Low- and Mixed-Precision Inference Accelerators

Maarten J. Molendijk, Floran A. M. de Putter, and Henk Corporaal

1 Introduction

Neural Networks can solve increasingly more complex tasks in fields such as Computer Vision (CV) and Natural Language Processing (NLP). While these Neural Networks can perform complex tasks with increasingly higher accuracy, the sheer size of these networks often prevents deployment on edge devices that have limited memory capacity and are subject to severe energy constraints. To overcome the issues preventing the deployment of neural networks onto edge devices, efforts toward reducing the model size and reducing the computational costs have been made. These efforts are most often focused on either the algorithmic side, tailoring the neural network and its properties, or on the hardware side, creating efficient system designs and arithmetic circuitry.

In an effort to reduce the computational cost and model size of neural networks, several approaches are taken. One of these approaches is to automate the synthesis of the neural network architecture while taking into account the hardware resources, this is called hardware-aware neural architecture search (NAS) [23, 28]. Another way to increase the energy efficiency is by compressing the model size, applying either quantization [10] or pruning [4].

In parallel to research on model compression, research has been performed on creating highly specialized hardware that exploits the opportunities arising from model compression. ASICs that support neural network inference for operand precisions as low as 1 bit exploit the advantages extreme quantization brings: low memory size and bandwidth and simplified compute logic. In the pursuit of the most energy-efficient hardware design, several design choices regarding memory

M. J. Molendijk (✉) · F. A. M. de Putter · H. Corporaal
Eindhoven Artificial Intelligence Systems Institute and PARsE lab, Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: m.j.molendijk@tue.nl; f.a.m.d.putter@tue.nl; h.corporaal@tue.nl

S. Pasricha, M. Shafique (eds.), *Embedded Machine Learning for Cyber-Physical, IoT, and Edge Computing*, https://doi.org/10.1007/978-3-031-19568-6_3

hierarchy, hardware parallelization of operations, and data-flow are made that impact both the ASIC's efficiency and its flexibility.

For instance, many architectures have a fixed datapath; the movement of the data is fixed at design time which can impose limitations on the layer types, channel dimensions, and kernel dimensions. Furthermore, these architectures typically have limited programmability and configurability, which restricts the execution schedules that can (efficiently) be run.

In this chapter, a look will be taken at several different approaches of neural network accelerators specifically designed for inference with very low-precision operands. The efficiency (and origin thereof) of the architectures will be analyzed and compared to the flexibility that these architectures offer.

In short, the contributions of this work are:

- Overview of state-of-the-art low- and mixed-precision neural network accelerators, in Sect. 3.
- Analysis on the trade-off between the flexibility and the energy efficiency of accelerators, in Sect. 4.

The remainder of this chapter is structured as follows: in Sect. 2, background information on neural network architecture and quantization is presented. Thereafter, in Sect. 3, the low- and mixed-precision accelerators are presented and a comparison is presented in Sect. 4. Section 5 concludes this chapter.

2 Background: Extreme Quantization and Network Variety

Modern neural network architectures consist of many different layers with millions of parameters and operations. The storage required to store all parameters and features is not in line with the storage capacity typically found on embedded devices, leading to costly off-chip memory accesses. Next to the memory and bandwidth limitations, computational costs for full-precision (`float32`) operations require power-hungry compute blocks that quickly overtax the energy requirements of the embedded devices. To reduce both the computational cost and the cost of data access and transport, quantization can be applied.

Quantization leads to lower precision parameters and therefore induces information loss. Naturally, when weights and activations can represent fewer distinct values, the representational capabilities of the network decrease. This decrease may create an accuracy loss. In [14], Gholami et al. show, however, that quantization down to `integer8` can be done without significant accuracy loss. But even when quantizing down to `integer8`, the memory requirements can still overtax the memory capacity typically found in embedded systems. Therefore, research has been done on extreme quantization, i.e., quantization below 8-bit precision.

In the next subsection, several frequently utilized building blocks for convolutional neural networks (CNNs) are listed. Thereafter, in Sects. 2.2 and 2.3, two

forms of extreme quantization, namely `binary` and `ternary` quantization, are discussed. Finally, in Sect. 2.4, the need for mixed-precision is considered.

2.1 Neural Network Architecture Design Space

Neural network architectures have a great variety in the type of layers, the size of these layers, and the connectivity between these layers. Furthermore, with mixed-precision architectures, the precision can also be chosen on a per-layer basis. An example network is shown in Fig. 1. Some common building blocks are listed below:

- Convolutional Layer
- Fully connected Layer
- Depth-wise Convolutional Layer
- Residual Addition
- Requantization
- Pooling

The working horse of CNNs is the convolutional layer. Between different convolutional layers, there can be variety in the kernel size, number of input feature maps, output feature maps, etc. In Fig. 2, the different parameters of a convolutional layer are presented. These parameters will later on prove to be an important basis for designing efficient hardware. The goal of Sect. 3 is to show how these network parameters relate to hardware design, hardware efficiency, and hardware flexibility.

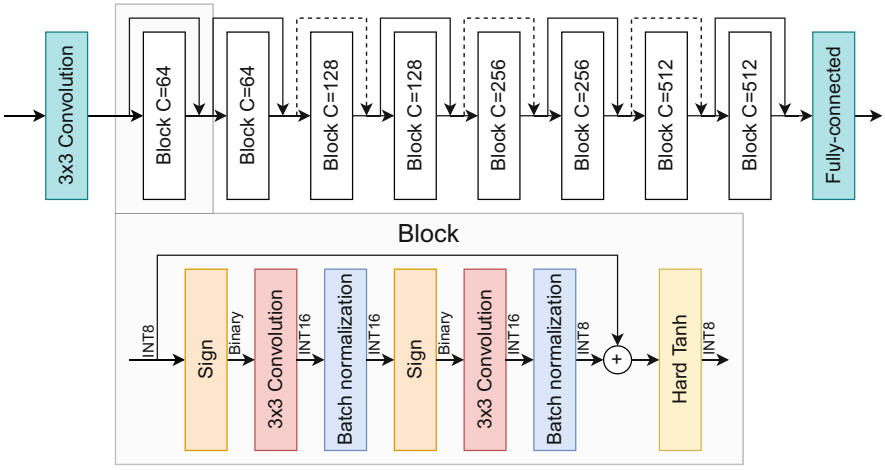

Fig. 1 Binary ResNet-18, an exemplary network containing several different building blocks and using different operand precisions. Note that the first layer and the "skip connections" have a higher than binary precision. Furthermore, the number of channels C can differ between the building blocks

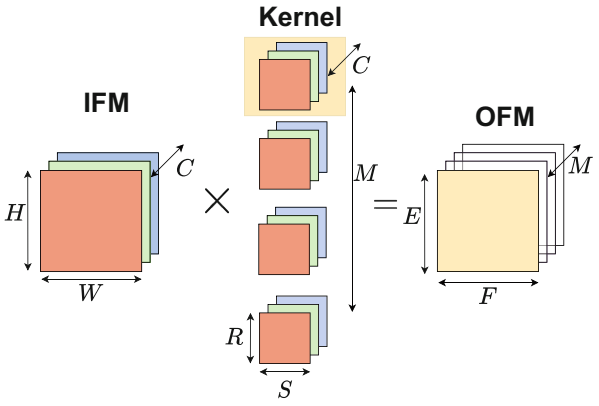

Fig. 2 A convolutional layer can vary in different ways. The Input Feature Map (IFM) has height H and width W and contains C channels; the Output Feature Map (OFM) has height E, width F, and M channels; and the kernel has height R and width S. Between different layers and different networks, these parameters vary

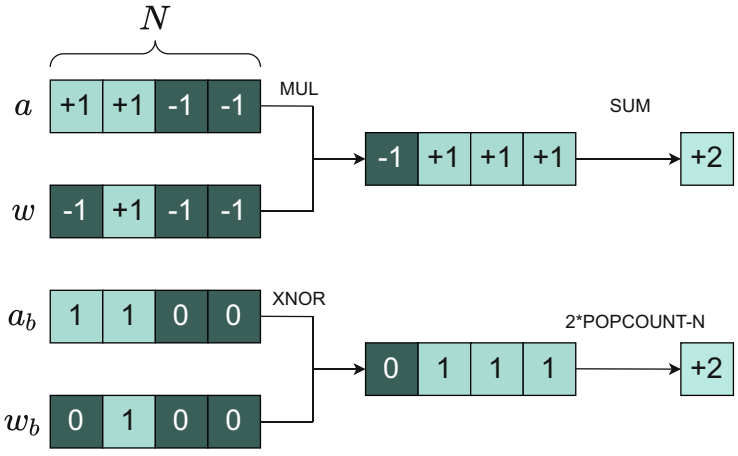

Fig. 3 Simplified arithmetic circuitry as a consequence of binary quantization. The top displays the default multiplication, while the bottom displays how binary quantization can replace it with `XNOR` and `popcount`. N is the number of bits of the input vector

2.2 *Binary Quantization*

On the extreme end of quantization is binary quantization. Binary quantization restricts both weights and activations to binary values. This means that the activations $a \in \{-1, +1\}$ and weights $w \in \{-1, +1\}$. Reduction of the precision of the operands introduces several advantages. First of all, the required storage capacity and bandwidth on the device are drastically reduced, compared to `float32` by a factor of 32. Furthermore, the Multiply–Accumulate (`MAC`) operation, involving expensive multiplication hardware, can be replaced by the much more simple and cheaper `XNOR` and `popcount` operations [21]. An example of this simplified arithmetic is shown in Fig. 3.

