NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps

Alessandro Aimar*, Hesham Mostafa*, Enrico Calabrese*, Antonio Rios-Navarro†, Ricardo Tapiador-Morales†,
Iulia-Alexandra Lungu*, Moritz B. Milde*, Federico Corradi‡, Alejandro Linares-Barranco†,
Shih-Chii Liu*, Tobi Delbruck*

*Institute of Neuroinformatics, University of Zurich and ETH Zurich, Switzerland
†Robotic and Tech. of Computers Lab. University of Seville, Spain
‡iniLabs GmbH, Zurich, Switzerland

***Abstract*—Convolutional neural networks (CNNs) have become the dominant neural network architecture for solving many state-of-the-art (SOA) visual processing tasks. Even though Graphical Processing Units (GPUs) are most often used in training and deploying CNNs, their power consumption becomes a problem for real time mobile applications. We propose a flexible and efficient CNN accelerator architecture which can support the implementation of SOA CNNs in low-power and low-latency application scenarios. This architecture exploits the sparsity of neuron activations in CNNs to accelerate the computation and reduce memory requirements. The flexible architecture allows full utilization of available computing resources across a wide range of convolutional network kernel sizes; and numbers of input and output feature maps. We implemented the proposed architecture on an FPGA platform and present results showing how our implementation reduces external memory transfers and compute time in five different CNNs ranging from small ones up to the widely known large VGG16 and VGG19 CNNs. We show how in RTL simulations in a 28nm process with a clock frequency of 500 MHz, the NullHop core is able to reach over 450 GOp/s and efficiency of 368%, maintaining over 98% utilization of the MAC units and achieving a power efficiency of over 3 TOp/s/W in a core area of 5.8 mm^2.**

***Keywords*—Convolutional Neural Networks, VLSI, FPGA, computer vision, artificial intelligence**

I. INTRODUCTION

Extracting semantic information from raw real-world images is a long-standing problem that has been addressed by a succession of different approaches [1]–[4]. Deep neural networks in particular, convolutional neural networks (CNNs) have emerged as one of the most popular approaches for solving a variety of large-scale machine vision tasks [5]–[8]. The conceptual simplicity of CNNs coupled with powerful supervised training techniques based on gradient descent have made CNNs the method of choice for extracting high-level semantic image features that form the basis for solving visual processing tasks such as classification, localization, and detection [9].

CNN are trained, typically using a backpropagation algorithm and stochastic gradient descent, to produce the correct output for a set of labeled examples. The learning phase is typically done on graphical processing units (GPUs) using many manual iterations to determine the optimal architecture. In this paper we describe in Sec. II-B a custom toolchain to train CNNs with fixed-point precision to run on the NullHop hardware.

In run-time inference, the trained network is used to process images in order to extract various semantic information. In a power-constrained setting, off-line training of the CNN on workstations or servers followed by the deployment of the trained CNN on a mobile device is common practice. The inference phase in state-of-the-art (SOA) CNNs is computationally demanding, typically requiring several billion multiply-accumulate (MAC) operations per image. Using a mobile processor or a mobile GPU to run the inference step in a CNN can become unfeasibly expensive in a power-constrained mobile platform. For example, the recent Nvidia Tegra X1 GPU platform, which targets mobile automatic driver assistance (ADAS) applications, has recently been shown to be able to process 640x360 color input frames at a rate of 15Hz through a computationally efficient semantic segmentation CNN [10]. In this example, each frame requires about 2 billion multiply-accumulate (MAC) operations, thus making the GPU compute about 60 billion operations per second (GOp/s), for a power consumption of about 10W. Therefore at the application level, this GPU achieves a power efficiency of about 6 GOp/W, which is only about 6% of its theoretical maximum performance; as a result, the Nvidia solution can process a single CNN at 30 frames per second (FPS) only if the network requires less than 2 GOp/frame.

To overcome these problems, dedicated digital IC CNN accelerator architectures have been proposed [11]–[17]. However, despite the fact that these accelerators achieve impressive multiply and accumulate operations per second per Watt (MAC/s/W) figures, many of these operations are redundant. This is because the Rectified Linear Unit (ReLU) activation function used after most convolutional layers in SOA CNNs results in feature maps that are sparse, i.e, have a high percentage of zeros. MAC operations involving these zeros do not influence the final convolution result. Therefore several architectures have been proposed to exploit this sparsity. [14], [16], [17]. Because CNNs employ a wide variety of layer and kernel sizes, it is important that the accelerator is able to achieve low power consumption, high utilization of its computing resources, and low redundancy in its external memory access patterns, across a wide range of kernel sizes and number of input and output feature maps in each layer.

