IEEE Open Journal of
Circuits and Systems

Received 18 May 2020; revised 8 August 2020 and 14 October 2020; accepted 4 November 2020. Date of publication 16 November 2020; date of current version 8 January 2021.

Digital Object Identifier 10.1109/OJCAS.2020.3037758

ConvAix: An Application-Specific Instruction-Set Processor for the Efficient Acceleration of CNNs

ANDREAS BYTYN, RAINER LEUPERS, AND GERD ASCHEID (Senior Member, IEEE)

Institute of Communication Technologies and Embedded Systems, RWTH Aachen University, 52074 Aachen, Germany
This article was recommended by Associate Editor J. Park.

CORRESPONDING AUTHOR: A. BYTYN (e-mail: bytyn@ice.rwth-aachen.de)

This work was supported by the German Federal Ministry of Education and Research (BMBF) via the PARIS Project under Grant 16ES0602.

ABSTRACT ConvAix is an application-specific instruction-set processor (ASIP) that enables the energy-efficient processing of convolutional neural networks (CNNs) while retaining substantial flexibility through its instruction-set architecture (ISA) based design. By utilizing a combination of data-level parallelism (DLP), instruction-level parallelism (ILP), and subword parallelism, the proposed design offers sufficient processing power for the execution of state-of-the-art CNNs in real-time. ConvAix's arithmetic logic units (ALUs) are C-programmable, thereby offering the degree of flexibility required to implement many different convolution layer types, e.g., depthwise-separable convolutions and residual blocks, as well as fully-connected and pooling layers. It comprises a total of 256 ALUs and leverages low-precision computations down to 4 bits. Furthermore, it exploits sparsity in feature maps and weights via zero-guarding of redundant computations to maximize its energy efficiency. The processor was implemented in a modern 28 nm CMOS technology operating at 1 V supply voltage with a resulting clock frequency of 513 MHz. The final design offers a precision-dependent peak throughput between 263 GOP/s (int16) and 1.1 TOP/s (int4), while consuming between 972 mW and 340 mW of power, resulting in effective energy-efficiencies ranging from 176 GOP/s/W to 2 TOP/s/W. Well-known CNNs, such as AlexNet, MobileNet, and ResNet-18, are simulated based on the placed and routed netlist, achieving between 233 (AlexNet) and 69 (ResNet-18) frames-per-second for a batch-size of 1, including times for off-chip transfers.

INDEX TERMS Application-specific instruction-set processor (ASIP), convolutional neural network (CNN), very large instruction word (VLIW), quantization, low-precision computing, instruction-set architecture (ISA), deep learning, machine learning, processor architecture, subword parallel.

I. INTRODUCTION

CONVOLUTIONAL neural networks (CNNs) are nowadays widely used for tasks such as image classification [1]–[4], object detection [5]–[7], semantic segmentation [8]–[10], face detection [11], and many more. This is primarily due to their excellent algorithmic performance paired with the ability to train them without the need for a human expert who handcrafts features, as previously done in classical machine learning [12]. While training a CNN is usually done offline, e.g., in the cloud, the execution (also referred to as inference) often happens on the mobile device itself, either due to latency constraints or because no data-link is available. Due to the limited energy and power budget of such mobile devices, e.g., smartphones and drones, energy efficiency is of paramount importance for their usability. However, well-known CNNs for image classification require on the order of several GOP just to process a single frame, e.g., AlexNet requires 1.4 GOP [1] and ResNet-18 requires 3.62 GOP [3]. This combination of high computing power and energy efficient processing is not easily realized via off-the-self general-purpose processors [13]. Instead, highly optimized application-specific accelerators are required.

Previous work mostly evolves around *application-specific integrated circuits (ASICs)* [14]–[18] and *field-programmable gate arrays (FPGAs)* [19]–[21], while fewer research has been conducted regarding the design of more flexible accelerators, e.g., by offering programmability through an *instruction-set architecture (ISA)* [22]–[24]. While ASICs can offer the desired energy-efficiency gains, they often

come at the expense of being hardwired solutions with very limited to no flexibility. Accelerators based on FPGAs offer a high throughput and large flexibility which, however, is paid for by smaller energy efficiencies compared to ASICs. To achieve the required efficiency gains, many different techniques like quantization [25]–[27], pruning [28], [29], compression [30], [31], and dataflow optimization [32], [33] can be incorporated. Eyeriss [14] is a well-known ASIC comprised of a 2-D processing array with 16×16-bit multipliers that utilizes a specific dataflow scheme named *row stationary* to maximize local data reuse. In addition, data compression via run-length encoding is used to minimize the total off-chip transfer size and multiplications are zero-guarded, i.e., only if both operands are non-zero the computation is actually executed. NullHop [18] is another ASIC that aims at maximally exploiting the inputs' sparsity by skipping multiplications in case of zero-valued operands. It also utilizes 16×16-bit multipliers and minimizes off-chip transfers by using a custom compression scheme based on *non-zero value lists* and *sparsity maps*. Both of these ASICs do not make use of aggressive quantization (e.g., 8 bits and below), but still achieve high throughputs and energy efficiencies, which are paid for by their lack of flexibility. Envision [22] promises to improve upon that aspect by using an ISA and by employing aggressive quantization, i.e., reducing the multipliers' wordwidth on a per-layer basis and employing a technique called *dynamic-voltage-accuracy-scaling (DVAS)* that reduces the supply voltage while keeping the clock frequency constant. Envision is comprised of an *application-specific instruction-set processor (ASIP)* with tightly integrated 2-D processing array that features aforementioned dynamic-precision multipliers, which have zero-guarding capabilities as well. Although it offers increased flexibility through its ISA, the underlying computations must be rigidly scheduled onto its 2-D processing array. Also, no mentioning of extended pooling support, e.g., for average pooling, and no support for *fully-connected (FC)* layers or *depthwise-separable (DW)* convolutions is found. The *deep-learning specific instruction-set processor (DSIP)* [23] is another ASIP that focuses especially on the aspect of scalability by making use of a master-slave architecture which allows to chain together multiple core instances to scale its overall processing power. However, DSIP does not make use of aggressive quantization and there is no mentioning of, e.g., support for FC layers or DW convolutions.

In this article, we extend our work on the ConvAix ASIP from [34], striving to improve upon several limitations in existing programmable architectures. Instead of relying on a single large 2-D processing array, ConvAix utilizes several smaller *single-instruction multiple-data (SIMD)* processing lanes that are fully accessible via primitives in the C-code. Compared to [34], the following improvements are presented:

1) State-of-the-art range-based quantization with per-layer wordwidths is employed.

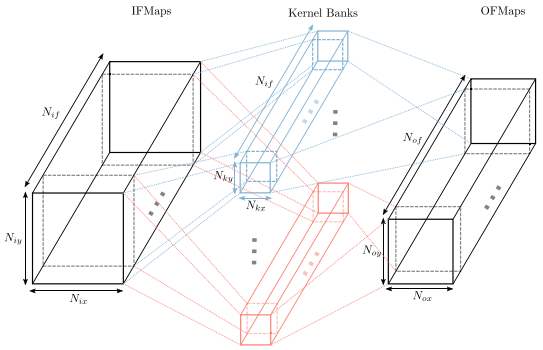

FIGURE 1. Basic structure of a convolutional layer.

2) The original ISA is extended from a 4-slot VLIW to an 8-slot VLIW pipeline to further foster ILP.
3) Subword parallelism [35] is leveraged to increase both throughput and energy efficiency at the same time.
4) Each multiplier unit is zero-guarded to exploit sparsity in the feature maps.
5) New CNN architectures, e.g., MobileNet [4] and ResNet [3], are benchmarked to showcase the ASIP's flexibility.

The dataflow can be optimized via ConvAix's software programmability, i.e., by using CNN-specific loop tiling factors that minimize off-chip transfers. The main intent of this article is to present novel insights into both the feasibility and achievable efficiency & performance of a programmable processor compared to less flexible designs. To demonstrate said flexibility, highly optimized software kernels not only for regular convolutions, but also for DW convolutions, residual blocks, and FC layers are implemented. These kernels have almost no limitations, e.g., arbitrary filter sizes, channel dimensions, etc., can be used with the only limitation being the on-chip memory's size.

The outline for the rest of this article is as follows. Section II introduces the basics of CNN processing and possible efficiency enhancing opportunities. Section III describes the ConvAix ASIP's instruction pipeline and arithmetic units in more detail. In Section IV, first a brief overview of the convolutional layer's software implementation is given, followed by an analysis of the placed and routed processor. Section V starts by investigating both its performance and efficiency using a synthetic CNN benchmark at first. Afterwards, the well-known CNNs AlexNet [1], MobileNet [4], ResNet-18 [3], and VGG16 [2] are simulated to obtain application-specific results. Section VI concludes this article.

II. CNN PROCESSING BACKGROUND

CNNs are composed of multiple consecutive layers that can have different types, e.g., convolutional layers, FC layers, pooling layers, etc. Usually, the majority of them are of the convolutional and pooling type, while only the last one is of the FC type, e.g., to generate per-class probabilities in case of a classification task. Fig. 1 depicts the basic concept

TABLE 1. Convolutional layer dimensions.