The output value of the `popcount` produces a value that needs to be stored with a larger bit-width compared to the binary input value, e.g., `integer16`. Therefore, to feed the outputs into a new layer, the nonlinear activation function needs to requantize the values back to the binary bit-width. For this purpose, the *sign* function is used. The quantized operand can be derived from its unquantized form as follows:

$$X_{quant} = Sign(X) = \begin{cases} +1 & \text{if } X \geq 0 \\ -1 & \text{if } X < 0 \end{cases} \tag{1}$$

This function is non-differentiable; for training, a Straight-Through Estimator (STE) [3] can be used that passes gradients as is. By employing an STE, gradient descent is possible, and binary neural networks can be trained.

2.3 Ternary Quantization

Compared to binary quantization, ternary quantization allows for only one—*albeit very important*—extra value to be represented in the operands, namely zero. Ternary networks therefore have operands $w, a \in \{-1, 0, +1\}$ called *trits*. Next to the increased representational capabilities, the ability to represent zero also solves some issues found in binary networks. First of all, zero padding is not possible in binary networks since it lacks the ability to represent zero, and this is most often solved by employing on–off padding. Furthermore, the ability to represent zero introduces the capability to exploit sparsity, i.e., skipping computations when either the weights or activation is zero. As will be seen later on, this can have a significant impact on the efficiency of the computational hardware if the network itself is sparse.

The arithmetic circuitry required to perform multiply–accumulate (`MAC`) operations on ternary operands is very similar to that of binary networks. The `MAC` operation can be replaced by a `Gated-XNOR` [8] (`XNOR` and `AND` gates) combined with two `popcount` modules, one for the +1s and one for the -1s. The arithmetic is shown in Fig. 4.

Again, as with the binary `popcount`, the final result has a higher bit-width and needs to be requantized before being fed into the next layer. The quantization function typically uses a symmetric threshold value Δ:

$$X_{quant} = Ternarize(X) = \begin{cases} +1 & \text{if } X > \Delta \\ 0 & \text{if } |X| \leq \Delta \\ -1 & \text{if } X < -\Delta \end{cases} \tag{2}$$

During computation, each trit occupies 2 bits. However, this is a wasteful way to store them since theoretically $log_2(3) = 1.58$ bits are needed for each trit. Muller et

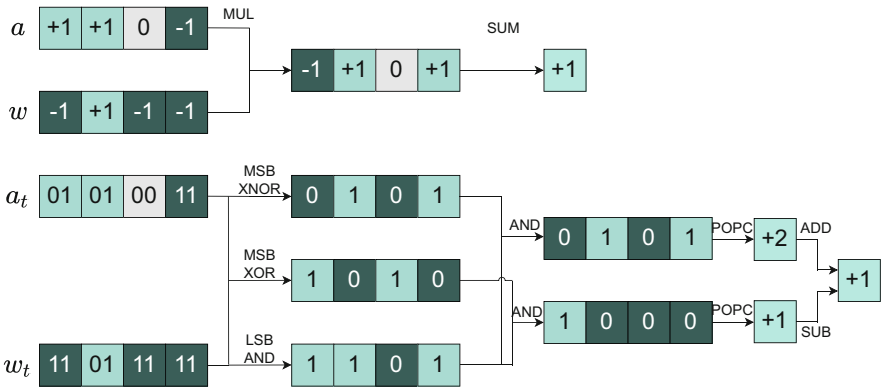

Fig. 4 Simplified arithmetic circuitry as a consequence of ternary quantization. The top displays the default multiplication, while the bottom displays the ternary simplified variant. Note that the ternary variant needs two popcount modules (one to count $+1$s and one to count -1s)

al. [18] derived an efficient mapping, compressing 5 trits into 8 bits, yielding a total storage of 1.6 bit per trit, close to the theoretical lower bound.

2.4 Mixed-Precision

Despite all the advantages of extreme quantization, binary and ternary quantization often induce severe accuracy loss, especially on more complex tasks. For example, there is a large gap in accuracy when comparing `integer8` quantization to `binary` and `ternary` [5, 10]. Moreover, the accuracy loss that is induced differs per layer in the network [11]; i.e., some layers are more resilient to extreme quantization than others. Therefore, a combination of different precisions in a per-layer fashion can give a good balance between accuracy and efficiency.

An overview of different data precisions typically found in neural network architectures is given in Fig. 5. The figure shows the width of different data formats and how the bits are allocated. Next to the data format, the range is displayed, i.e., the minimum and maximum value that can be attained using that data format. Note that the range for the floating-point number only displays the positive numbers, while it is able to represent negative numbers using the sign bit.

In the past, `float32` was used as the *de facto* standard for neural networks. Gradually, movements toward smaller data types like `float16` were made to save on storage and computational cost. Moreover, it was found that the dynamic range of the data types has a larger impact on the accuracy than the relative precision, leading to the creation of `bfloat16` [27] (Brain Floating Point) and `tf32` [15] (TensorFloat32), both trading off relative precision in favor of increased range. Using `integer8` precision completely gets rid of the expensive floating-

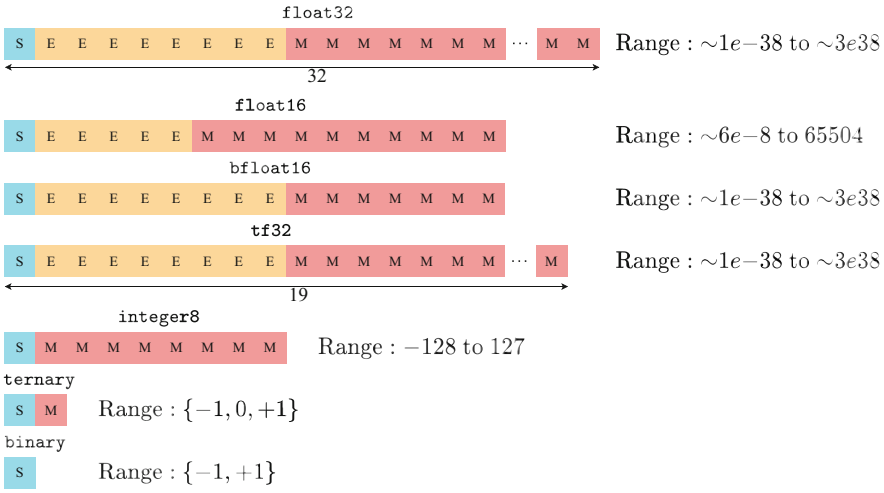

Fig. 5 Breakdown of the bit usage inside data formats commonly used in neural networks. S is sign, E is exponent, and M is mantissa. Floating point data formats specifically for neural networks prefer higher range over more precision

point arithmetic, vastly increasing the throughput and energy efficiency, with at the extreme end `binary` and `ternary` quantization.

By the nature of floating-point arithmetic units, exponents are added up together, while the mantissa bits are multiplied. Therefore, `bf16`, which has 3 less mantissa bits compared to `float16`, will have a *two times* smaller footprint, while compared to `float32` it will even have an *eight times* smaller area. This is because the area of the multiplier unit is roughly proportional to the square of the mantissa bits. In Sect. 3, accelerators that support `integer8` (which can also be used for fixed-point arithmetic), `binary`, and `ternary` precisions are discussed.

3 Accelerators for Low- and Mixed-Precision Inference

With the aim to get the energy per `MAC` operation as low as possible, several accelerators specifically designed for low-precision inference have been created. Some of these architectures also support different precisions on the same platform. The accelerators can be split into two groups: fully digital accelerators and mixed-signal/analog compute-in-memory (CIM) approaches. Although state-of-the-art CIM architectures [2, 26] and mixed-signal implementations [25] have the potential to achieve high energy efficiency, they also introduce new unique challenges. These challenges include longer design time and chip-to-chip variation due to CMOS process variation, which makes it more difficult to benchmark the actual performance of such a design, and no programmability, making it more difficult to

use the accelerator. The further focus in this chapter will therefore be solely on fully digital implementations.