Here we propose a CNN accelerator architecture that addresses all these requirements. The three main features of the proposed NullHop accelerator are: 1) its ability to skip over zeros in the input CNN layers. No clock cycles are wasted and no redundant MAC operations are performed on zero entries. The time to output is directly proportional to the number of non-zero entries in the input layer. 2) Reduced external memory access due the use of a novel compression scheme optimized for sparse CNN layers that is more efficient than currently used run-length encoding schemes [14] and is suited for operating directly on compressed representations, unlike [16]. The processing pipeline operates directly on the compressed input representation. Since the compressed representation is not expanded in the accelerator, more input data can be stored in the accelerator memory. In many cases, this eliminates the need to load the same data multiple times, further reducing external memory access. 3) A configurable processing pipeline that maintains high efficiency across a range of CNN kernel sizes; and the number of input and output feature maps.

II. CNN PRINCIPLES OF OPERATION

CNNs extract high-level features from input images using successive stages of convolutions, non-linearities, and subsampling operations. These three stages are shown in Fig. 1. The convolution stage takes a 3D array $\mathbf{F_{in}}$ of dimension $N_{in} \times H \times W$ as input, which represents a set of N_{in} feature maps of dimensions $H \times W$. A typical vision application starts with 3 input feature map channels, corresponding to the red, green, and blue color channels of the camera image. Each feature map represents the distribution of a particular feature over the image space. These feature maps are convolved with the kernel array $\mathbf{K}$ to produce N_{out} output feature maps. $\mathbf{K}$ is a 4D array of dimensions $N_{out} \times N_{in} \times k_w \times k_h$, where k_w and k_h are respectively the kernel width and height. The result of the convolution is the 3D array $\mathbf{F_{out}}$ of dimension $N_{out} \times (H - k_h + 1) \times (W - k_w + 1)$. Considering z_{in} and z_{out} respectively as the input and output feature map index, and (y_{out}, x_{out}) the pixel position in the feature map, the pixel at position $(z_{out}, y_{out}, x_{out})$ in $\mathbf{F_{out}}$ is then given by:

$$\mathbf{F_{out}}(z_{out}, y_{out}, x_{out}) = \sum_{z_{in}=0}^{(Nin-1)} \sum_{i=0}^{(k_w-1)} \sum_{j=0}^{(k_h-1)} \mathbf{F_{in}}(z_{in}, y_{out}+i, x_{out}+j) \times \mathbf{K}(z_{out}, z_{in}, i, j) \tag{1}$$

A point-wise non-linearity is then applied to the output feature map array $\mathbf{F_{out}}$. In current SOA CNNs, the ReLU [18] is the most widely used non-linearity. ReLU is computed by $f(x) = max(0, x)$. In addition to being computationally cheap, using ReLUs empirically often yields better performance compared to saturating non-linearities, i.e, sigmoidal non-linearities [19]. The point-wise non-linear transformation is usually followed by a sub-sampling operation. A common sub-sampling strategy with many empirical advantages is max-pooling [20] where each pooling window is replaced by the maximum value in the window. Fig. 1 shows an example of a 2x2 non-overlapping pooling stage.

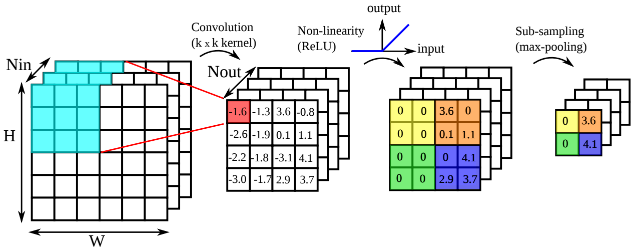


Fig. 1: The three main processing stages in a CNN.

A. Sparsity of Feature Maps

Applying a ReLU non-linearity to the output of a convolutional stage sets all negative-valued output pixels to zero. The resulting feature maps are therefore often sparse. These sparse feature maps form the input to the next convolutional stage. We quantitatively assessed the sparsity of the feature maps in three pre-trained SOA CNNs on the ImageNet data set: GoogLeNet [7], a 50-layer residual net [6], and VGG19 [21]. We fed 1000 random images from the ImageNet validation set to each of the three networks and calculated the average sparsity, i.e. percentage of zeros, of the feature maps produced in the different layers. The results after applying the ReLU non-linearity are outlined in Fig. 2. If a ReLU stage is followed by a pooling stage, we show the sparsity of the post-pooling feature maps. The shown sparsity figures thus represent the sparsity of the feature maps that form the inputs to subsequent convolutional layers. The sparsity ranges from about 20% to almost 90% even after max-pooling.