	Filters		IFMaps			OFMaps		
	Height	Width	Height	Width	Chan.	Height	Width	Chan.
Dimension	N_{ky}	N_{kx}	N_{iy}	N_{ix}	N_{if}	N_{oy}	N_{ox}	N_{of}
Tiling	T_{ky}	T_{kx}	T_{iy}	T_{ix}	T_{if}	T_{oy}	T_{ox}	T_{of}
Unrolling	P_{ky}	P_{kx}	P_{iy}	P_{ix}	P_{if}	P_{oy}	P_{ox}	P_{of}

of the convolutional layer: It consumes a 3-D input tensor,[1] often jointly referred to as *input feature maps (IFMaps)*, and convolves it with a number of filter kernels to generate a 3-D output tensor, the so called *output feature maps (OFMaps)*. In this article, the different feature map and kernel dimensions are denoted according to the scheme introduced in [32], which is summarized in Table 1. In addition to denoting the pure dimensions, Table 1 also denotes tiling and unrolling factors used later on in the software implementation.

The computations involved in this convolution are concisely defined using a *sum of products (SOP)* representation as follows:

$$\mathbf{O}[c_o, y_o, x_o] = \sum_{c_i=0}^{N_{if}} \sum_{k_y=0}^{N_{k_y}} \sum_{k_x=0}^{N_{k_x}} \mathbf{W}[c_o, c_i, k_y, k_x] \times \mathbf{I}[c_i, y_o s, x_o s] + \mathbf{B}[c_o] \quad (1)$$

where $\mathbf{I}$ represents the IFMaps of dimension N_{if} x N_{iy} x N_{ix}, $\mathbf{O}$ the OFMaps of dimension N_{of} x N_{oy} x N_{ox}, and $\mathbf{W}$ the filter weights of dimension N_{of} x N_{if} x N_{ky} x N_{kx}. Furthermore, s denotes the stride of the convolution, c_o the output channel number, while y_o and x_o denote the pixel's spatial position within its respective OFMap. It is common practice to reduce the OFMaps' spatial dimension by using strided convolutions ($s > 1$) and/or by applying, e.g., max-pooling or average-pooling at the layer's output. More recent CNNs like MobileNet [4] make use of DW convolutions that split the regular 3-D convolution across both spatial (height, width) and channel dimensions, referred to as depthwise and pointwise convolution respectively. In doing so, the overall number of parameters (filter weights) and computations is reduced, leading to a smaller memory footprint and less computational requirements for the CNN. Reference [3] introduces ResNet, a specific CNN topology that makes use of residual blocks that enable the training of very deep CNNs. The basic concept is to have two paths within each residual block, one that calculates a regular convolution, and a second path that preserves a residual representation of the input, thereby allowing for easier back-propagation of the gradients during training. At the output, both paths are joined via an element-wise addition. A flexible processing architecture should be capable of handling such variations in the convolutional layer, which is why both MobileNet and ResNet are implemented and benchmarked in this article. Since the convolutional layers constitute the majority of computations found in modern CNNs (usually well above 90 %), the optimization focus is set on them hereafter. Nevertheless, a well-rounded accelerator also possesses the ability of computing, e.g., FC and pooling layers efficiently.

1. In case of batch-processing, each input-tensor gets a fourth dimension with size equal to the batch-size.

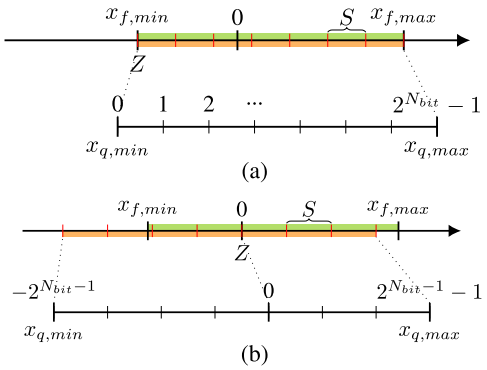

FIGURE 2. Asymmetric (a) and symmetric (b) integer-only quantization.

A. QUANTIZATION AND PRUNING

Among the different efficiency enhancing techniques for CNNs, quantization is probably the most common and important one. As stated in [36], multiplications, which build the foundation of CNN processing, cost over 18 times more energy per operation if 32-bit floating-point is used instead of 8-bit fixed-point arithmetic. Research on quantization has been conducted regarding the use of specialized number formats, e.g., by using logarithmic quantization [37] or by quantizing CNNs to single bits [38]. The use of such highly specialized number formats does, however, stand in contrast to the initial goal of flexibility. Because of this, a very efficient quantization scheme using integer-only arithmetic, as proposed in [26], is used here as well. The method proposed in [26] decomposes a floating-point number x_f into a quantized integer value x_q and its corresponding scale S and zero-point Z as follows:

$$x_f = S(x_q - Z) \quad (2)$$

Given the limited range of the values found in CNNs, i.e., weights are usually close to zero and feature maps do not exceed a certain upper bound, it is possible to determine an appropriate scale and zero-point offline based on the minimum ($x_{f,min}$) and maximum ($x_{f,max}$) values observed during a calibration run. This quantization method thereby allows to optimally utilize the underlying machine wordwidth. Although the authors of [26] make use of an asymmetric scheme, meaning that the effectively covered range of quantized values can contain more positive than negative values or vice versa, there is also the option of using a symmetric scheme. Fig. 2 visualizes the differences in both schemes, where N_{bit} refers to the number of bits available for each integer value.

While asymmetric quantization maps x_f to an unsigned integer range, thereby allowing to tailor the covered range

TABLE 2. Average wordwidths after quantization.

		AlexNet	ResNet-18	MobileNetV1	VGG16 (BN)
Average wordwidth (in bits)	Weights	6.9	6.5	8	6.9
	Activations	6.9	6.8	8	7.1

TABLE 3. Pruning and quantization results.

		AlexNet	ResNet-18	MobileNetV1	VGG16 (BN)
Top-1 accuracy (in %)	Baseline (fp32)	56.53	69.63	68.85	73.46
	+Pruning	56.32	69.70	69.60	-
	+Quantization	55.63	67.22	66.79	71.40
Top-5 accuracy (in %)	Baseline (fp32)	79.00	89.07	88.69	91.49
	+Pruning	79.29	89.24	89.26	-
	+Quantization	78.86	87.57	87.34	90.53
Activation sparsity (in %)[a]	Baseline (fp32)	33.68	39.67	56.65	59.01
	+Pruning	30.29	39.12	59.18	-
	+Quantization	32.82	46.49	47.77	63.90
Weight sparsity (in %)[a]	Baseline (fp32)	0.00	0.00	0.00	0.00
	+Pruning	50.87	62.67	48.89	-
	+Quantization	54.26	62.77	55.17	15.09

[a] Convolutional layers only.

very closely to the expected range, its symmetric sibling uses signed two's complement numbers and can only tailor its range in a more coarse manner. However, one big disadvantage of the asymmetric scheme is that simple *multiply-accumulate (MAC)* operations, as depicted in (1), require an additional accumulation of the IFMaps in case of $Z \neq 0$. This is not the case for the symmetric quantization, i.e., $Z = 0$, making it the choice for this work. The SOP calculated during convolution, as depicted in (1), can therefore be rewritten as follows:

$$S_O O_q = \sum_{i=0}^{N} \left(S_W W_{q,i}\right)\left(S_I I_{q,i}\right)$$
$$O_q = \underbrace{\frac{S_W S_I}{S_O}}_{M} \sum_{i=0}^{N} W_{q,i} I_{q,i} \text{ with } M = 2^{-n} M_0 \qquad (3)$$

As can be seen in (3), a small overhead due to quantization remains in the form of an output scale-adaptation, i.e., multiplying the SOP with M, which is a floating-point value. In practice, M is usually much smaller than 1 and the multiplication is realized by decomposing it into a multiplicative value $1 > M_0 \geq 0.5$ and a right-shift by n [26], which are both easily realized using integer-only arithmetic.

While in [26] a uniform wordwidth is assigned to all layers, the authors of [39] assign individual wordwidths to both weights and feature maps on a per-layer basis. We adopt their method for this work and quantize the selected CNNs to a per-layer wordwidth between 4 and 8 bits using the symmetric scheme.[2] Additionally, we mildly prune the different CNNs using the *automated gradual pruner (AGP)* [29] before quantization is applied. For this purpose, the *Distiller* framework [40] is used. The average wordwidths assigned to each CNN are summarized in Table 2. Table 3 depicts the resulting top-1 and top-5 accuracies of the quantized and pruned networks for the well-known ImageNet classification task, together with the average sparsities found both in their weights and activations (IFMaps). Since quantizing MobileNet has proven to be more difficult than the other CNNs, a uniform wordwidth of 8 bits for all layers is used here. Furthermore, VGG16 was not pruned due to its large size and the corresponding long expected runtime of the AGP algorithm.