First, characterization criteria that are important to embedded neural network accelerators will be established, and these include key performance indicators to measure the efficiency (in both area and energy) of the architecture. Furthermore, the basis for the flexibility analysis is laid out, based on the robustness of architectures against different layer types, dimensionality, and precisions. Thereafter, five state-of-the-art digital inference accelerators will be discussed.

3.1 Characterization Criteria

The accelerators will be characterized according to both their flexibility and their energy efficiency. Defining flexibility as a quantitative metric can often be cumbersome, although some recent effort toward bringing structure has been made [12]. Next to the flexibility aspects, the most important quantitative performance evaluation criteria for neural network inference accelerators will be listed and motivated.

3.1.1 Flexibility

Before the characterization criteria are established, a closer look is taken at the nature of a convolution kernel. A convolution kernel can be described by 6 nested for-loops (7 when adding the batch dimension), and an exemplary schedule is shown in Listing 1. It is assumed that the target application is image processing, i.e., inputs are referred to as *pixels*.

In Listing 1, the for-loops are arranged in a so-called *output stationary* way, i.e., one output pixel is calculated as soon as possible. In other words, all calculations that are needed for a set of output pixels are performed before moving to the next set of output pixels. This avoids having to store and reload partially calculated output pixels.

```
for h in [0, H - R + 1]:                          Output feature map height
  for w in [0, W - S + 1]:                        Output feature map width
    for m in [0, M]:                              Output channels
      acc = bias[m]
      for c in [0, C]:                            Input channels
        for r in [0, R]:                          Kernel height
          for s in [0, S]:                        Kernel width
            acc += ifm[h + r][w + s][c] * weights[n][r][s][m]
      ofm[h][w][m] = acc
```

Listing 1 A naive convolutional layer with output stationary schedule and a stride of 1; *acc* is the temporary accumulated value, for simplicity, the IFM is assumed to be padded. The loop iterators are visualized in Fig. 2

Loop nest optimization (LNO) can be performed to increase data locality. Two important techniques, part of LNO, are loop tiling (also known as loop blocking), where a loop is split up into an inner and an outer loop, and loop interchange, where two loops are swapped in hierarchy level. The problem of finding the best combination of the two is called the **temporal mapping** problem (i.e., finding the best execution schedule). The temporal mapping greatly influences the number of memory accesses needed and therefore indirectly greatly influences the energy efficiency of an accelerator.

Next to the temporal mapping, the operations performed in the convolutional kernel can also be parallelized in hardware. The problem of finding the optimal parallelization dimensions is called the **spatial mapping** problem. Using optimal spatial mapping can increase data reuse in hardware and reduce memory traffic. A good example of this is the mapping on a systolic array. It is important to note that the spatial mapping should be carefully chosen, as it imposes constraints on the dimensions being parallelized.

Hardware parallelization over a dimension is called vectorization. Vectorization over any of the dimensions given in Fig. 2 will be denoted as the vectorization factor v_{param}, where param can be any of the dimensions in Fig. 2. For instance, when parallelizing over the C dimension using a vectorization factor of 32, it is denoted as $v_C = 32$. This vectorization factor also implies constraints: any convolutional network layer that does not have an input channel multiple of 32 will not run at 100% utilization. There will be a trade-off between the vectorization factor and the flexibility with respect to convolutional layers with certain layer dimensions being able to run at full utilization.

Research has been done on structurally exploring the temporal and spatial mapping design space [20, 29]. Most recently, the ZigZag framework [17] has been published aiming to fully co-design temporal mapping with hardware architecture finding the best spatial and temporal mappings available.

One other facet of flexibility is **programmability**. Programmability allows running different, possibly even non-DNN workloads on the accelerators. Especially, high-level programmability increases the usability of the device since it allows the workload to be configured while programming it via a high-level language, requiring less knowledge about the hardware implementation from a user perspective.

3.1.2 Performance Characteristics

To compare the performance of the several accelerators reviewed, some quantitative metrics that reflect the performance of the accelerator are established. First of all, the most widely promoted metric to compare accelerators is to compare the energy efficiency, defined as the energy per operation (either [pJ/op] or inversely in [TOPS/W]).

Secondly, the memory capacity plays an important role in the efficiency of the accelerator. Since off-chip memory access energy is much larger than the energy needed to compute, off-chip memory access should be avoided at all costs.

More on-chip memory means fewer external memory accesses, benefiting energy consumption. Two different ways to implement on-chip memory are SRAM and Standard-Cell Memory (SCM). While SRAM has a much higher memory density, it is less efficient in terms of energy usage for smaller sizes compared to SCM. Especially, when applying voltage–frequency scaling, the SCM can be scaled to a much lower voltage than SRAM. Therefore, SCM tends to be a popular choice to keep down the energy cost of the total system while sacrificing area and storage capacity. Other important metrics are throughput [GOPS] and area efficiency [GOPS/mm^2].

3.2 Five Low- and Mixed-Precision Accelerators Reviewed

Five state-of-the-art accelerators will be discussed and compared against one another. These accelerators were chosen because of their support for very low precisions (i.e., `binary` or `ternary`). These accelerators are:

- *XNOR Neural Engine* [6] is a binary neural network accelerator built into a programmable microcontroller unit. A full system on a chip (SoC), implemented in 22-nm technology, is presented including the accelerator, RISC host processor, and peripherals.
- *ChewBaccaNN* [1] is an architecture for binary neural network inference that exploits efficient data reuse by co-designing the memory hierarchy with the neural network ran on the architecture. The hard-wired kernel size allows efficient data reuse.
- *CUTIE* [22] is an accelerator for ternary neural networks. This is a massively parallel architecture, hard-coding all the network parameters into the hardware design. Furthermore, it exploits sparsity opportunities from ternary networks that are not present in binary networks.
- *Knag et al.* produced a binary neural network accelerator in 10-nm FinFet technology [16]. The design focuses on utilizing the compute near memory paradigm, minimizing the cost of data movement by interleaving memory and computational elements.
- *BrainTTA* is a flexible, fully programmable solution based on a Transport-Triggered Architecture. The architecture has support for mixed-precision and focuses, next to the energy efficiency objective also on flexibility, trying to minimize the concessions made while still pursuing energy efficiency.

A summary of these architectures is given in Table 1, and the strengths and weaknesses of the architectures are discussed in Sect. 4.

3.2.1 XNOR Neural Engine (XNE)

XNOR Neural Engine [6] is a binary accelerator exploiting the arithmetic simplifications introduced by binarizing the weights and activations (see Fig. 3). Conti et

Table 1 Comparison of performance, efficiency, and flexibility of the architectures discussed

	ChewBaccaNN [1]	CUTIE [22]	XNE [6]	10nm FinFet [16]	BrainTTA
Implementation characteristics					
Tech node [nm]	22	22	22	10	28
Supply voltage [V]	0.4	0.65	0.6 / 0.4	0.39	0.9
Inference precision[a]	b	b[b], t	b	b	b, t, i8
Memory technology	SCM	SRAM / SCM	SRAM / SCM	SRAM	SRAM
Key Performance Indicators					
Peak throughput [GOPS]	240	16,000	67 / 5	3400	880
Energy/op [fJ] binary	4.48/15.38[c]	–	115 / 21.6	1.62	101
Energy/op [fJ] ternary	–	2.19 / 1.70	–	–	188
Energy/op [fJ] 8-bit	–	–	–	–	1105
Core area [mm^2]	0.7	7.5	2.32	0.39	3.6
Area efficiency [$GOPS/mm^2$]	343	2133	28.88	8717	244.44
Memory capacity [kB]	153	1190 / 281	520 / 16	161	1024
Flexibility					
Full utilization for					
C multiple of	16	128	128	1024	32/16/4[d]
M multiple of	Any	128	128	128	32
R is	7	3	Any	2	Any
S is	7	3	Any	2	Any
Partial result support (for scheduling freedom)	Yes	No[e]	No	No	Yes
Residual layer support	Yes	No	No	No	Yes
Programmability	None	None	None	None	C-language

[a] b = `binary`, t = `ternary`, i8 = `integer8`
[b] Only estimates were provided, under the assumption that all ternary specific hardware is removed
[c] For 7×7 and 3×3 convolution, respectively
[d] For `binary`, `ternary` and `integer8`, respectively
[e] Partial result support is not needed since the output pixel computation is fully unrolled in hardware

al. present an SoC consisting of an accelerator core (XNE) inside a microcontroller unit (MCU) and peripheries. The accelerator can independently run simple network configurations but requires the programmable MCU to execute more complex