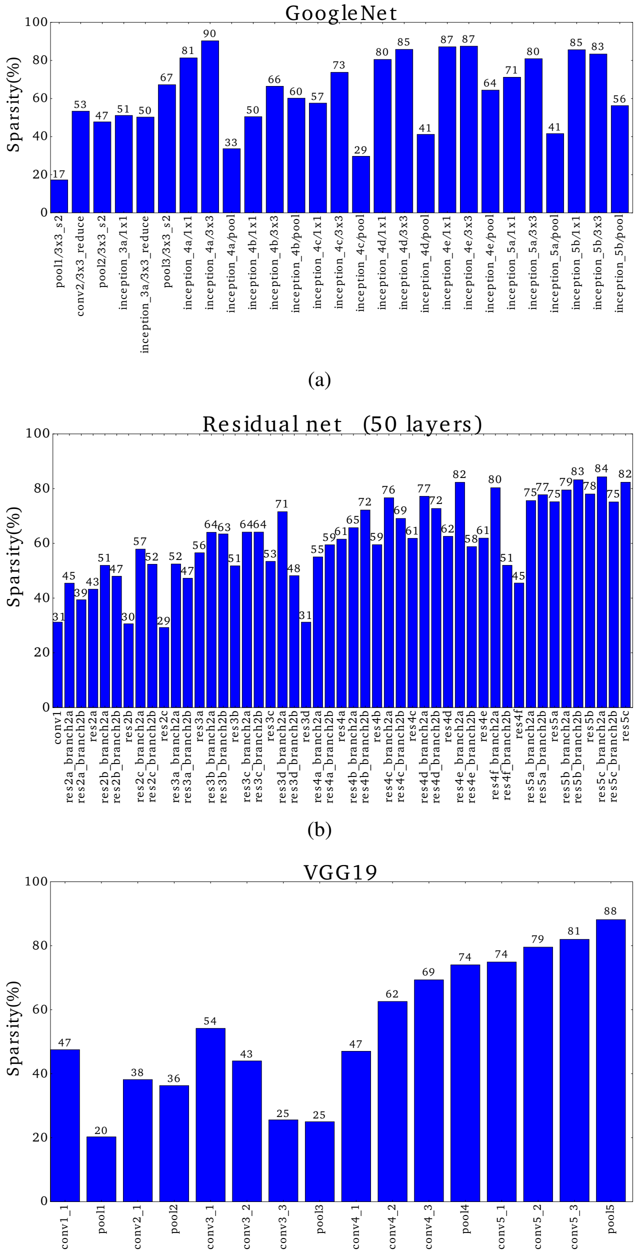
B. Reduced Numerical Precision CNNs

It is possible to operate deep networks with reduced bit precision as well as with fixed-point number representation as we and others demonstrated in [22]–[25]. Scaling down the number of bits allows a reduction in the amount of data transfer from memory as well as a smaller design area for the digital circuits. The current SOA reduced precision deep learning library is based on [26] and is called Ristretto [27]. This tool can be used to train and test CNNs with reduced numerical precision of weights and activations. When directly compared to top-1 accuracy of deep neural networks (DNNs), such as VGG16 [21], Ristretto is able to recover close to the full precision classification accuracy of 68.3%, however Ristretto only quantizes the weights to a fixed-point notation, whereas the activations are quantized using 16 bit floating-point notation. Activations are actually much harder to represent using fixed-point, since the dynamic range of activations span nine orders of magnitude, for example in the case of VGG16.

In order to run reduced precision CNNs on our accelerator, we developed a custom branch of Caffe [1]. It can be used to train networks from scratch as well as to fine-tune existing 32-bit floating-point networks to any user specified fixed point precision for weights and activations. It also includes tools for estimating the required per-layer decimal point locations. We achieved VGG16 [21] 67.5% top-1 accuracy after quantizing the weights and activations to 16 bits, with an accuracy drop of only 0.8% compared with the floating point VGG16. Reducing

---

[1] git@github.com:NeuromorphicProcessorProject/ADaPTION.git



(a)

(b)

(c)

Fig. 2: Sparsity of the feature maps produced by different layers in: (a) GoogLeNet, (b) 50-layer residual net, and (c) VGG19. Statistics were obtained after the ReLU operation and after pooling (if the convolutional layer was followed by a pooling stage).

the number of bits also results in up to 50% increased sparsity per layer of activations as shown in Fig 3. The average sparsity in the activations increased from 57% in floating precision to 82% in reduced precision.

III. ACCELERATOR ARCHITECTURE

Figure 4 shows the high-level schematic of the NullHop accelerator. The accelerator implements one convolutional stage followed by a ReLU point-wise non-linearity followed by a max-pooling stage. The ReLU and max-pooling stages can be disabled. To implement the full forward pass in a CNN, the accelerator evaluates the successive convolutional stages one after the other in a cascaded manner. The input feature maps and the kernel values for the current convolutional layer are stored in two independent SRAM blocks. The internal memory structures are described in Sec. III-B2 and Sec. III-C1.

