64 power. This was expected because most of the time only layer $NL2$ runs while other layers wait, so the idle power dominates. But as throughput increases, this difference increases up to $3\times$. Actually for one frame, NN-128 performs 618 million multiply-accumulate operations while NN-64 performs 155 million, so a difference of $4\times$ instead of $3\times$ was expected. This is due to our implementation of parallelism inside neurons: for a given framerate, the neurons in NN-128 have to be parallelized twice more than in NN-64, but this impacts only the leaves of the adder tree and not the accumulator and scan chain.

We extrapolate the idle power as the intersection with the y-axis, and we obtain around $1.8\ W$ for NN-64 and $4\ W$ for NN-128. The possible sources are the static power, the clocks and the IPs related to the PCI-Express interface. According to the synthesis tool estimations (Vivado 2015.3) for parallelism level 64, the highest contributors to the idle power are the clocks ($2.2\ W$ and $3.2\ W$) and the static power ($0.5\ W$ and $0.6\ W$). The power related to PCI-Express can be high (up to $2.8\ W$) but, assuming that it scales according to the ratio of the maximum throughput, that communication interface should account for only $0.3\ W$. So Vivado values are over-estimated for NN-64, but rather close for NN-128.

V. Related work

Our results are presented in Table II, along with results from others FPGA-based works using the same datasets. For each dataset and neural network architecture, our platform provides the highest throughput and it outperforms accuracy of related works.

Umuroglu *et al.* [13] propose FINN, an FPGA implementation of NN-64 with binary weights on board ZC706. Their design can classify the datasets CIFAR10 and SVHN at a throughput of $21\,900\,\text{fps}$. Our raw processing speed (frames per second) is higher than their, but this is exclusively due to our higher frequency (250 MHz instead of 200 MHz). This difference may be linked to us using a higher-performance FPGA technology (Virtex-7 where they use Zynq-7000). Our power efficiency (throughput per watt) is lower than their by $33-37\%$. Indeed, using ternary weights makes neuron operations a little more complex than with binary weights, which contributes to power. However, we are using a higher-performance FPGA technology and PCI-Express communication interface, and our FPGA is largely oversized for NN-64. Actually, our design would fit in their board. The strongest difference is accuracy: our error rate is only 13.29% for CIFAR10 and 2.40% for SVHN, where they have 19.9% and 5.1%, respectively. Given how difficult it usually is to reduce error rate, this shows superiority of ternary over binary-only weights.

Zhao *et al.* [22] propose BNN, an FPGA implementation of NN-128 with binary weights on board ZedBoard. They focus on accelerating the neural network in a very reduced FPGA, so the resulting throughput is very low. All weights don't fit in the FPGA, so they have to transfer them from the external on-board DDR memory. Moreover, the FPGA is so small that the power consumption of the on-chip processor subsystem dominates. Their accuracy is also notably lower than ours with a neural network of identical size. Overall, the resulting efficiency and accuracy is still interesting as an accelerator for the small on-chip processors, but it is far from related works who focus on performance per watt and/or accuracy.

Li *et al.* [21] propose BCNN, an FPGA implementation of NN-128 with binary weights on FPGA xc7vx690t (same chip than our board VC709). Their design is not entirely binary:

they use 2-bit weights in the first neuron layer. They use Vivado HLS to generate their design and their results are the Vivado-estimated execution times and power consumption. The communication interface is unknown. Their HLS-generated design runs at 90 MHz and processes the dataset CIFAR10 at 6218 fps. With our hand-written RTL, our frequency is higher (250 MHz) and our design is notably faster. But even if they used our frequency, their throughput would be only 17272 fps which is still much lower than our platform. Their design is presented as a 7.663 TOP/s accelerator (with multiply and accumulate counted as different operations). We have 4.19 TMAC/s for NN-64 and 8.36 TMAC/s for NN-128, hence respectively 8.38 TOP/s and 16.72 TOP/s, an improvement of respectively 9.4% and 118% over their design.

In [11], Alemdar *et al.* propose ternary neural networks similar to our NN-64 and NN-128, but with 12-channel ternary input. Only a fixed parallelism of 8× is used in their FPGA version and the power consumption is based on pessimistic estimations. Our results bring a significant improvement over their work: our designs are more power-efficient with about 6× better throughput per watt, and error rate is lower.

VI. Conclusion

Thanks to their very good performance in solving inference problems when properly trained, TNN are good candidates for efficient hardware implementations. In this work, we have designed a set of blocks that can be stacked and pipelined to build arbitrarily complex convolutional neural networks making use of ternary values for weights and/or activations. The ternary nature of the network leads to significantly better inference results than binary NN, for an increase in resource usage and power affordable in many applications. The resulting designs feature high density, high throughput and low power. With no impact on accuracy, parallelism levels can be tuned to span a broad range of power-area-throughput trade-offs.

Acknowledgment

This project is being funded in part by Grenoble Alpes Métropole through the Nano2017 Esprit project. The authors would like to thank Olivier Menut from ST Microelectronics for his valuable inputs and continuous support.

References

[1] O. Temam, "The rebirth of neural networks," Keynote speach at the International Symposium on Computer Architecture, 2010.

[2] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 257–260.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2016.

[4] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Advances in Neural Information Processing Systems*, 2015, pp. 1117–1125.

[5] M. Kim and P. Smaragdis, "Bitwise neural networks," in *International Conference on Machine Learning, Workshop on Resource-Efficient Machine Learning*, 2015.

[6] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *Proceedings of the 2017 IEEE/ACM Conference on Design, Automation and Test in Europe*, 2017, pp. 1775–1780.

[7] K. Asanovic and N. Morgan, "Experimental determination of precision requirements for back-propagation training of artificial neural networks," in *Proceedings 2nd International Conference on Microelectronics for Neural Networks*, 1991.

[8] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[9] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[10] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[11] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," in *30th International Joint Conference on Neural Networks*, 2017.

[12] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *IEEE Computer Society Annual Symposium on VLSI*, 2016, pp. 236–241.

[13] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," *CoRR*, vol. abs/1612.07119, 2016. [Online]. Available: http://arxiv.org/abs/1612.07119

[14] T.-D. Chiueh and R. M. Goodman, "Learning algorithms for neural networks with ternary weights," *Neural Networks*, vol. 1, p. 166, 1988.

[15] B. Hoskins, M. Haskard, and G. Curkowicz, "A vlsi implementation of multi-layer neural network with ternary activation functions and limited integer weights," in *1995 20th International Conference on Microelectronics*, 1995, pp. 843–846.

[16] P. Škoda, T. Lipić, Ã. Srp, B. M. Rogina, K. Skala, and F. Vajda, "Implementation framework for artificial neural networks on fpga," in *2011 Proceedings of the 34th International Convention MIPRO*, May 2011, pp. 274–278.

[17] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 4, Sep. 2015.

[18] A. Krizhevsky, "Learning multiple layers of features from tiny images," Toronto University, Tech. Rep., 2009.

[19] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," in *International Joint Conference on Neural Networks*, 2011.

[20] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[21] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, "A 7.663-tops 8.2-w energy-efficient fpga accelerator for binary convolutional neural networks," *arXiv:1702.06392*, Feb 2017.

[22] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 15–24.