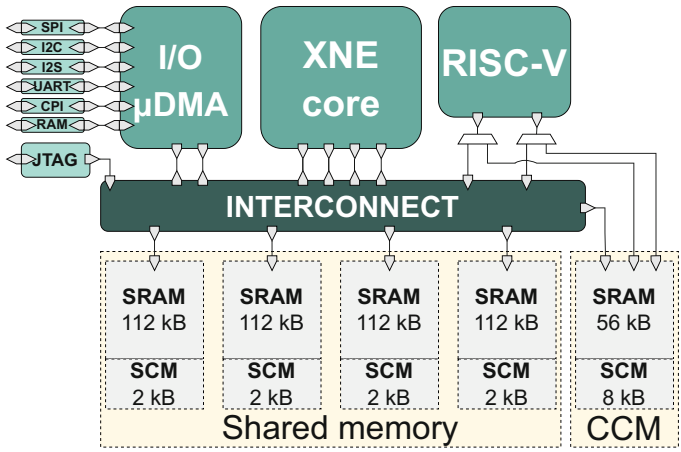

Fig. 6 Top-level view of the SoC with XNE inside the MCU. The memory is a hybrid of latch-based SCM and SRAM

layers. The MCU is programmed using some assembly dialect. The full system is shown in Fig. 6. It consists of:

- *XNE core*, where the `binary MAC` operations are performed; this core consists of a *streamer*, to stream feature maps and weights in and out of the architecture, a *controller* consisting of a finite-state machine, the programmable microcode processor, and a latch-based register file.
- *RISC-V host processor*, used to realize more complex layer behaviors than supported with the XNE core alone.
- *Shared Memory*, shared between the μDMA, RISC-V core, and XNE core. This memory is a hybrid of SRAM and SCM, allowing aggressive voltage scaling when the SRAMs are turned off.
- *Core-Coupled Memory (CCM)*, primarily for the RISC-V core, again composed of both SRAM and SCM.
- *μDMA*, which is an autonomous unit able to send and receive data via several communication protocols from and to the shared memory.

The accelerator core, XNE, is shown in Fig. 7. The throughput of the design can be chosen at design time by means of a throughput parameter `TP`. This throughput parameter can be described as follows: it takes the accelerator `TP` cycles to calculate `TP` output pixels. While doing this, the accelerator keeps the same input activations for `TP` cycles while loading `TP` weights each cycle (for a total of `TP` sets of `TP` weights). Therefore, this `TP` parameter essentially hard-wires the C and M dimension of the convolution dimensions shown in Fig. 2 into the design.

For instance, each accumulator in Fig. 7 contains the partial result of one output pixel (i.e., the number of accumulators is equal to the output feature map channel vectorization v_M). Therefore, all the inputs that are processed while a single accumulator is selected via the mux should contribute to the same output pixel.

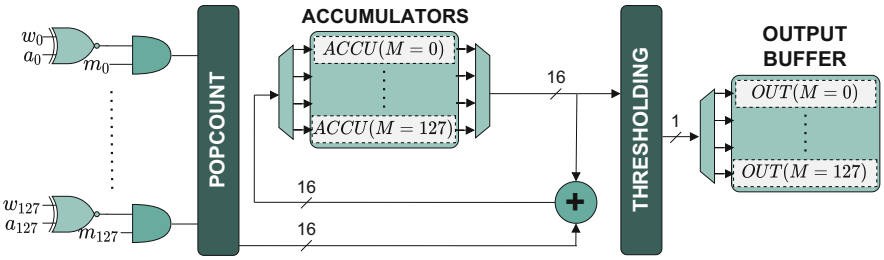

Fig. 7 Accelerator core of XNOR Neural Engine with `TP` = 128. The XNOR operation is performed on the activations a and weights w. Whenever the number of input operands is not a multiple of `TP`, the outputs can be masked by masking bits m to make sure that they do not contribute to the popcount output

In this case, the different pixels concurrently offered to the compute core belong to different input channels. Therefore, the choice of `TP` directly imposes a constraint on the C and M loops in order to run at full efficiency. Furthermore, the output of the `popcount` operation is directly fed through the binarization function; this means that partial (higher bit-width) results cannot be extracted, prohibiting their use for residual layers. For benchmarking the platform, a `TP` factor of 128 was chosen, which means that $v_C = 128$ and $v_M = 128$ for this design point.

3.2.2 ChewBaccaNN

ChewBaccaNN [1] is like XNE, an accelerator utilizing binary weights and binary activations. Contrary to XNE, this architecture does not implement a full SoC and is therefore purely based on the accelerator core. ChewBaccaNN is designed using GF22 technology and uses SCM to enable aggressive voltage scaling. A top-level view of the architecture is shown in Fig. 8. The components in this architecture are:

- *BPU Array* consists of seven Basic Processing Units (BPUs) and forms the computational heart of the accelerator; the BPU is detailed in Fig. 9 and is discussed in the next paragraph.
- *Feature Map Memory (FMM)* holds the input and output feature maps and also has the ability to store partial results (e.g., for residual layers). The FMM is implemented using SCM only. This enables aggressive voltage scaling for the whole chip at the expense of sacrificing memory capacity.
- *Row Banks* buffer the input feature map rows and kernel rows. The *crossbar* (x-bar) is utilized when the convolutional window slides down. Since each BPU processes one kernel row, the kernel weights can stay inside the BPUs, while the input feature map needs to move one row down. This is done by loading one new row and shifting the other rows by one BPU (using the crossbar).

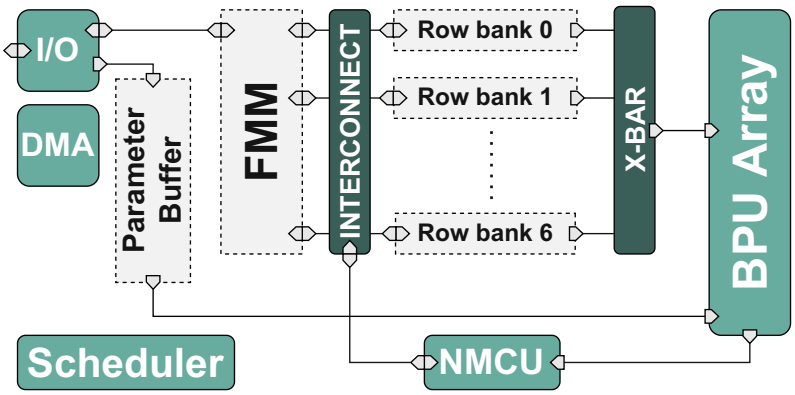

Fig. 8 The top-level architectural overview of the ChewBaccaNN accelerator. All the memories are implemented using latch-based standard-cell memory. The control signals are not shown in this overview

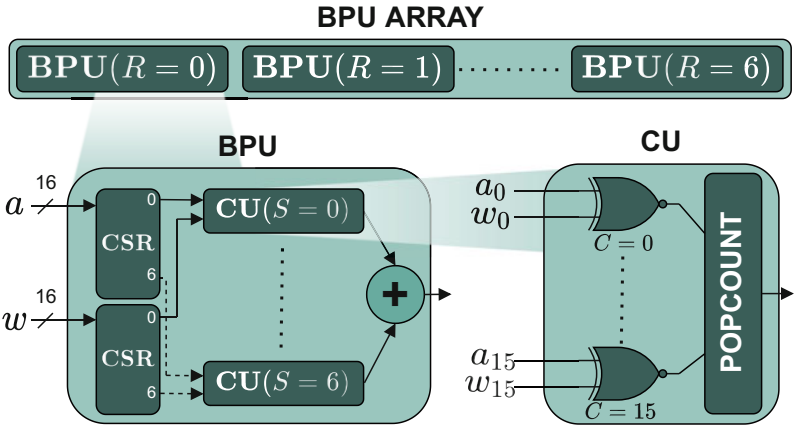

Fig. 9 ChewBaccaNN compute core. Hardware parallelization is performed over the kernel height (R) in the Basic Processing Unit (BPU) array, over the kernel width (S) inside a single BPU and over the input channel dimension (C) inside the compute unit (CU). The Controlled Shift Register (CSR) enables data reuse in a sliding window fashion. The architecture contains a total of $16 \times 7 \times 7 = 784$ ($v_C \times v_R \times v_S$) binary multipliers

- *Scheduler*, used to control the crossbar behavior and make sure that the row banks are timely rotated to the next BPU and the correct weights and IFM pixels are loaded.
- *Near Memory Compute Unit (NMCU)*, which writes output data from the BPU array to the correct location in the FMM, accumulates residual paths, rebinarizes results, and is used for bit-packing (rebinarized) outputs into 16-bit packets.