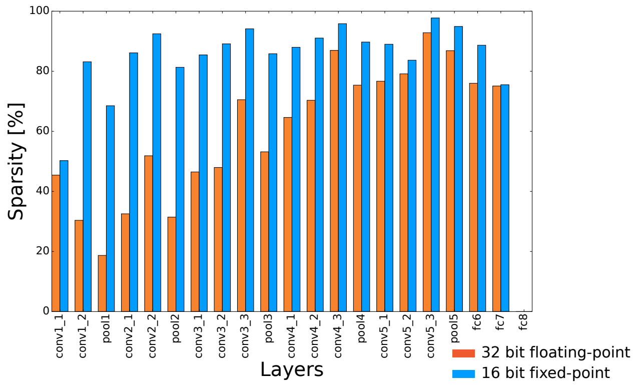


Fig. 3: Sparsity comparison before (orange) and after (blue) quantizing activations to 16 bit fixed-point of VGG16.

The output feature maps produced by the current layer are streamed off-chip to the external memory. They are then streamed back to the accelerator SRAM when the accelerator has finished processing the current layer. The feature maps are always stored in a compressed format that is never decompressed but rather decoded during the computation.

The following subsections describe the different functional blocks of the accelerator, the compression scheme employed, and the processing pipeline.

We first give a high-level overview of the processing pipeline. The input decoding block reads a small portion of the compressed input feature maps to generate pixels that are passed to the pixel allocator block. Only a few pixels, typically in one mini-column (of height equal to $k_h + 1$) are fully decoded at any one time. These pixels are *all non-zero*. The input decoding block is able to directly skip over zero pixels in the compressed input feature maps without wasting any MAC operation. In addition to the pixel values, the input decoding block also provides the pixels' positions (row, column, and input feature map index) to the pixel allocator block. The pixel allocator block allocates each incoming pixel to a MAC unit controller. Each controller manages the operation of a subset of the MAC blocks and submits the appropriate read requests to the kernel memory banks. All MAC blocks under the same controller receive the same input pixel from their controller, but weights from different kernels, producing pixels in different output feature maps. The convolution results are sent through an optional ReLU transformation and a max-pooling stage before going to the pixel stream compression block. The compressed output feature maps are then sent off-chip.

A. Sparse Matrix Compression Scheme

The NullHop accelerator uses a novel sparse matrix compression algorithm. It is based on two different elements: a Sparsity Map (SM) and a Non-Zero Value List (NZVL). The SM is a binary, 3D mask, having the same number of entries as number of pixels in the feature maps. Each binary entry

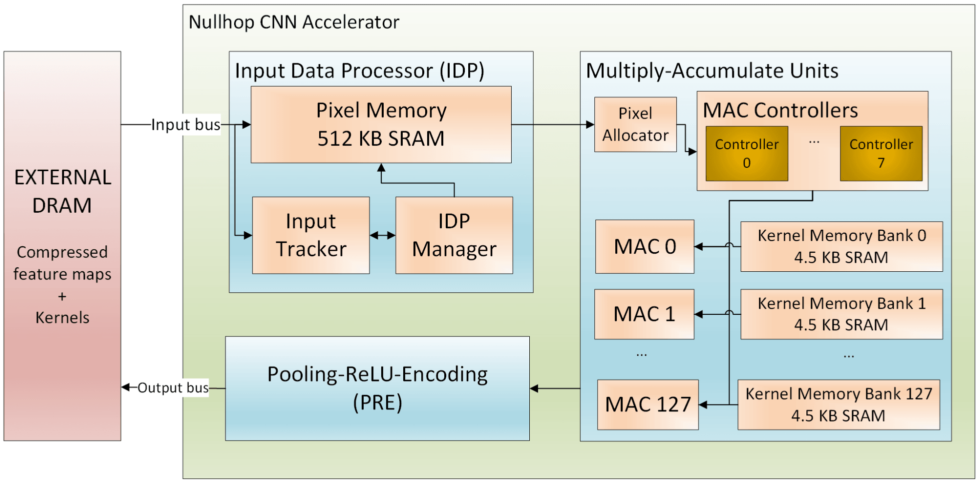

Nullhop CNN Accelerator
EXTERNAL DRAM
Compressed feature maps + Kernels
Input bus
Output bus
Input Data Processor (IDP)
Pixel Memory 512 KB SRAM
Input Tracker
IDP Manager
Multiply-Accumulate Units
Pixel Allocator
MAC Controllers
Controller 0
...
Controller 7
MAC 0
MAC 1
...
MAC 127
Kernel Memory Bank 0 4.5 KB SRAM
Kernel Memory Bank 1 4.5 KB SRAM
...
Kernel Memory Bank 127 4.5 KB SRAM
Pooling-ReLU-Encoding (PRE)

Fig. 4: High-level schematic of the proposed NullHop CNN accelerator.