III. ASIP ARCHITECTURE

Fig. 3(a) depicts the pipeline of the ConvAix ASIP. Since CNNs require a large degree of parallelism, ConvAix leverages *instruction-level parallelism (ILP)*, in the form of an 8-slot VLIW instruction-set, and *data-level parallelism (DLP)*, by implementing SIMD vector operations. The degree of DLP, i.e., the number of elements in a vector, is denoted by N_{vsize} and it is fixed to $N_{vsize} = 16$ hereafter as this yields the best trade-off for our design in terms of area, energy, and runtime according to several parameter studies that we have conducted. For implementing the subsequently described processor architecture, we use the *Synopsys ASIP Designer* toolsuite [41].

The first two slots (*S0* & *S1*) are reserved for control and scalar operations, e.g., instructions that set up zero-overhead hardware loops or calculate pointer addresses using the *address generation units (AGUs)*. Additionally, in slot *S0* instructions that trigger the integrated *direct memory access (DMA)* controller can be issued. Slots *S*2 and *S*3 contain both scalar and vector load/store instructions in addition to specialized load-instructions accessing the dedicated *line buffer (LBuf)* module. More specialized vector instructions, e.g., used for calculating activation functions like the *rectified linear unit (ReLU)*, are housed in *S*2. The following slots *S*4 to *S*7 comprise the *vector arithmetic logic units (VALUs)* capable of executing various operations, e.g., element-wise vector addition, multiplication, fused MAC, etc. The 9-stage instruction pipeline consists of an *instruction fetch (IF)*, *instruction decode (ID)*, 6 *execute (E1..E6)*, and a *writeback (WB)* stage. All VALUs are deliberately placed in the later execute stages in order to allow fused VLIW instructions that combine a dual vector-load, e.g., filter weights and IFMaps, with subsequent vector MAC operations, resulting in a hazard-free schedule that enables a maximum utilization of the VALUs. The ASIP's program code is located in a dedicated single-port 32 KB SRAM, named *program memory (PM)*, while data is stored in a bank of 16x8 KB dual-port SRAMs, named *data memory (DM)*. A sophisticated memory controller, in which the DMA is also integrated, arbitrates accesses to the on-chip memories and handles seamless movement of data to and from off-chip DRAM memory. It also minimizes any stall-cycles introduced due to structural hazards, e.g., port conflicts. The off-chip interface itself is realized according to the AXI specification [42] and has a design-time configurable bus-width, which is set to 256 bits here. Two elementary

2. Each layer is processed using 8-bit integer arithmetic. Quantized values of layers with fewer than 8 bits are filled with zeros at their MSBs.

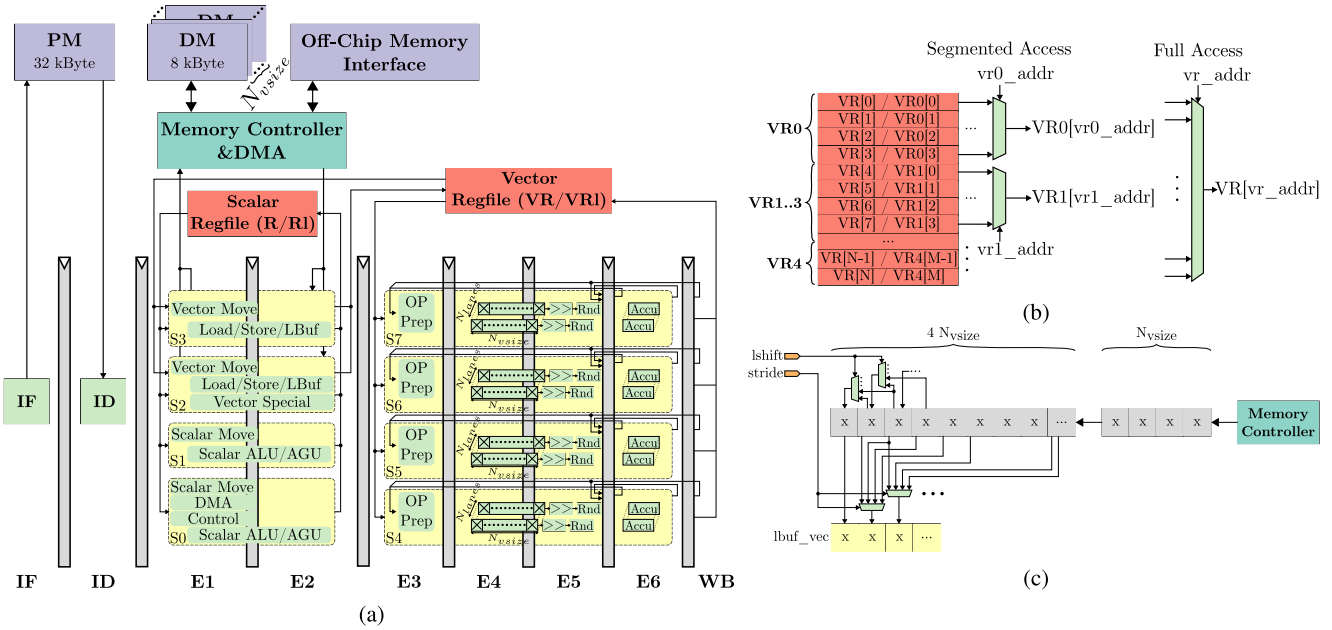

FIGURE 3. Pipeline overview of the proposed ConvAix ASIP (a), high-level structure of the vector register file (b), and schematic of the line buffer unit (c).

scalar data types are supported by the ASIP: 16-bit short integers (stored in register file *R* with space for 32 entries), and 32-bit long integers (stored in register file *Rl* containing 16 entries). Vectors can be either 16×16 bits (stored in *VR* of size 24) or 16x48 bits (stored in *VRl* containing 16 entries). To reduce the complexity of the port-multiplexing in the vector register files, only the load/store and vector move instructions have full access to all entries. The VALUs, on the other hand, only have access to smaller segments, as illustrated in Fig. 3(b). Additionally, the ASIP comprises several smaller special-purpose registers, e.g., used for storing the stack-pointer (*SP*), vector masks (*VG*), etc. These are not all shown here for brevity.

A. LINE BUFFER

The LBuf module is depicted in Fig. 3(c) and was designed with the goal of being able to efficiently load IFMap rows during CNN processing. While performing a convolution, it often happens that the same pixels within one row are repeatedly required in subsequent steps. Furthermore, especially in case of strided convolutions, the on-chip SRAM memory can quickly become a bottleneck since each computational step requires a vector of non-adjacently stored IFMap pixels. This scenario is illustrated in Fig. 5, where a 3x3 convolutional filter kernel w (blue) is convolved with an IFMap (yellow), stored in row-major fashion, using a stride $s = 2$. Each overlap between a weight (in blue) and a yellow square marks a required input pixel. The proposed LBuf unit uses an internal buffer with size for $4N_{vsize} = 64$ elements with an additional pre-fetch buffer containing 16 elements. It is user-configurable in the C-code and immediately starts to pre-fetch an IFMap row once it is activated. For this purpose, it is directly coupled to the memory controller and is capable of simultaneously accessing both ports of the on-chip SRAM. Once initialized, it allows for easy access to both unstrided and strided versions of the desired IFMap row, in which the supported strides are limited to $s \in \{1, ..4\}$. After each load, the internally saved row is left-shifted by a user-specified number of pixels, which can be between 0 and 3. This feature can be used, e.g., to implement dilated convolutions without any overhead. Depending on the user-settings, e.g., the maximum row-length and the left-shift count, the LBuf unit automatically fetches additional IFMap pixels once its pre-fetch buffer is depleted.

B. VECTOR ALUS

Each of the slots *S*4..*S*7 contains one VALU, in which each VALU itself is comprised of several parallel instantiated multipliers, as depicted in Fig. 4(a). Especially in CNN processing, a very large degree of local data reuse is possible, e.g., by performing calculations on multiple output channels in parallel while maximally reusing the available IFMaps. ConvAix supports this by leveraging SIMD parallelism in two dimensions as follows: Each VALU contains 4 separate vector lanes (the rows of multipliers and adders in Fig. 4(a)), i.e., the first dimension, whereas each row contains 16 processing units in itself (the individual multipliers and adders in Fig. 4(a)), thereby constituting the second dimension. Based on the vector-type operands *op1* and *op2*,[3] fetched from *VR* in the E3 stage, the individual lanes' vector operands are created via a software-configurable broadcast network that allows to select arbitrary elements from the input vectors (*op1*, *op2*) and use them to compose the lane-wise input

3. It is also possible to load a scalar value from *R* and to directly extend it to a vector, which then forms one of the VALU's operands.