In Fig. 9, the compute core of ChewBaccaNN is depicted. It can be seen that several of the parameters listed in Fig. 2 are hard-wired into the design. The kernel height

(R) and width (S) are completely unrolled (in this case with a factor of 7), while the channel dimension (C) should be a multiple of 16 (the number of `XNOR` gates) to achieve full utilization; in other words, the vectorization factors are $v_C = 16$, $v_R = 7$, and $v_S = 7$.

The Controlled Shift Register (CSR) allows using the sliding window principle to get data reuse; for each IFM image row, initially, the full kernel width (in this case 7) is transferred, while the iterations thereafter only need one new column ($v_R \times 1 \times v_C$) of activations.

3.2.3 Completely Unrolled Ternary Inference Engine (CUTIE)

Completely Unrolled Ternary Inference Engine (CUTIE) [22] is, as the name suggests, an inference accelerator using `Ternary` operands. The main design philosophy behind CUTIE is to avoid iteration by spatially unrolling most of the convolutional loops found in Listing 1, namely the loops over the R, S, C, and M dimensions. Furthermore, `ternary` operands allow the representation of zero, therefore making the accelerator capable of exploiting neural network sparsity by silencing compute units. The top-level design of CUTIE is depicted in Fig. 10. The main components within the CUTIE architecture are:

- *Output Channel Compute Unit (OCU)*, the basic compute building block of this architecture, computing the output pixels belonging to one single output channel. A detailed view of the OCU is given in Fig. 11.
- *Feature Map Memory (FMM)*, used to store the inputs coming either from previous computations (OCUs) or from an external interface. The FMM is double-buffered such that the latency for loading new input feature maps can be hidden.

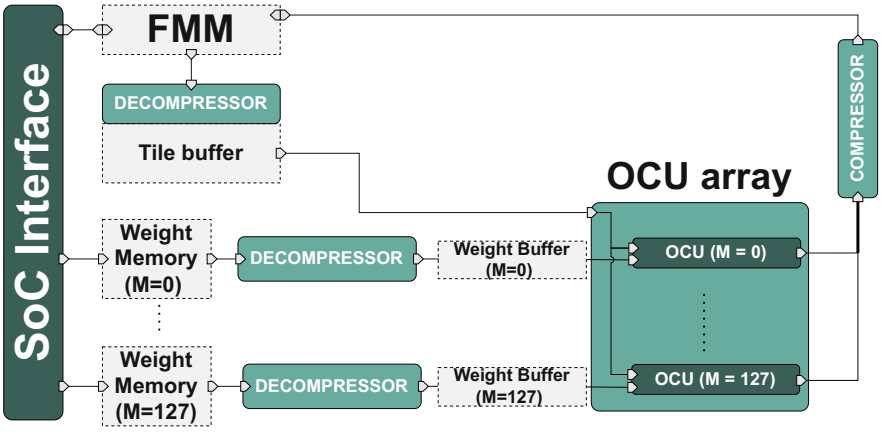

Fig. 10 CUTIE top-level architecture. The OCU array contains one output channel compute unit for each output channel in the neural network design

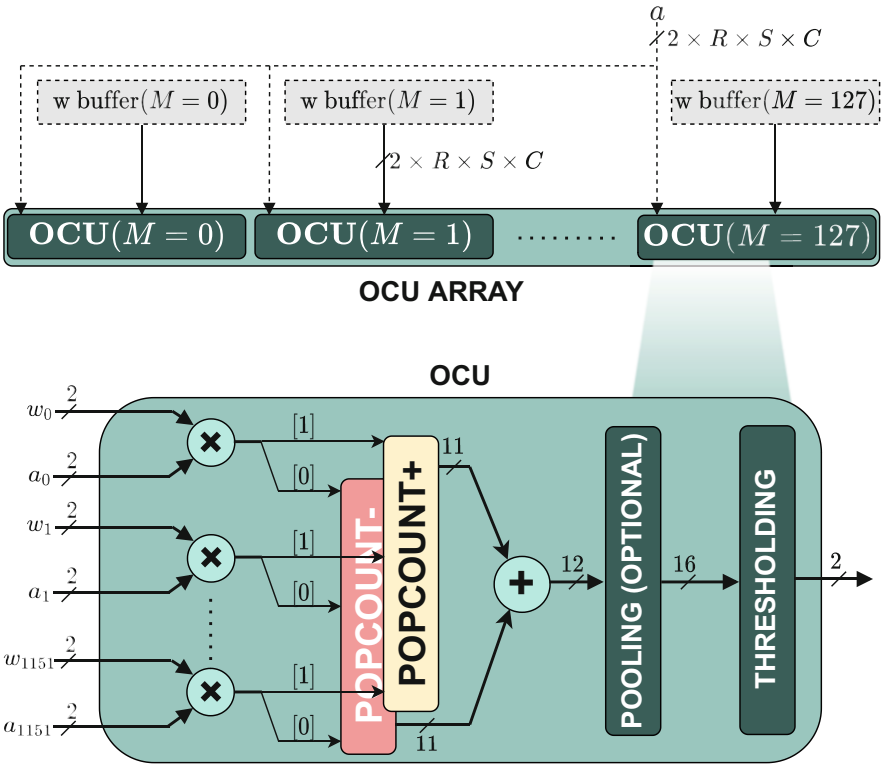

Fig. 11 CUTIE compute core, consisting of several Output Channel Compute Units (OCUs) and one weight buffer per OCU. For brevity, decompression and pipelining are omitted in this figure. The ternary multipliers are unrolled over the R, S, and C dimensions, which in this case gives $3 \times 3 \times 128 = 1152$ ternary multipliers. In total, the architecture can process $3 \times 3 \times 128 \times 128 = 147{,}456$ ($v_R \times v_S \times v_M \times v_C$) inputs each compute cycle

- *Tile buffer*, used to buffer IFM pixels in a sliding window fashion.
- *Weight buffer*, one is attached to each OCU: it is designed with enough capacity to contain the full kernel for a single output channel ($R \times S \times C$), which enables great weight reuse. The weight buffer is also double-buffered to hide latency.
- *Compression/decompression units* are used to shift between the computational form of the trits, i.e., 2 bits, and the compressed form of the trits which is 1.6 bits wide.

The compute core of CUTIE is depicted in Fig. 11. Its main workhorse is the Output Channel Compute Unit (OCU), which is a unit that calculates pixels *exclusive to a single output channel*. Having a separate compute unit for each output channel brings the advantage that the weight kernel can stay inside the weight buffer (w buffer) while moving the convolutional window over the IFM giving maximum weight data reuse. Alongside the weight reuse, there is also IFM reuse being utilized

in two different ways: (1) the IFM is broadcasted to each of the OCUs and (2) just like ChewBaccaNN, when sliding the convolutional window over the IFM image, only R new IFM pixels are needed (i.e., only one new column of the IFM needs to be loaded, assuming a stride of 1).

Each Output Channel Compute Unit (OCU) processes $128 \times 3 \times 3$ ($v_C \times v_R \times v_S$) input pixels each cycle. By hard-wiring many of the convolutional layer parameters, CUTIE sacrifices area in favor of avoiding temporal iteration. This also means that this architecture sacrifices most flexibility by constraining C, M, R, and S. Therefore, the only dimensions that are freely schedulable are W and H. By constraining many of the dimensions into the hardware, flexibility crumbles, but the temporal mapping is greatly simplified. The fully spatially unrolled structure also minimizes the movement of (large) partial sums. Since each OCU directly computes an output pixel, there is no need, in contrast to the other architectures, to move around partial results. This is beneficial since the partial results have a higher bit-width than the final (requantized) results.

3.2.4 Binary Neural Network Accelerator in 10-nm FinFet

In [16], Knag et al. show a fully digital accelerator with `binary` operands which is implemented using 10nm FinFet technology. The SoC designed intersperses arithmetic with memory according to the Compute Near Memory (CNM) paradigm. Contrary to the other architectures discussed, this work focuses more on the physical implementation and circuit-level design choices rather than the architectural design aspects. The design of this accelerator is shown in Fig. 12. The main components of this accelerator are:

- *Control Unit*, which consists of four 256-bit wide SRAM memory banks used as main storage and a Finite-State Machine (FSM) that controls the flow of data between memory banks and the MEUs.
- *Memory Execution Unit (MEU)*. Each MEU can compute two output pixels in a time-interleaved manner (see Fig. 12, each MEU contains two output registers). The MEUs are interleaved with latch-based memories to utilize the compute near memory advantages. In total, there is an array of 16×8 MEUs. Having 8 weight SCM banks was found to be the right trade-off between energy consumed by the computational elements and energy consumed by the transportation of data to the compute units. The SRAM memory banks are connected to the MEUs by means of a crossbar network. Since the input feature map pixels are stored in an interleaved manner, the crossbar network allows any (2×2) combination of the input feature map to be read. The weights are also loaded from this memory.