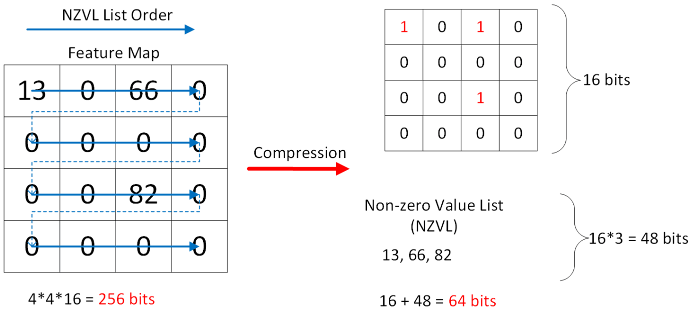


Fig. 5: Sparse compression scheme acting on a single sparse feature map. The non-zero values are stored as an ordered non-zero value list. The order goes row-wise from top to bottom and from left to right in each row as shown on the feature map. The sparsity map (SM) is a binary mask with 1s at the positions of non-zero pixels, and 0s at the positions of zero pixels.

in the SM is 1 if the corresponding pixel is non-zero, and 0 otherwise:

$$SM(z,y,x) = \begin{cases} 1, input(z,y,x) \neq 0 \\ 0, \text{otherwise} \end{cases} \tag{2}$$

The SM is used to reconstruct the positions of the non-zero pixels. These non-zero pixel values are stored as the NZVL: an ordered, variable-length list containing all the non-zero values in the feature maps. The compression scheme is illustrated in Fig. 5. The accelerator sequentially reads both the SM and the NZVL and uses the information from the SM to decode the positions of the pixels in the NZVL in a sequential manner as described in the next subsection. The compressed image size in bits is given by (3):

$$CS = E(1 + N(1 - S)) \tag{3}$$

where

CS	input image size in bits
E	total number of entries (including zeros) in input matrix
S	sparsity of input image (range 0-1)
N	input bit precision

TABLE I: Minimum sparsity needed for compression of input feature maps.

Bit Precision	Threshold Sparsity
8	12.5%
12	8.33%
16	6.25%
24	4.16%
32	3.12%

Eq. (3) shows how the compressed size of the input image has a lower limit $CS = E$ when sparsity $S = 1$. From Eq. (3), it is also possible to demonstrate that the algorithm provides a reduction in memory when condition (4) is satisfied:

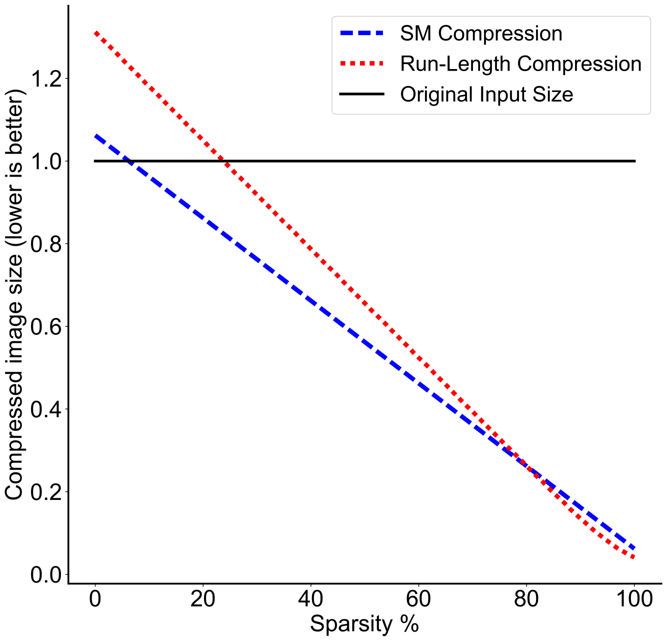
$$S > \frac{1}{N} \tag{4}$$

Solving Eq. 4 for different data bit precisions leads to Table I where a precision of 16 bits shows a threshold sparsity of 6.25% which is low enough to guarantee compression in most CNN layers. The algorithm provides better performance than run-length compression proposed by [14] for almost all compression levels as shown in Fig. 6. At the same time our sparsity map compression is equivalent in terms of level of compression to the Huffman coding used by [16] but the data structure permits an easier decoding during the computation, allowing the accelerator to operate directly on compressed representations without decompression.