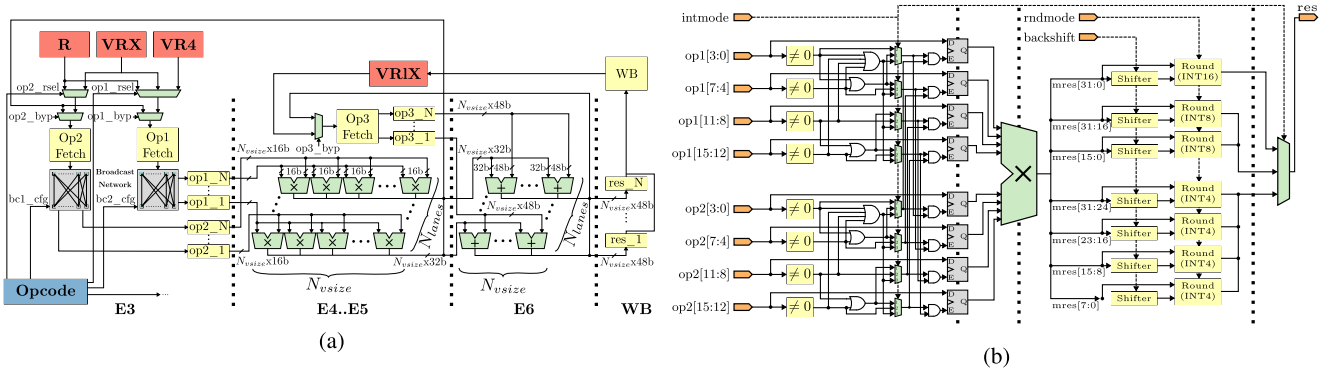

FIGURE 4. High-level schematic of a VALU as instantiated in each of the slots *S4.S7* (a), and schematic of a multiplier unit (b).

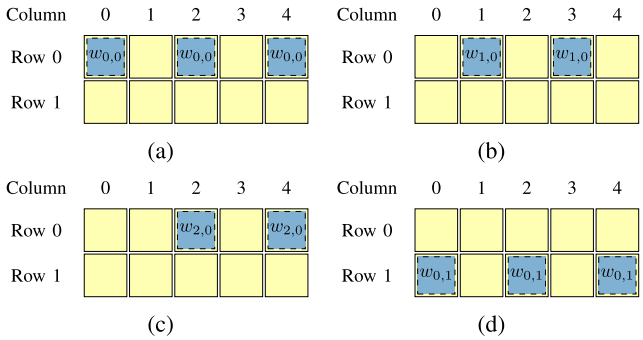

FIGURE 5. Unrolled steps of strided convolution, exemplary depicted for a 3x3 filter and a stride of 2.

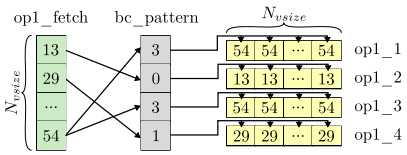

FIGURE 6. Operand broadcasting to lane-wise inputs.

vectors *op1_1..op1_4* and *op2_1..op2_4*. Fig. 6 schematically depicts this for the first operand *op1*. In addition, it is possible to use the originally fetched operands without further modification, i.e., passing them to each lane-wise input vector.

The 16×16-bit multiplier units are two-stage pipelined (stages *E4* and *E5*) to achieve the target frequency, whereas the following accumulators are non-pipelined. To provide sufficient headroom in case of accumulating large channel counts, the accumulators are 48-bit wide. Their operands *op3_1..op3_4* are fetched from fixed positions within the wide vector register file *VRl*, thereby avoiding the need to encode individual indices and circumventing the necessity of additional ports in *VRl*. Furthermore, operand bypassing within each VALU is fully implemented to avoid data hazards as much as possible, e.g., via allowing to directly forward a multiplication result from *E5* to *E3*, as depicted in Fig. 4(a) (for brevity, only a few exemplary bypasses are shown in this figure). While most of the four VALUs are homogeneous, i.e., they are all instantiated based on the same template, slight modifications are added to the first one in *S4*: Because the range-based quantization accumulates lower precision multiplication results, i.e., 16×16-bit multiplications, in higher precision registers (48 bits), it is necessary to have a limited number of multipliers with larger wordwidth, which are used to adapt the output-scales of the accumulated results, as depicted in (3). Since it is possible to execute the right-shift by n before multiplying with M_0 and because we have empirically found that n is always large enough to shift the 48-bit values into a 32-bit range,[4] a limited number of 32x32-bit multipliers are necessary. These are instantiated only in the first lane of the first VALU, i.e., 16 out of 256 multipliers have double the wordwidth. In addition to fused MAC operations, each VALU is capable of executing, e.g., additions/subtractions, bit-wise shifts, and logical operations (AND, XOR, etc.). These operations are, however, not instantiated in multiple lanes like the multipliers and accumulators are.

C. MULTIPLIER UNITS

Fig. 4(b) depicts the individual subword-parallel multiplier units. Instead of computing a single 16×16-bit (*int16*) multiplication, they can be configured at runtime to alternatively execute two 8x8-bit (*int8*) or four 4x4-bit (*int4*) multiplications using the same input words.[5] The subsequent accumulator units are also implemented with the same subword parallelism which, however, is not depicted here in detail for reasons of brevity. Based on the high sparsity found in CNNs, the multiplier's inputs are zero-guarded, i.e., if at least one of the two respective multiplicands is zero, the corresponding pipeline registers are disabled via clock-gating to prevent the multiplier cells from switching. Furthermore, the output is fed into a runtime configurable shifter used to back-shift the result into a desired range, which is then fed into a rounding unit implementing the *round-to-nearest even* scheme in order to minimize any

4. Even if n was not large enough, it is always possible to trade-off some precision in the last few digits by performing a larger right-shift in order to end up in a 32-bit range.

5. Synopsys DesignWare [43] is used for implementation.

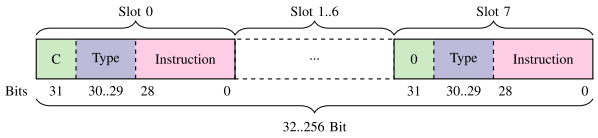

FIGURE 7. VLIW instruction encoding.

rounding bias inflicted otherwise. In practice, this configurable shifter is used together with lower-precision arithmetic modes (*int8*, *int4*) to ensure that no saturation in the accumulator registers occurs. Considering their wordwidth of 48-bit in total, this effectively leaves 24-bit per subword in case of the *int8* mode and 12-bit per subword for the *int4* mode to accumulate results. Taking into consideration the doubled wordwidth of the multiplier's output, 8 bits (*int8*) and 4 bits (*int4*) are left as accumulator headroom respectively, which is not always sufficient. The multipliers can then be configured to perform a larger back-shift, thereby preventing saturation at the cost of some increased rounding-noise in the least-significant digits. Based on offline experiments, it is - in our experience - always possible to find a suitable setting for the *int8* mode, while quantization to 4-bit proves more difficult and might require expensive re-training of the CNN.

D. INSTRUCTION-SET ARCHITECTURE

As initially mentioned, the presented ASIP comprises an 8-slot VLIW instruction-set which adds to the flexibility of the core. Each of the per-slot instruction words is 32-bit wide, resulting in 256-bit wide VLIW instructions. Since especially the vector instructions (*S*4..*S*7) are only utilized in the application-specific parts of the overall program code, filling up each VLIW instruction with *nops* would quickly crowd the available program memory. Instead, we dynamically decode each VLIW instruction based on its length and contained slot-types. As illustrated in Fig. 7, each 32-bit instruction starts with a continue-bit, indicating whether another slot follows, and its slot-type is encoded as well, making it possible to, e.g., have a VLIW instruction containing only vector instructions.

This flexibility in the ISA together with the VALUs' configurability allows to seamlessly create computational kernels, as exemplary depicted for a 1-D convolution in Listing 1. At the innermost loop-level (lines 20-25), 1 LBuf load, 1 vector load, and a vector MAC operation are executed. These can efficiently be combined into a single VLIW instruction that is repeatedly executed by means of a zero-overhead loop, as exemplary depicted in Fig. 8. Depending on the structure of the final program code, very high VALU utilization rates can be realized this way.

IV. IMPLEMENTATION

In the following, the implementation both from a software perspective, i.e., the structure of the computational kernels,

```
1  void conv1d_4ch(vint* pvI, vint* pvW, // IFMaps, Weights
2                  uint16_t* pO,          // OFMaps
3                  uint16_t nix, uint16_t nox,
4                  uint16_t nkx, uint16_t stride) {
5    // Line buffer & arithmetic setup
6    lbuf_set_stride(stride);  // strided fetching
7    set_backshift(0);         // backshift after mult
8    set_int_mode(INT16_MODE); // int16, int8, int4
9    set_round_mode(RND_NEAREST_EVEN | RND_SATURATE);
10   vbool vmask = to_vbool(0xFFFF); // guard mask
11   uint32_t pattern = valu_bc_pattern(0, 2, 4, 6);
12
13   vlint  vCh1, vCh2, vCh3, vCh4;
14   for(uint16_t ox=0; ox<nox; ox+=16) {
15     // set starting address of IFMap row
16     lbuf_set_addr((uint16_t*)&pvI[ox*stride]);
17     for(uint16_t kx=0; kx<nkx; kx++) {
18       vint vI = lbuf_read_sl(1); // lshift by 1
19       vint vW = pvW[kx*4];
20       vmacX4(vI, vW, pattern,      // op1, op2, pattern
21              vCh1, vCh2, vCh3, vCh4, // op3_1,..,op3_4
22              vmask,                  // guarding mask
23              vCh1, vCh2, vCh3, vCh4);// dst1_1,..,dst1_4
24     }
25     // store the right-shifted and saturated vectors
26     masked_vstore(vmask, (vint*)&pO[0*nox + ox],
27                   vrshift_sat_extlow(vCh1, 8));
28     [...]
29   }
30 }
```

Listing 1. Code example of a rudimentary 4-channel 1-D convolution demonstrating the ASIP's programmability.