The authors of the work do not discuss the external interfacing required on this chip.

Binary arithmetic is relatively cheap, compared to the cost of accessing memory (e.g., for loading weights). To amortize the costs of memory reads and data movement, the computational intensity should be sufficient to balance the energy

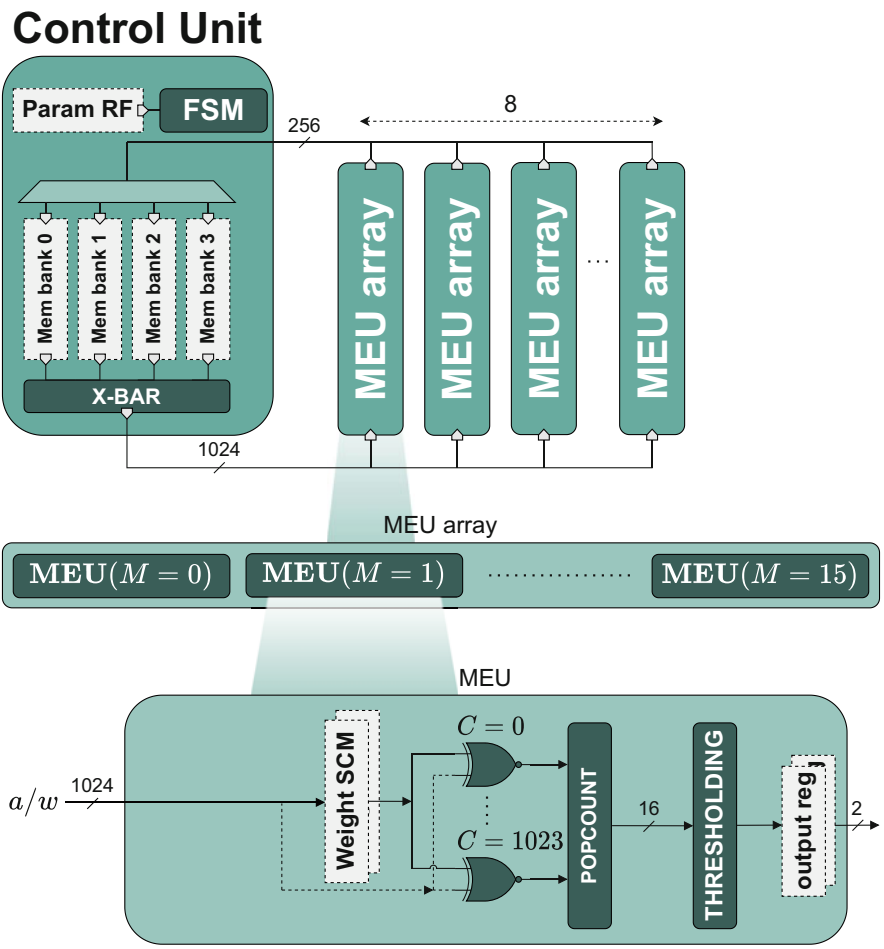

Fig. 12 Top-level view of the 10nm FinFet BNN accelerator. The central memory inside the control unit consists of 4× 256-bit wide SRAM banks to enable 2 × 2 convolutional window access in a single cycle and a finite-state machine (FSM). The MEUs are placed in an 8 × 16 array to exploit the compute near memory principle. In total, 1024 × 16 × 8 = 131,072 ($v_C \times v_M \times 8$) binary operations can be performed each cycle

consumption. Parallelism of the `MAC` unit (as shown in Fig. 12) is used to balance the power mismatch of the (expensive, high bit-width) accumulator, present in the `popcount` module, and the (cheap) `XNOR` gates. By enlarging the number of inputs of the `popcount` module, the fixed accumulator cost is amortized by many `XNOR` gates. Like the other architectures, this accelerator parallelizes the `MAC` operation over the input channel (C) dimension. The parallelization should be high enough to offset the accumulator cost while being low enough to not impose unreasonable constraints on the number of input channels (C) required for

full utilization. Therefore, a trade-off study was performed to see which level of parallelism was needed to offset the accumulator cost. A design with an input feature map parallelization factor of 1024 ($v_C = 1024$) was chosen as the sweet-spot. Negligible energy improvements were shown when going for more parallelism.

Furthermore, the idea of pipelining the `popcount-adder` tree was explored. When pipelining the design, the voltage can be lowered at iso-performance (i.e., iso-frequency). However, due to the sequential logic and clock-power dissipated while adding more pipeline stages, the final design choice was to not pipeline the `popcount-adder` tree.

3.2.5 BrainTTA

BrainTTA is a fully compiler-programmable mixed-precision flexible-datapath architecture. Contrary to the fixed-path accelerators, BrainTTA is based on the Transport-Triggered Architecture (TTA) [7] that provides a fully programmable datapath (via a compiler) directly to the user. Before diving into the BrainTTA architecture, a proper introduction to the Transport-Triggered Architecture is given.

Transport-Triggered Architectures are programmed by data movements instead of arithmetic operations typically found in Very Long Instruction Word (VLIW) architectures. This means that the movement of data between function units (FUs) and register files (RFs) is exposed to the programmer; the TTA is an explicit datapath architecture. This is in stark contrast to VLIW architectures, where the data movement is implicit and performed in hardware (i.e., not exposed to the programmer). With the control of the datapath given to the compiler, several optimizations can be performed like operand sharing and register file bypass.

An example instance of a TTA is displayed in Fig. 13. The TTA consists of a Control Unit (CU) used for instruction fetching and decoding, Register Files (RFs) for temporary storage, and Load-Store Units (LSUs) to access the memories. The gray circles inside the busses denote that this bus is connected to the corresponding input- or output-port of some function unit. This connectivity is design time configurable, visible to the compiler, and can be made as generic or specific for certain applications as desired; more connectivity is at the expense of larger instruction size and more switching activity in the interconnect. In [19], Multanen presented several ways to alleviate this effect by applying techniques that reduce the instruction overhead such as instruction compression. An example instruction is shown in Fig. 13, which shows that the instruction can be broken down into *move operations* for each bus.

BrainTTA is based on the TTA, built specifically for inference with precisions `integer8`, `binary`, and `ternary`. A top-level view of the BrainTTA SoC is shown in Fig. 14. BrainTTA is designed using the open-source toolchain TTA-based Co-design Environment (TCE) [9, 13]. The SoC consists of:

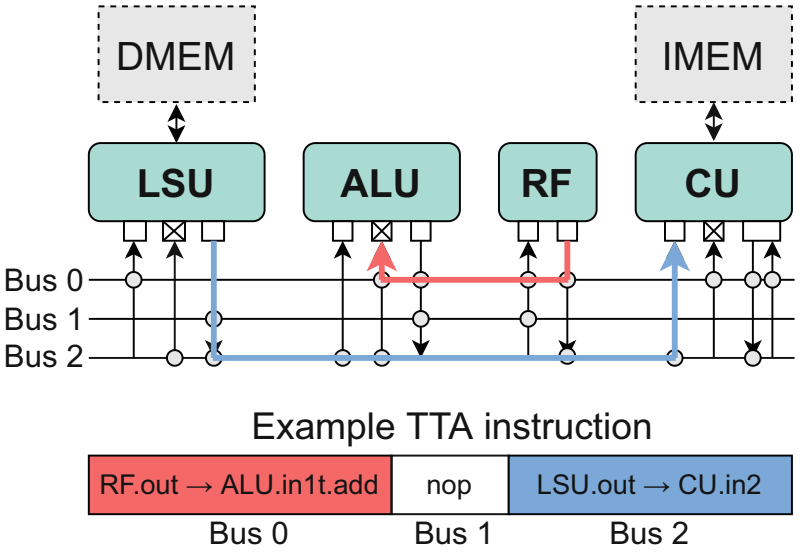

Fig. 13 An example TTA instance and instruction, the square blocks denote *input-* and *output-ports*. A cross denotes a *trigger-port*. The colored arrows drawn on the architecture illustrate the *move operations* inside the example instruction