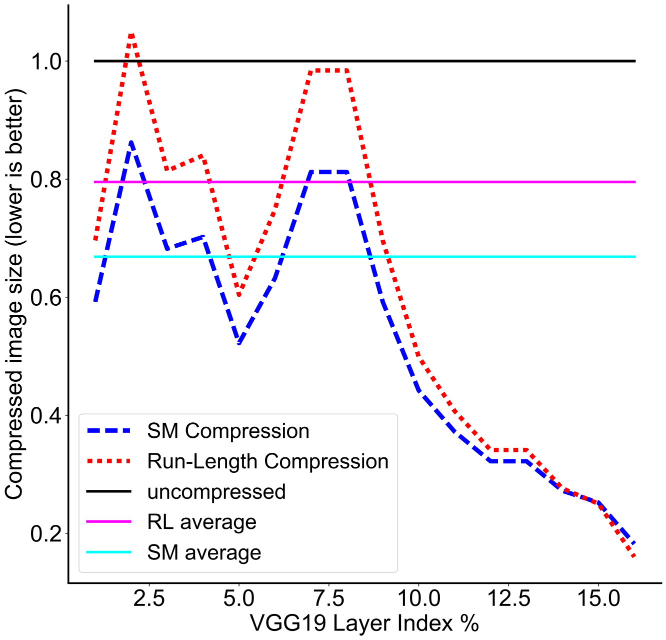
B. *Pixel Memory and Decoding: Input Data Processing Unit*

1) Input format: To reduce the idle loading time, the 3D SM is streamed to the CNN accelerator in fragments that are interleaved with NZVL pixels. The length of SM fragments is implementation dependent and is equivalent to the pixels bit precision. In this way, the input bus to the CNN accelerator can contain two values of pixels or SM segments, interleaved as shown in Fig. 7.

The number of ones in an SM segment indicates the number of pixel values that will follow the SM segment as shown in Example 1 of Fig. 7. The position of the ones indicate the offsets of these pixels. The first field in the first word that



(a) Comparison of compression methods over 10,000 images with different amounts of sparsity.



(b) Comparison on 1000 runs of VGG19.

Fig. 6: Compression performance comparison between the Sparsity Map algorithm and Run Length Compression.

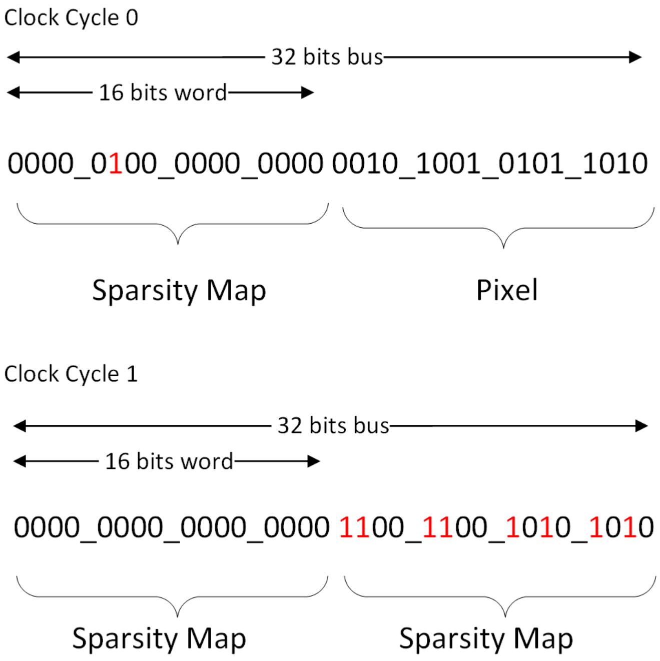


Fig. 7: Format of the words sent to the accelerator, using 16-bit words and 32-bit bus as an example. Sparsity map segments covering 16 positions are interleaved with non-zero pixel values.

is sent to the accelerator is always a SM segment. All fields afterwards can be SM segments or pixel values. If there is a run of 16 zero pixels, an SM segment can be all zeros as shown in Example 2 of Fig. 7. The SM segment will then be followed by another SM segment and this will continue until an SM segment that is not all zeros is streamed in. The compressed rows are streamed into the accelerator one after the other starting from the top row: let $p(z,x,y)$ be the pixel at position (x,y) belonging to the z^{th} input feature map. The pixels are streamed into the accelerator as:

$$p(0,0,0), p(1,0,0), .., p(N_z,0,0), p(0,1,0), .., p(N_z,1,0), ..$$

2) *Decoding the compressed rows:* The Input Data Processor (IDP) is responsible for storing and decoding the compressed rows of the input feature maps. The IDP contains multiple SRAM banks and can start decoding the data from these banks while the input feature maps are still being loaded. The IDP maintains a pointer to the beginning of each row stored inside the SRAM and uses these row starting addresses to sequentially decode the pixels in each row. The module is thus able to perform a random access to the first entry of each row, while entries in a row can only be accessed in a sequential manner. At each clock cycle, the IDP can read up to $k+1$ non-zero pixels from memory to be sent to the MACS module, generating pixels in a winding manner within a horizontal stripe as shown in Fig. 8. The pixels within this stripe are the only pixels needed to generate a double row in the output feature maps, assuming a vertical convolution stride of 1. The IDP supports zero padding at feature map boundaries without having to load any extra data and without wasting any clock cycles. This is possible by adding proper offsets to each pixel position while sending them to MACS module.

There are three main blocks inside the IDP unit:

a) Pixel Memory: The Pixel Memory module stores the input feature maps to be processed and is composed of several SRAM memory banks and arbiters. The arbitration scheme gives maximum priority to write requests from off-chip, followed by read requests received by the IDP Manager (described in Sec. III-B2c). The module also handles the communication between the IDP and MACS modules, sending pixel values to the processing unit and stalling IDP operations if MACS cannot receive more data in their internal FIFOs.

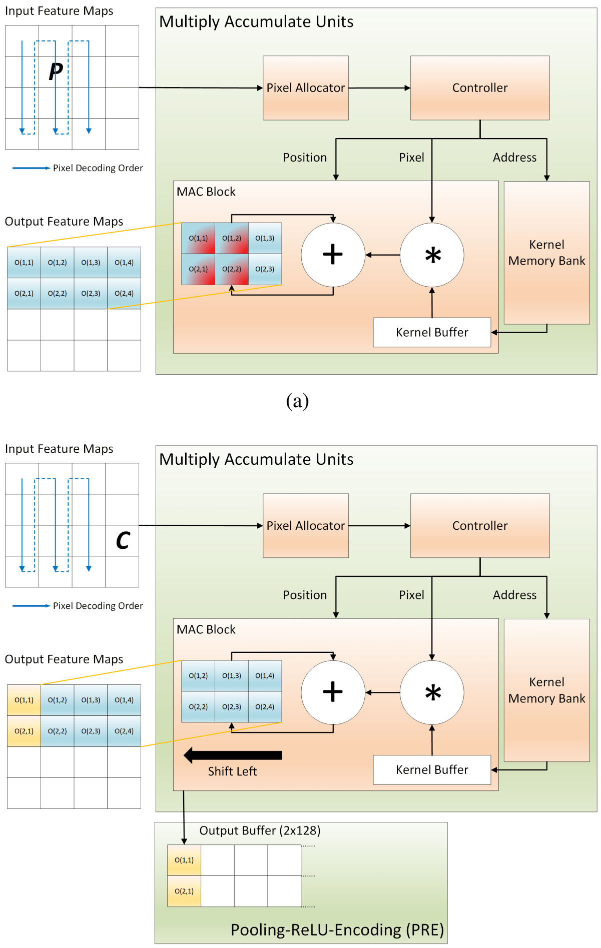
b) Input Tracker: In order to access rows in compressed mode, a memory that stores the pointers to each row's starting position in the Pixel Memory is required. The Input Tracker accomplishes this task by storing addresses received by the Pixel Memory in a small SRAM memory when a new row is stored; and providing them to the IDP Manager when requested. The read/write arbitration in the Input Tracker follows the same scheme as the one in the Pixel Memory.

c) IDP Manager: IDP operations are controlled by the IDP Manager, a set of FSMs acting as control units for the module. The number of FSMs is equal to the maximum kernel size supported plus 1; each control unit is in charge of sending read requests to Pixel Memory for a specific row in the currently active stripe. During processing, $k+1$ FSMs are turned on, while others are disabled. Each FSM stores in its internal registers an SM and a memory pointer to the next address to be read. The IDP FSMs also share a register containing spatial coordinates of the next pixel to be read and use it to compute the addresses of pixels read.

At the beginning of the layer, the IDP Manager reads the pointers to the first $k+1$ rows from the Input Tracker and stores them into each IDP FSM memory pointer register. FSMs then send a read request to the Pixel Memory and store the read result into their SM registers. In the same clock cycle, the position of the first non-zero entry is computed using the SM itself and the memory pointer register is increased by one to be ready for then next read operation. Next, clock cycles are dedicated to the actual reading of pixels from memory: FSMs first send a read request to the Pixel Memory for the current memory pointer (providing its spatial coordinates as extra information to be forwarded to MACS), then incrementing the memory pointer by 1, and looking up the next non-zero entry in the SM in order to get the coordinates of the pixel to be read in the next clock cycle. The IDP iterates over SMs and pixels until the row ends, moving to the next feature map stripe when all FSMs have completed their task. No zero pixel is forwarded to the MAC blocks.