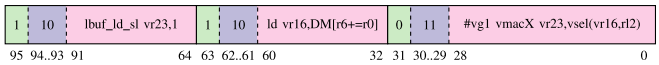

FIGURE 8. VLIW instruction for innermost loop body in Listing 1.

as well as the physical implementation using a modern 28 nm CMOS technology are presented. It is important to keep in mind that both software and hardware development were conducted interdependently to find a good trade-off between achieving even higher efficiencies and throughputs, i.e., via introducing more specialized instructions, and maintaining a sufficient degree of flexibility in the final instruction-set. Aforementioned flexibility is showcased by not only implementing one type of convolutional layer, but by implementing different kernels like DW convolution and residual blocks too.

A. SOFTWARE

As mentioned earlier, each computational kernel is written in the C-language and compiled to efficient machine code using a retargetable compiler included in the *ASIP Designer* toolsuite that was used here [41]. While regular constructs in the code, e.g., loops, branches, and pointer arithmetic, are automatically mapped to the corresponding instructions, using the VALUs and specialized units like the LBuf requires more fine-tuning in some instances. This is demonstrated in Listing 1 by the use of some processor-specific primitives, e.g., *lbuf_set_stride* and *lbuf_read_sl*. Furthermore, it is often desirable to perform manual code optimization, e.g., by unrolling certain loops or by optimizing memory access patterns and pointer arithmetic in such a way that less overhead[6] is generated by the compiler. It is

6. The term *overhead* here refers to any processing cycles spent that do not contribute to the final computational result, e.g., convolution.

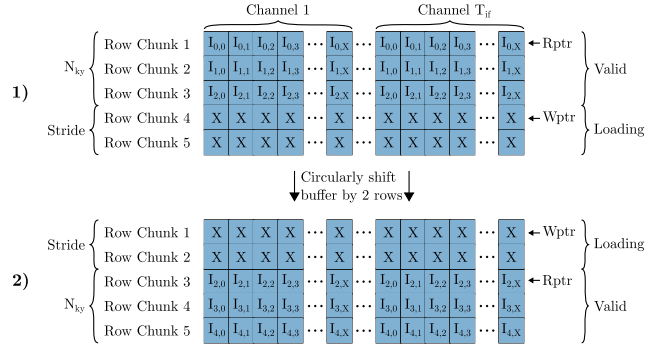

FIGURE 9. **Circular buffer storing IFMap rows for a filter kernel of size 3x3 and a stride of 2.**

clearly not possible to describe all these optimization steps in detail here, just like it is not possible to dissect the entire code for each computational kernel. In general, each CNN layer that was implemented uses the same base function (hereafter also referred to as *wrapper*) which comprises the code required to move data to/from off-chip memory and to perform basic calculations, e.g., determining on-chip buffer sizes, setting up pointers, etc. This wrapper function is configurable via a *struct* that stores relevant parameters, e.g., the layer dimensions as introduced in Table 1, pointers to the off-chip data (IFMaps, OFMaps, weights), arithmetic configuration (backshift, arithmetic precision), etc. Embedded into this dataflow management wrapper are calls to highly optimized kernels that work exclusively on the on-chip data fetched using the surrounding wrapper. We include kernels for regular convolutions using arbitrary filter sizes as well as kernels optimized for certain filter sizes (1x1, 3x3, 5x5, 7x7, 11x11). Furthermore, separate kernels for the depthwise and pointwise parts of DW convolution are available. Subsequent CNN modules, e.g., activation, pooling, and element-wise addition with a residual, are fused with the preceding kernel to maximize data reuse within the on-chip memories. Each computational kernel usually consists of a few hundred lines of C-code, while the wrapper consumes roughly 700 lines of code. The resulting machine code also usually consumes a few hundred instructions per kernel. Algorithm 1 depicts the overall dataflow scheme of a regular convolution, in which the inner computational kernel resides in lines 11-23, while the remaining code belongs to the wrapper. Since no hardware caches are used, the programmer is responsible for managing all storage on his own, as evident by the necessity to explicitly trigger DMA transfers in the code. To maximally exploit the available on-chip storage, circular buffers are used for storing IFMaps and OFMaps. Since we first calculate one entire row of multiple OFMap or *partial sum (PSum)* channels, depending on the selected loop tiling factors T_{if} and T_{of}, the IFMaps used in that step can partially be reused once the filter kernels are vertically shifted to compute the next row. Fig. 9 depicts the implemented buffer structure.

Another advantage of this setup is that computations and DMA transfers can be interleaved by marking parts of the

Algorithm 1: Convolutional Layer as Implemented on the ConvAix ASIP

Input: CNN parameters: $N_{kx/ky}, N_{if}, N_{of}$, etc.
Tiling & unrolling parameters: $T_{of}, T_{if}, P_{ox}, P_{of}$
Configuration flags: *is_resid_conv*, *apply_actfunc*, *pooling_active*

```
1  N'_of = ⌈N_of / T_of⌉,  N'_if = ⌈N_if / T_if⌉
2  for t_of = 0; t_of < N'_of; t_of ← t_of + 1 do
3      for t_if = 0; t_if < N'_if; t_if ← t_if + 1 do
4          DMA_Load_Filters()
5          DMA_Load_IFMaps_Initial()
6          DMA_Load_PSums_Initial()
7          for y_o = 0; y_o < N_oy; y_o ← y_o + 1 do
8              DMA_Load_IFMaps_Next()
9              DMA_Load_PSums_Next()
10             DMA_Load_Residual()
11             for c_o = t_of · T_of; c_o < (t_of + 1) · T_of;
               c_o ← c_o + P_of do
12                 for x_o = 0; x_o < N_ox; x_o ← x_o + P_ox do
13                     SRAM_Load_PSum()
14                     for k_y = 0; k_y < N_ky; k_y ← k_y + 1 do
15                         LBuf_Setup()
16                         for c_i = t_if · T_if; c_i < (t_if + 1) · T_if;
                           c_i ← c_i + T_if do
17                             for
                               k_x = 0; k_x < N_kx; k_x ← k_x + 1
                               do
18                                 LBuf_Load()
19                                 MAC(P_ox · P_of per cycle)
20                     if t_if + 1 == N'_if then
21                         if is_resid_conv then
                           Add_Residual(); if apply_actfunc
                           then Apply_Actfunc();
22                     SRAM_Store_OFMap_or_PSum()
23             if pooling_active then Apply_Pooling();
               DMA_Store_OFMap_or_PSum_Row()
```

buffers as valid, i.e., these can be used to perform computations, while other parts are currently used to load new data. The same scheme is applied to loading residuals and storing the OFMaps/PSums. As indicated in line 19 of Algorithm 1, at the innermost loop level $P_{of} = 16$ OFMap channels are unrolled (each channel is assigned to a single lane of the total 4 lanes per VALU). Each OFMap channel in itself is then also unrolled along its x-dimension with an unrolling factor of $P_{ox} = 16$, which is exactly the vector element count of each lane in the VALUs. Similar to the example shown in Listing 1, these multiple vector MAC operations together with the filter load and LBuf access are fused into a single VLIW instruction that is iterated over via a zero-overhead hardware loop.

To utilize the subword-parallel processing abilities of ConvAix, only comparatively small changes to the existing code are necessary. Since the entire architecture is already designed with this use-case in mind, a C-code primitive can be used to switch between the different modes (int16, int8, and int4). This mode-flag is stored in a global configuration register to which all functional units within the processor have access. For example, in case the int8-mode

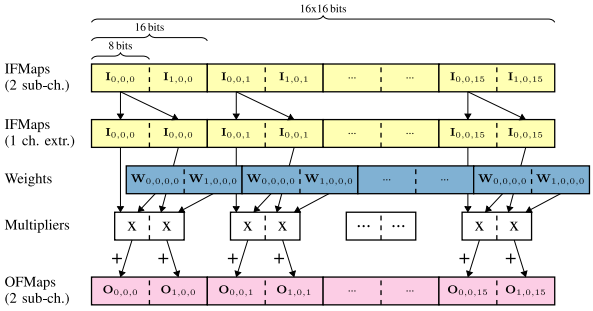

FIGURE 10. IFMaps, OFMaps, and weights packed into vectors in case of subword-parallel processing using the int8-mode.

is activated, the VALUs automatically treat each 16-bit word as two separate 8-bit words, i.e., performing all arithmetic and logic operations on the individual subwords and re-combining them into a 16-bit word in the end. Also, instructions used, e.g., to broadcast scalar values into vectors, to shift and saturate the accumulator registers, etc., access the same mode-flag and behave accordingly. Using the int8-mode, instead of calculating $P_{of} = 16$ output channels in parallel, $P_{of} = 32$ channels are calculated (the int4-mode calculates 64 output channels in parallel). The only necessary change in Algorithm 1 is to add another loop within the innermost body that iterates over the subword channels (this loop is unrolled). Of course, additional complexity is added to the data pre-processing, i.e., correctly packing the CNN's weights and biases into 16-bit words, but this is done offline once. Also, the IFMaps for the very first layer must be correctly packed, e.g., for the int8-mode two pixels at the same spatial position (x & y coordinate) in two adjacent channels are packed into a single word. If the channel count is not a multiple of 2 (int8) or 4 (int4), the remaining subword-channels are zero-padded. The broadcast network of the VALUs is then used to extract the individual single-channel IFMap rows for subsequent processing. Fig. 10 exemplary depicts this process and the packing scheme for the int8-mode.