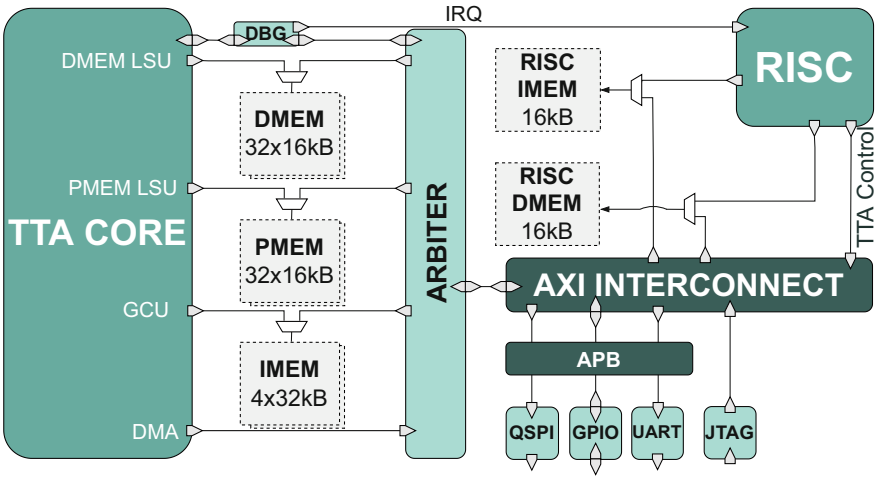

Fig. 14 Top-level view of the BrainTTA SoC, the arbiter forms the border between the RISC and TTA part of the SoC

- *RISC-V host processor*, which is taken from an open-source repository [24], the host processor starts and halts execution of the TTA core and takes care of the external communication (e.g., loading the on-chip memories).
- *TTA core*, the workhorse of the architecture, supports mixed-precision inference.

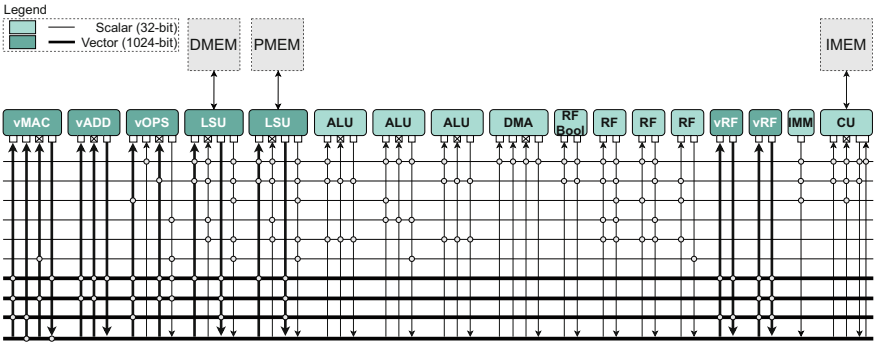

Fig. 15 BrainTTA core instance, thicker lines denote vector busses, thinner lines scalar busses

- *SRAM Memories*, separate memories for the RISC and TTA core, the TTA core memories are highly banked to allow efficient access of smaller bit-widths, while also supporting wide vector accesses. The TTA core is connected to three memories, the DMEM, used for storing input and output feature maps, the PMEM, used to store the weights, and the IMEM used for instructions to program the behavior of the TTA core.
- *Debugger (DBG)*, used to control the execution of the TTA core, can signal task completion to the RISC-V.
- *AXI interconnect*, used for on- and off-chip communication between the RISC, TTA core, and peripherals.

The workhorse of this architecture is the TTA core, where the actual inference happens. The details of the TTA core instantiation used in BrainTTA can be found in Fig. 15. The core contains different Function Units (FUs), divided into scalar and vector FUs. The FUs are interconnected via the busses, with 32-bit scalar busses (bus 0–5) and 1024-bit vector busses (bus 6–9). The core consists of the following units:

Control Unit (CU) it contains the logic to fetch and decode instructions and steers the other units to execute the correct operations. Furthermore, the CU contains a hardware loop buffer to save energy on the instruction memory accesses. This can be very beneficial since all network layers are essentially described by multiple nested loops (see Listing 1).

Vector Multiply–Accumulate (vMAC), the actual number cruncher. This unit supports the following operations: `integer8 MAC` (scalar–vector product and vector–vector product), `binary MAC`, and `ternary MAC`. Its vector size is 1024-bit, with 32 entries of 32-bits each. The scalar–vector `MAC` multiplies a scalar by a vector by broadcasting the (32-bit) scalar value to all vector entries. This is beneficial when multiple inputs share the same weights (as in convolution).

For each precision `MAC` operation, the vectorization factor is different. All arithmetic circuitry contains 32 accumulators for the (intermediate) output channel

result, i.e., $v_M = 32$. The number of concurrent input channels is the vector size (1024) divided by 32 (the number of output channels) divided by the operand size (i.e., 1, 2, or 8 bits). Therefore, the input channel vectorization is $v_C = 32$, $v_C = 16$, and $v_C = 4$ for `binary`, `ternary`, and `integer8`, respectively.

Vector Add (vADD) is used to add two (either 512- or 1024-bit) vectors. This can for example be used to support residual layers.

Vector Operations (vOPS), auxiliary (vector) operations that are required in the network, alongside to the computations. This FU can perform `requantization`, `binarization`, `ternarization`, as well as activation functions, e.g., `ReLU` and pooling functions such as `MaxPool`. Furthermore, various other operations to extract and insert scalar elements into a vector are also supported by this unit.

Register Files (RFs) come in different bit-widths, namely binary, 32-bit scalar, and 1024-bit vector. These registers can be used to facilitate data reuse and store intermediate results without performing (more costly) access to the SRAM.

Load-Store Units (LSUs) form the interface between the TTA core and the SRAM memory. For each memory, there is a separate LSU to facilitate concurrent weight and input loading. The units support loads and stores for different bit-widths ranging from 8 bits all the way up to 1024 bits. Since the memory is banked, a strobe signal can be used to selectively turn on banks when data with smaller bit-widths are loaded/stored, in order to save energy.

Scalar ALUs are mostly used for address calculations needed as inputs to the LSUs. These units support basic arithmetic on values up to 32 bit.

4 Comparison and Discussion

All architectures discussed in Sect. 3 are evaluated on flexibility and energy efficiency. These results are given in Table 1. This table is split into three sections: the implementation characteristics, performance characteristics as discussed in Sect. 3.1.2, and the flexibility aspects as discussed in Sect. 3.1.1.

The energy efficiency of the accelerators ranges from 1.6 to 115 fJ per operation for binary precision, a large range. It should be noted, however, that the two architectures that have the highest energy usage (XNE and BrainTTA) are the only architectures that show a full autonomous SoC including peripherals. Furthermore, all architectures except BrainTTA utilize voltage–frequency scaling to run the accelerator at lower than nominal supply voltage, trading off throughput for better energy efficiency.

Next to the energy efficiency, the table also lists the neural network layer requirements that these architectures impose in order to fully utilize the arithmetic hardware. It is seen that the most energy-efficient architectures, CUTIE [22] and the BNN accelerator in 10-nm FinFet from Knag et al. [16], are also the most constrained architectures, in terms of neural network layer requirements. Therefore,

the question arises, does hard-wiring the neural network layer parameters directly improve the energy efficiency of an architecture, for different models, also when layer variety is high?

Interestingly, the XNE and BrainTTA share very similar layer constraints. Both are only constrained in the input channel (C) and output channel (M) dimensions. The energy consumption of BrainTTA is somewhat lower at an older technology node while using a higher supply voltage. The reason for this is that BrainTTA better exploits data reuse. The execution schedule for BrainTTA was tuned to maximize data reuse, while XNOR neural engine only reuses a set of input feature maps for `TP` (in this case 128) cycles while reloading the weights for each `MAC` operation.

The inefficient schedule of XNE is confirmed by the energy numbers of the implementation that only uses SCM. XNE was benchmarked using SCM only, severely cutting the very high energy cost associated with these redundant memory fetches, at the cost of losing memory capacity. Some architectures report energy numbers for an SCM as well as an SRAM implementation. The memory capacity of the SCM versions is very low compared to the SRAM versions, hindering the ability to run full-size networks on it without adding expensive off-chip memory accesses. For the sake of comparison, for all the architectures with an SRAM version available, the SRAM version is chosen for further analysis.

Support for residual layers can only be found in ChewBaccaNN and BrainTTA. Other architectures are not able to support this due to their fixed datapath. The dataflow through these accelerators is very static, and the accumulated value will directly be `binarized` or `ternarized` after all inputs are accumulated. This prohibits the use of residual layers since residual layers need the intermediate (larger bit-width) results that were obtained before requantization.

It is clear that parallelism and data reuse (either in the form of locally buffering or by broadcasting) are the keys to amortizing the memory access cost, which is so much larger than the low-precision arithmetic cost. Techniques to mitigate these costs are to replace SRAM with low-voltage SCM, hard-wire network parameters to enable broadcasting, and use the sliding window principle (like the FMM banks in combination with the crossbar in ChewBaccaNN [1]). In essence, all these solutions boil down to designing the architecture around the data movements in a less-flexible manner. These architectures solve the mapping problem by fixing most parameters using **spatial mapping**, greatly simplifying the task of **temporal mapping** at the cost of losing flexibility. XNE and BrainTTA fix the least number of parameters using **spatial mapping**, therefore leaving a larger **temporal mapping** space to be explored.