B. SYNTHESIS & PLACE AND ROUTE

To obtain accurate performance, power, and area results for the proposed ASIP, we implemented the design using a modern 28 nm CMOS standard cell library provided by TSMC. The supply voltage is set to 1.0 V and the library used in this article is characterized for a typical process corner, operating at a temperature of 25 °C. Logic synthesis was performed using *Synopsys Design Compiler P-2019.03*, while *place and route (PnR)* was executed using *Cadence Innovus 18.10*. Fig. 11(a) shows the final layout of the ASIP, while Table 4 summarizes the relevant figures achieved after PnR. The processor occupies an area of 3.53 mm^2 (including SRAM macros), while running at a clock frequency of 513 MHz. As can be seen in the area breakdown in Fig. 11(b), both the SRAMs and the VALUs consume slightly over a third of the total area respectively.

TABLE 4. ConvAix implementation summary.

Technology	28nm 1P8M CMOS
Supply voltage [V]	1.0
Gate count[a] [kGE]	4036
Core area [mm^2]	3.53
On-chip SRAM [kB]	160
Clock frequency [MHz]	513
#MAC-Units	256
Arithmetic precision	4, 8, 16-bit progr.
Peak throughput [GOP/s]	262.6[b], 1050.4[c]

[a] Logic-only in NAND-2 equivalents.
[b] Using the int16 mode.
[b] Using the int4 mode.

V. RESULTS AND DISCUSSION

The different CNNs selected as benchmarks in this article are simulated using the post-layout, fully back-annotated netlist of the processor. Based on these very accurate simulations, switching activity is recorded and the average power consumption is estimated using *Synopsys PrimeTime PX*. Only for VGG16 we use switching activity obtained via RTL simulation that is subsequently annotated to the final post-layout netlist and propagated using zero-delay simulation.[7] All DRAM memory accesses are accurately traced in our testbench and the VALU utilization, i.e., the portion of time that the VALUs are performing computations, is calculated based on the ideal latency[8] and the actual measured latency, which includes times for DRAM accesses, stall cycles, control overhead, etc. Furthermore, the results for each CNN layer stated hereafter always include subsequent pooling layers, i.e., the time and energy spent for executing any existing pooling operation is included in ConvAix's final results.

A. SYNTHETIC CNN BENCHMARK

To explore the ASIP's maximum and minimum potential power consumption and its dependency on the IFMap sparsity, a synthetic CNN layer with the following configuration was created: $N_{if} = N_{of} = 256$, $N_{kx} = N_{ky} = 3$, $N_{ix} = N_{iy} = 16$, stride 1. Since the aim of this experiment is to stress the VALUs, the weights were randomly drawn from a normal distribution $\mathcal{N}(0, 1)$, while the inputs were drawn uniformly from the range [0, 15]. In a post-processing step, the IFMaps were randomly sparsified to emulate different degrees of IFMap sparsity found in actual CNNs. Fig. 12 depicts the resulting power consumption and energy efficiency values using the three different arithmetic modes. The maximum power of 972 mW is consumed using the int16 mode at 0 % sparsity, resulting in an effective energy efficiency of 176 GOP/s/W (the VALU utilization is 65 % here). If the int4 mode is used and 90 % of the IFMaps are sparse, a scenario often encountered in the last few layers of a CNN, only 340 mW of power is consumed and an

7. Based on empirical studies that we conducted using our design, we determined that these results are 10-15 % over-optimistic regarding power.
8. The ideal latency is calculated based on each layer's number of MAC operations divided by the accelerator's peak throughput.

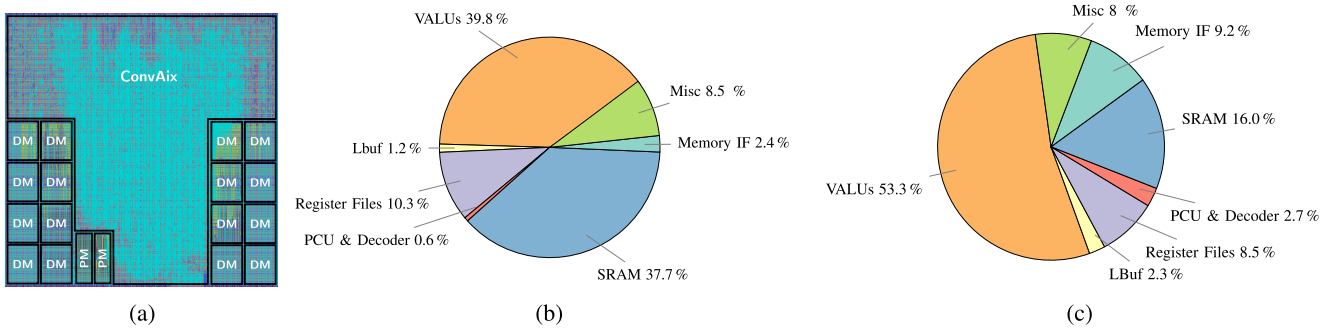

FIGURE 11. Layout view of the ConvAix ASIP after PnR (a), area breakdown (w/o filler cells) (b), and power breakdown for running all convolutional layers of AlexNet (c).

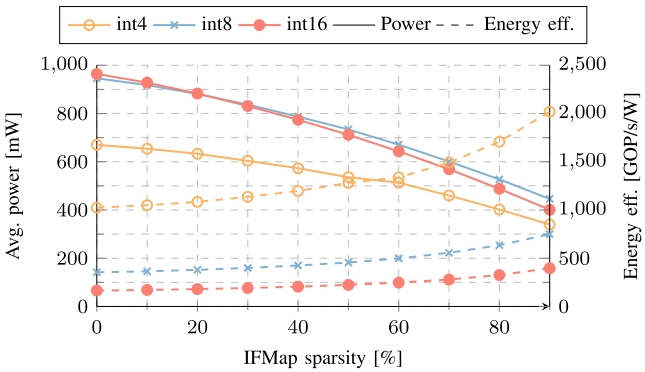

FIGURE 12. Power consumption and energy efficiency for different arithmetic modes and varying degrees of IFMap sparsity.

TABLE 5. Detailed results for AlexNet (batch size 1).

	C1	C2	C3	C4	C5	Total	FC1-3
Power [mW]	692	699	598	480	476	601	237
Energy [mJ]	0.55	0.88	0.39	0.45	0.31	2.58	1.94
Latency [ms]	0.80	1.26	0.66	0.93	0.65	4.29	8.18
DRAM accesses [MB]	0.28	0.47	0.90	1.67	0.93	4.25	57.29
MMAC/Frame	109.3	223.9	149.5	112.1	74.8	669.7	58.6
VALU utilization	0.52	0.68	0.87	0.46	0.44	0.59	0.03
Throughput [GOP/s]	274.5	356.7	455.5	240.8	230.2	312.3	14.3
Energy eff. [GOP/s/W]	396.7	510.3	761.7	501.7	483.7	519.6	60.5

energy efficiency of 2 TOP/s/W is achieved. In summary, for all arithmetic modes a spread in the power consumption of over 2.5× is observed.

B. ALEXNET

Fig. 11(c) breaks down the ASIP's power consumption in case it runs all convolutional layers of AlexNet. Over half of the entire power is consumed by the VALUs, while the SRAMs constitute the second largest consumer with 16 % of the total power. Table 5 gives a detail summary of executing all individual layers of AlexNet on the ConvAix ASIP, using a batch-size of 1 since real-time processing is the goal here. In total, executing all convolutional layers takes 4.29 ms at an average power consumption of 601 mW and a resulting energy efficiency of 520 GOP/s/W. On average, the VALUs are utilized 59 % of the time and a total off-chip transfer size of 4.25 MByte is required, which results to 0.0063 Byte/MAC. Based on these results, it is possible to run AlexNet at 232.5 frames/s, which is significantly faster than the performances achieved by comparable accelerators. Processing the FC layers of AlexNet, however, takes an additional 8.18 ms, which is an inherent bottleneck of this CNN topology that can only be resolved by a massively increased memory bandwidth.

C. MOBILENET, RESNET-18 AND VGG16

Fig. 13 and Fig. 14 depict the per-layer latency, average power consumption, and energy efficiency of running MobileNet and ResNet-18 on the ConvAix ASIP, while the overall results are summarized in Table 6. Executing the convolutional layers of these CNNs takes 14.6 ms (ResNet-18) and 14.2 ms (MobileNet), or an equivalent 68.5 frames/s and 70.4 frames/s, respectively. Due to its depthwise convolution, which does not offer the opportunity of sharing weights as in the regular convolution, MobileNet only achieves a reduced VALU utilization of 15 % vs. 47 % for ResNet-18. Furthermore, based on this reduced VALU utilization, MobileNet achieves a lower energy efficiency of 256 GOP/s/W compared to 512 GOP/s/W in case of ResNet-18. At the same time though, it consumes less average power (313 mW vs. 486 mW), which ultimately leads to a higher processing efficiency of 16.0 frames/s/mJ compared to 9.7 frames/s/mJ for ResNet-18. Additionally, MobileNet only requires 12.3 MByte of DRAM accesses, while ResNet-18 consumes 20.7 MByte in total. In Fig. 14, some drops in the power consumption and energy efficiency are observed for the downsample layers of ResNet-18. These are explained by the layers' low computation-to-memory ratio, resulting from using 1x1 filter kernels and strided convolutions that cause one stall cycle each time the LBuf unit is reset. The slight drop in the last layers of both CNNs is explained by the relatively small feature map sizes (7x7), resulting in a sub-optimal usage of the MAC-units in each VALU. VGG16 shows by far the highest VALU utilization rate at 69 %, which is founded especially in its large feature map dimensionality that allows to spend more time in the innermost loop body (see Algorithm 1), thereby avoiding some control flow overhead introduced in the outer loops. Based

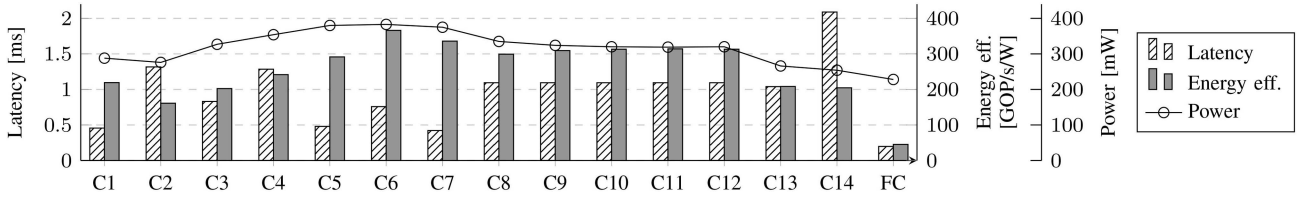

FIGURE 13. Results for convolutional layers of MobileNet. Stated values depict the combined depthwise and pointwise convolution.

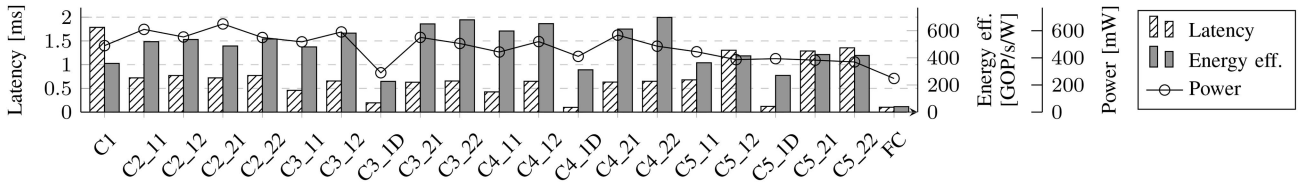

FIGURE 14. Results for convolutional layers of ResNet-18. Entries with a *D* suffix are downsample layers.

TABLE 6. Comparison to other state-of-the-art accelerators.

Reference	**Envision** [22]	**Eyeriss** [14]		**DSIP** [23]	**NullHop** [18]	**FCCM'2017** [20]		**WRA** [21]			**ConvAix (ISCAS'19)** [34]		**ConvAix (this work)**			
Technology	Silicon (40nm)	Silicon (65nm)		Silicon (65nm)	PnR (28nm)	FPGA (16nm)		FPGA (16nm)			PnR (28nm)		PnR (28nm)			
Architecture type	RISC + MAC array	ASIC		ASIP	ASIC	Xilinx ZCU102		Xilinx XCVU9P			ASIP		ASIP			
Core voltage [V]	0.85-0.92	1.00		1.20	1.00	0.85		0.85			1.00		1.00			
Gate count [kGE]	1600 [b]	1852 [b]		282 [a]	-	-		-			1293 [a]		4036 [a], 6380 [b]			
Core area [mm$\hat{2}$]	2.4 [b]	12.3 [b]		10.6 [b]	6.3 [b]	-		-			-		3.53 [b]			
On-chip SRAM [kB]	148.0	183.3		139.6	1088.0	32832.0		20224.0			144.0		160.0			
Clock frequency [MHz]	204	200		250	500	200		330			400		513			
#MAC-Units	256 [d]	168		64	128	2520		4096			192		256 [d]			
Peak throughput [GOP/s]	104.4 [e]	67.2		32.0	128.0	-		6300.0			153.6		262.6 [e]			
Arithmetic precision	1-16 progr.	16 fixed		16 fixed	16 fixed	16 fixed		8 fixed			1-16 progr.		4, 8, 16 progr.			
Filter sizes	all	1-12 (h), 1-32 (v)		-	1-7 (h/v)	3, 5 (h/v)		1-7 (h/v)			all		all			
Stride support	1-4 (h), all (v)	1-12 (h), 1-4 (v)		-	-	1		1-2 (conv), 1-4 (pool)			1-4 (h), all (v)		1-4 (h), all (v)			
FC support	-	none		-	none	yes		none			-		all			
Pooling support	maxpool (2x2, 3x3)	none		maxpool	maxpool (2x2)	maxpool		maxpool (1-7), avgpool (1-7)			maxpool (all), avgpool (all)		maxpool (all), avgpool (all)			
DW convolution	-	-		-	-	-		1-7 (h/v)			-		all			
Residual blocks	-	-		-	-	-		-			-		all			
Feature map size	-	-		-	512	all		1-1024			all [j]		all [j]			
Input channels	all	1024		-	1024	all		1-1024			all		all			
Output channels	all	1024		-	-	all		1-1024			all		all			
CNN Model	AlexNet	AlexNet	VGG16	AlexNet	VGG16	AlexNet	VGG16	AlexNet	VGG16	MobileNetV1	AlexNet	VGG16	AlexNet	VGG16	ResNet-18	MobileNetV1
Processing Time [ms]	21.1	25.9	1251.6	56.7	72.9	1.3	10.1	0.4	5.9	0.4	12.60	263.0	4.3	84.5	14.6	14.2
Power consumption [mW]	77.9	277.0	235.3	93.4	155.0	23.6	23.6	35000.0	35000.0	35000.0	228.8	223.9	600.9	552.3 [i]	486.4	313.1
Off-Chip I/O [MByte]	19.8 [f]	3.9 [f]	107.1 [f]	5.8	-	-	-	-	-	-	10.79	208.14	4.2	87.1	20.7	12.3
MAC utilization rate	0.61	0.77	0.36	0.74	3.29 [g]	1.00	3.02	1.29	1.93	0.99	0.69 [k]	0.76 [k]	0.59	0.69	0.47	0.15
Area efficiency [GOP/s/MGE]	39.7	27.9	13.2	-	-	-	-	-	-	-	82.23	90.26	48.9	57.0	39.0	12.6
Energy efficiency [h] [GOP/s/W]	597.8	433.8	241.9	846.1	2714.8	17.6	53.3	41.0	62.4	31.5	459	497	519.6	675.6 [i]	511.9	256.3
Processing efficiency [h,l][Frames/s/mJ]	21.2	12.5	0.0063	11.2	1.2127	10.3	0.2	82.9	0.3	65.4	27.5	0.0646	90.5	0.2607 [i]	9.7	16.0

[a] Logic-only. [b] Logic + SRAM macros. [c] Calculated from original figures via division by the batch-size. [d] Each unit is subword-parallel, i.e. it computes either 1xint16 operation, 2xint8 or 4xint4.
[e] Using the int16 mode. [f] Stated values are for compressed off-chip transfers. [g] Utilization is higher than 1.0 due to effective zero-skipping. [h] Power values were scaled to a uniform 28 nm CMOS technology (operating at 1 V) according to the following formula: $P_{scaled} = P_{old}(L_{new}/L_{old})(V_{DD,new}/V_{DD,old})^2$. [i] Values determined based on RTL simulation due to enormous runtime. [j] Feature map size is only limited by the SRAM size but not by the ASIP itself.
[k] Convolution only, no pooling. [l] This metric is the inverse energy-delay-product: $\frac{1}{E_{proc} \cdot T_{proc}}$.

on this comparatively high utilization, it also achieves the highest energy efficiency at 677 GOP/s/W.[9]