5 Summary and Conclusions

Neural networks are all around and are making an advance into the embedded domain. With the increasing popularity of edge computing, new methods are needed to port the typically power- and memory-hungry neural networks to devices that

have limited storage and are subject to severe energy constraints. Quantization is a fundamental ingredient in overcoming these challenges. Very low precisions, down to 1 bit, have shown to achieve great energy efficiency while drastically reducing the model size and computational cost involved in neural network inference. To fully exploit the reduced computational complexity and memory requirements of these networks, neural network accelerators aimed specifically at these heavily quantized networks have been developed.

In this chapter, state-of-the-art low- and mixed-precision architectures are reviewed. Taking into account the variety present in network layers of CNNs, the architectures are compared against each other in terms of flexibility and energy efficiency. It was found that spatially mapping more dimensions of the neural network layer increases the energy efficiency as it allows minimization of data movement by tailoring the memory hierarchy design, which is a big contributor to energy cost in inference accelerators. Contrary to the group of accelerators that maps most layer dimensions spatially, there is a group of accelerators that minimizes the layer dimension requirements by less heavily relying on spatial mapping, retaining more freedom in the temporal mapping domain. They are more flexible and can handle a larger part of the neural architecture design space. In addition, they may have support for multiple bit precisions.

With new attempts to streamline the process of finding the best combination of temporal and spatial mappings [17], while co-designing the memory hierarchy, the question arises if an optimized temporal mapping in combination with memory hierarchy co-design can close the energy efficiency gap with the more constrained, heavily spatially mapped accelerators, giving better energy efficiency at a wider range of neural network layers.

References

1. Andri, R., Karunaratne, G., Cavigelli, L., Benini, L.: ChewBaccaNN: A flexible 223 TOPS/W BNN accelerator. arXiv (May), 23–26 (2020)
2. Bankman, D., Yang, L., Moons, B., Verhelst, M., Murmann, B.: An always-on 3.8 μ J/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS. IEEE J. Solid-State Circuits **54**(1), 158–172 (2019). https://doi.org/10.1109/JSSC.2018.2869150. https://ieeexplore.ieee.org/document/8480105/
3. Bengio, Y., Léonard, N., Courville, A.: Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation pp. 1–12 (2013). http://arxiv.org/abs/1308.3432
4. Blalock, D., Ortiz, J.J.G., Frankle, J., Guttag, J.: What is the State of Neural Network Pruning? (2020). http://arxiv.org/abs/2003.03033
5. Bulat, A., Tzimiropoulos, G.: XNOR-Net++: Improved binary neural networks. In: 30th British Machine Vision Conference 2019, BMVC 2019 pp. 1–12 (2020)
6. Conti, F., Schiavone, P.D., Benini, L.: XNOR neural engine: a hardware accelerator IP for 21.6-fJ/op binary neural network inference. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **37**(11), 2940–2951 (2018). https://doi.org/10.1109/TCAD.2018.2857019
7. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. Wiley, Hoboken (1997)

8. Deng, L., Jiao, P., Pei, J., Wu, Z., Li, G.: GXNOR-Net: training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. Neural Netw. **100**, 49–58 (2018). https://doi.org/10.1016/j.neunet.2018.01.010
9. Esko, O., Jääskeläinen, P., Huerta, P., De La Lama, C.S., Takala, J., Martinez, J.I.: Customized exposed datapath soft-core design flow with compiler support. In: Proceedings - 2010 International Conference on Field Programmable Logic and Applications, FPL 2010, pp. 217–222 (2010). https://doi.org/10.1109/FPL.2010.51
10. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A Survey of Quantization Methods for Efficient Neural Network Inference (2021). http://arxiv.org/abs/2103.13630
11. Gluska, S., Grobman, M.: Exploring Neural Networks Quantization via Layer-Wise Quantization Analysis (2020). http://arxiv.org/abs/2012.08420
12. Huang, S., Waeijen, L., Corporaal, H.: How flexible is your computing system? ACM Trans. Embedd. Comput. Syst. (2022). https://doi.org/10.1145/3524861. https://dl.acm.org/doi/10.1145/3524861
13. Jääskeläinen, P., Viitanen, T., Takala, J., Berg, H.: HW/SW co-design toolset for customization of exposed datapath processors. In: Computing Platforms for Software-Defined Radio, pp. 147–164. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-49679-5_8
14. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 2704–2713 (2018). https://doi.org/10.1109/CVPR.2018.00286
15. Kharya, P.: TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x (2020). https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/
16. Knag, P.C., Chen, G.K., Sumbul, H.E., Kumar, R., Hsu, S.K., Agarwal, A., Kar, M., Kim, S., Anders, M.A., Kaul, H., Krishnamurthy, R.K.: A 617-TOPS/W all-digital binary neural network accelerator in 10-nm FinFET CMOS. IEEE J. Solid-State Circuits **56**(4), 1082–1092 (2021). https://doi.org/10.1109/JSSC.2020.3038616
17. Mei, L., Houshmand, P., Jain, V., Giraldo, S., Verhelst, M.: ZigZag: enlarging joint architecture-mapping design space exploration for DNN accelerators. IEEE Trans. Comput. **70**(8), 1160–1174 (2021). https://doi.org/10.1109/TC.2021.3059962
18. Muller, O., Prost-Boucle, A., Bourge, A., Petrot, F.: Efficient decompression of binary encoded balanced ternary sequences. IEEE Trans. Very Large Scale Integr. Syst. **27**(8), 1962–1966 (2019). https://doi.org/10.1109/TVLSI.2019.2906678
19. Multanen, J.: Energy-Efficient Instruction Streams for Embedded Processors. Ph.D. Thesis, Tampere University (2021)
20. Parashar, A., Raina, P., Shao, Y.S., Chen, Y.H., Ying, V.A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S.W., Emer, J.: Timeloop: A systematic approach to DNN accelerator evaluation. In: Proceedings - 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, pp. 304–315 (2019). https://doi.org/10.1109/ISPASS.2019.00042
21. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: XNOR-net: ImageNet classification using binary convolutional neural networks. In: Computer Vision—ECCV 2016. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), LNCS, vol. 9908, pp. 525–542 (2016). https://doi.org/10.1007/978-3-319-46493-0_32
22. Scherer, M., Rutishauser, G., Cavigelli, L., Benini, L.: CUTIE: Beyond PetaOp/s/W Ternary DNN Inference Acceleration with Better-than-Binary Energy Efficiency pp. 1–14 (2020). http://arxiv.org/abs/2011.01713
23. Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., Le, Q.V.: MnasNet: Platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2019-June, pp. 2815–2823 (2019). https://doi.org/10.1109/CVPR.2019.00293

24. Traber, A., Gautschi, M.: PULPino: Datasheet. ETH Zurich, University of Bologna (2017)
25. Ueyoshi, K., Papistas, I.A., Houshmand, P., Sarda, G.M., Jain, V., Shi, M., Zheng, Q., Giraldo, S., Vrancx, P., Doevenspeck, J., Bhattacharjee, D., Cosemans, S., Mallik, A., Debacker, P., Verkest, D., Verhelst, M.: DIANA: An end-to-end energy-efficient digital and ANAlog hybrid neural network SoC. In: 2022 IEEE International Solid- State Circuits Conference (ISSCC), pp. 1–3. IEEE (2022). https://doi.org/10.1109/ISSCC42614.2022.9731716. https://ieeexplore.ieee.org/document/9731716/
26. Valavi, H., Ramadge, P.J., Nestler, E., Verma, N.: A 64-Tile 2.4-Mb in-memory-computing CNN accelerator employing charge-domain compute. IEEE J. Solid-State Circuits **54**(6), 1789–1799 (2019). https://doi.org/10.1109/JSSC.2019.2899730. https://ieeexplore.ieee.org/document/8660469/
27. Wang, S., Kanwar, P.: BFloat16: The secret to high performance on Cloud TPUs (2019). https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus
28. Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., Keutzer, K.: FBNET: Hardware-aware efficient convnet design via differentiable neural architecture search. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2019-June, pp. 10726–10734 (2019). https://doi.org/10.1109/CVPR.2019.01099
29. Wu, Y.N., Emer, J.S., Sze, V.: Accelergy: An architecture-level energy estimation methodology for accelerator designs. In: IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD 2019-Nov (2019). https://doi.org/10.1109/ICCAD45719.2019.8942149