D. COMPARISON TO STATE OF THE ART

Table 6 compares ConvAix with other existing accelerators known from literature. Since the CNN model used to benchmark each accelerator has a huge impact on the resulting figures, as evident, e.g., by the large discrepancies between AlexNet and VGG16 for Eyeriss [14], we present performance values for each CNN separately. While previous architectures are often specifically tailored towards a certain type of CNN, ConvAix supports a wide spectrum of different CNNs. In addition, only very few limitations in terms of, e.g., feature map sizes or channel counts apply to ConvAix, while other designs are often limited to a maximum number of channels (Eyeriss, NullHop, WRA), have no support for DW convolutions (all but one), offer limited pooling kernel sizes, etc. To gain this kind of flexibility, a price in the form of additional silicon area must be paid which, however, does not result in a decreased area efficiency compared to other accelerators. More importantly, our fully software-programmable ASIP achieves energy efficiencies of over 500 GOP/s/W, which is comparable to those of other hard-wired solutions, e.g., Eyeriss [14]. Furthermore, even though no off-chip compression is used in this work, ConvAix achieves the second lowest total DRAM transfer count for AlexNet and the lowest for VGG16. This is due to the use of 8-bit computations and the ability to fine-tune the dataflow in software, thereby allowing to optimally tile loops and maximize data reuse. FPGA based accelerators, such as [21] and [20] that both use Winograd decomposition to optimize the convolution, offer good flexibility and lowest processing

9. Based on RTL switching activity, which is 10-15% over-optimistic.

times in our comparison. This speed comes together with an increased power consumption and resulting lower energy efficiency compared to ConvAix and others. Although energy efficiency on its own is one relevant metric to look at, it is also important to consider the achieved performance (in frames per second) per energy spent. The reason being that high energy efficiencies can be obtained by using very low processing speeds and maximally reducing the power consumption by using aggressive dynamic voltage and frequency scaling (DVFS). Especially the FPGA accelerators perform good in this metric as they offer high processing speeds. Amongst the remaining competitors, ConvAix outperforms most of them regarding the processing efficiency with the only exception being NullHop [18].

Finally, as seen in this comparison ConvAix offers several advantages over existing ASIP solutions: compared to Envision [22] and DSIP [23], ConvAix shows improved processing speeds and efficiencies for all investigated CNNs, while exhibiting similar and sometimes even higher energy efficiencies. Furthermore, existing ASIPs have less flexibility than ConvAix, e.g., both Envision and DSIP are not capable of executing DW convolutions. In addition, these other ASIPs require more off-chip transfers, even though they use similar (DSIP) or even lower (Envision) arithmetic precision.

VI. CONCLUSION

In this article, a fully software-programmable, yet highly energy efficient ASIP for the acceleration of CNN inference was introduced and subsequently implemented in a modern 28 nm CMOS technology down to the placed and routed design. By exploiting several different levels of parallelism, i.e., ILP, DLP, and subword parallelism, the necessary computational abilities to execute modern CNNs, like ResNet-18 and MobileNet, at real-time speeds were demonstrated. To achieve the best possible energy efficiency and performance, algorithm-level optimizations, e.g., quantization and pruning, were applied to the benchmark CNNs in a first step. These optimized CNNs were executed using 8-bit integer arithmetic to leverage the subword-parallel VALUs of the ASIP, ultimately leading to energy efficiencies and performance figures on the same level as those achieved by less flexible solutions. However, although it is advisable to use aforementioned optimization techniques, our proposed design in no way depends on them since it is always possible to use a higher-precision mode, i.e., 16-bit computations. Finally, compared to other known accelerators the ConvAix ASIP offers increased flexibility via its ISA-based architecture. This was proven by implementing and benchmarking not a single CNN topology but four different ones, which exhibit diverse dataflow patterns and computational requirements.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1–9.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. [Online]. Available: arXiv:1409.1556.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[4] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: arXiv:1704.04861.

[5] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Comput. Vis. ECCV*. 2016, pp. 21–37.

[6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 779–788.

[7] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2015, pp. 1440–1448.

[8] T. Pohlen, A. Hermans, M. Mathias, and B. Leibe, "Full-resolution residual networks for semantic segmentation in street scenes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 3309–3318.

[9] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017.

[10] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 2980–2988.

[11] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Process. Lett.*, vol. 23, no. 10, pp. 1499–1503, Oct. 2016.

[12] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, 2005, pp. 886–893.

[13] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput CNN inference on embedded ARM big.LITTLE multicore processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2254–2267, Oct. 2020.

[14] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[15] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2014, pp. 609–622.

[16] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W convolutional network accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 11, pp. 2461–2475, Nov. 2017.

[17] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 243–254, Jun. 2016.

[18] A. Aimar *et al.*, "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.

[19] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *Proc. 21st Asia South Pac. Des. Autom. Conf. (ASP-DAC)*, 2016, pp. 575–580.

[20] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field Programmable Custom Comput. Mach. (FCCM)*, 2017, pp. 101–108.

[21] C. Yang, Y. Wang, X. Wang, and L. Geng, "WRA: A 2.2-to-6.3 tops highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 9, pp. 3480–3493, Sep. 2019.

[22] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, Apr. 2017.

[23] J. Jo, S. Cha, D. Rho, and I. Park, "DSIP: A scalable inference accelerator for convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 53, no. 2, pp. 605–618, Feb. 2018.

[24] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 393–405.

[25] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, vol. 48, 2016, pp. 2849–2858.
[26] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.
[27] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 1737–1746.
[28] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, vol. 1, 2015, pp. 1135–1143.
[29] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2017. [Online]. Available: arXiv:1710.01878.
[30] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015. [Online]. Available: arXiv:1510.00149.
[31] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis.*, Munich, Germany, 2018, pp. 815–832. [Online]. Available: https://doi.org/10.1007/978-3-030-01234-2_48
[32] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2017, pp. 45–54.
[33] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, Jun. 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.54
[34] A. Bytyn, R. Leupers, and G. Ascheid, "An application-specific VLIW processor with vector instruction set for CNN acceleration," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2019, pp. 1–5.
[35] R. B. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, Aug. 1996.
[36] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, Feb. 2014, pp. 10–14.
[37] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016. [Online]. Available: arXiv:1603.01025.
[38] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advanced Neural Information Processing Systems 28*. Red Hook, NY, USA: Curran Assoc., Inc., 2015, pp. 3123–3131.
[39] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 8604–8612.
[40] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for DNN compression research," Oct. 2019. [Online]. Available: arXiv:1910.12232.
[41] B. Wu and M. Willems, "Rapid architectural exploration in designing application-specific processors," White Paper, Synopsy, Mountain View, CA, USA, 2015.
[42] ARM. (2019). *AMBA AXI and ACE Protocol Specification*. [Online]. Available: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf
[43] Synopsys. *Configurable SIMD Multiplier*. Accessed: Jul. 11, 2020. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?c=DWF_dp_simd_mult

ANDREAS BYTYN received the B.Sc. and M.Sc. degrees from RWTH Aachen University in 2011 and 2014, respectively, where he is currently pursuing the Ph.D degree as a Full-Time Research Assistant with the Institute for Communication Technologies and Embedded Systems. His current research interests are the exploration of programmable hardware architectures for deep learning and the optimization of their overall energy efficiency as well as the investigation of the flexibility-efficiency trade-offs involved in these systems.

RAINER LEUPERS received the M.Sc. and Ph.D. (Hons.) degrees in computer science from the University of Dortmund, Dortmund, Germany, in 1992 and 1997, respectively, where he was the Chief Engineer with the Embedded Systems Chair, from 1997 to 2001. He was a Team Leader with Integrated Community Development, where he was the Head of Industrial Service Projects, from 1999 to 2001. He joined RWTH Aachen University, Germany, in 2002, as a Professor of Software for Systems on Silicon. He has been the Co-Founder of LISATek, Inc., Menlo Park, CA, USA, and as an Electronic Design Automation Tool Provider for embedded processor design, acquired by CoWare, Inc., San Jose, CA. He has served as a Consultant for various companies, as an Expert for the European Commission in FP7, and in the management boards of large scale projects like UMIC, HIPEAC, ARTIST, SHAPES, and EURETILE. His current research interests include software development tools, processor architectures, and electronic design automation for embedded systems, with an emphasis on compilers, application-specific instruction-set processors, and multiprocessor system-on-chip design tools.

GERD ASCHEID (Senior Member, IEEE) received the Diploma Dr.-Ing. and Ph.D. degrees in electrical engineering with a specialization in communication engineering from RWTH Aachen University, Aachen, Germany. He was the Co-Founder and the Managing Director of CADIS GmbH, Aachen, in 1988, which successfully brought the system simulation tool COSSAP to the market. In 1994, CADIS was acquired by Synopsys, Inc., Mountain View, CA, USA, a California-based electronic design automation market leader, where he was the Director/Senior Director from 1994 to 2003. He joined the Institute for Communication Technologies and Embedded Systems, RWTH Aachen University in 2003, as a Full Professor. He is the Founder of several successful start-up companies. He has coauthored three books, authored numerous papers in the domain of digital communication algorithms and application-specific integrated circuit implementation. His current research interests include wireless communication algorithms and application specific integrated platforms, in particular, mobile terminals and cyber physical devices.