C. The Compute Core: Pixel Allocator, Controllers, and MAC Blocks

The main processing element in the compute core is the MAC block. Each MAC block generates pixels or partial pixels in one output feature map. At any point in time, the MAC block internally maintains the partial pixel values for a $2 \times k$ region in its target output feature map. This memory is needed to enable 2×2 pooling on the fly as we will describe shortly. Each MAC accumulates in full precision, without truncating or rounding until the computation of the convolution is completed. This differs from [14], where multiplication's results are truncated before accumulating them. The compute core can be configured in multiple ways. We start by describing the simplest configuration in which only one controller is active and this controller manages all the MAC blocks. In this configuration, each MAC block receives the same pixel from the controller and uses this pixel to update the currently active region in its output feature maps; thus in this example 128 output features maps are processed in parallel. In this configuration, the kernel memory bank associated with each MAC block stores the kernel values corresponding to one output feature map. In the notation of Eq. 1, the kernel bank associated with MAC block r stores the kernel values $\mathbf{K}(r,:,:,:)$. In this configuration, each MAC block produces pixels in a different output feature map.



(a)

(b)

Fig. 8: Pixel processing scheme for the example case of a 3×3 kernel, one input feature map, and 128 output feature maps. (a) One MAC block is shown, the pixel P is used to update the red-shaded pixels in the current output patch. (b) On receiving pixel C, the controller detects (based on the column index of C) that the left-most pixels in the current output patches, i.e. pixels $o(1,1)$ and $o(2,1)$, are complete and issues a command to shift all the output patches in the MAC blocks by one position to the left. Pixels $o(1,1)$ and $o(2,1)$ are shifted into an output buffer.

Figure 8 illustrates the operation of a single MAC block when the kernel's dimensions are 3×3. Assume for simplicity there is only one input feature map. The IDP sends the pixel labeled P together with its position to the pixel allocator block which allocates this pixel to the only active controller. Based on the position of pixel P, the controller sends a read request to the kernel memory banks to read the kernel values that should be multiplied by pixel P in order to update the output patches in the MAC blocks. All kernel banks receive the same address as the kernels are laid out identically in each of them. The kernel banks thus produce kernel values that differ only in their output feature map index.

Initially, the 2×3 patch maintained by each MAC block is the upper left patch in the output feature map. The multiplier and adder in each MAC block use the received pixel and kernel values to update the active patch. Since P is a border pixel (it is less than $s-1=2$ positions from the left edge of the input feature map), it is only used to update the left-most 4 pixels in the MACs. A non-border pixel would be used to update all $2 \times k = 6$ pixels in each MAC.

The IDP decodes pixels in a winding manner within a horizontal stripe of height $k+1=4$ in the input feature map as shown in Fig. 8. The pixels within this stripe are the only pixels needed to generate a double row in the output feature maps, assuming a convolution stride of 1. When input decoding moves beyond the first $k=3$ columns in the horizontal stripe, this indicates that the left-most pixels in the MAC output patches are complete. As shown in Fig. 8b, when pixel C is dispatched to the controller, the controller issues a command to all MAC blocks to shift their internal patch one position to the left. The internal patches in the MAC blocks now represent a new region in the output feature maps as shown in Fig. 8b. The completed left-most pixels that have been shifted out go into the pooling/ReLU/encoding (PRE) buffer which is discussed in the next subsection. This process continues: as input decoding moves to a new column, the patches are shifted one position to the left until all the pixels in a double row in the 128 output feature maps are produced.

1) Handling multiple feature maps: In nearly all CNNs, there are a multitude of input and output feature maps, ranging from a minimum of 1 or 3 (monochrome or color images) to 512 maps in the well-known examples from Fig. 2. In the case of multiple input feature maps, the decoding pattern first moves along the z-dimension (feature map dimension) to exhaust all feature map values at a particular x-y position before moving to a new x-y position. The processing scheme is otherwise unchanged.

This simplest accelerator configuration can produce N output feature maps in one pass through the input data. In cases where the convolutional layer has less than N output feature maps, the accelerator is able to assign multiple MAC blocks to the same output feature map to prevent some MAC blocks from idling. We also assign multiple MAC blocks to the same output feature map if the kernel values for one output feature map cannot fit into one kernel memory bank, i.e. the storage requirements for N output feature maps kernels exceed the available kernel memory. We designed the compute core so that 1, 2, 4, or 8 MAC blocks can be clustered together and used to produce one output feature map. Each MAC block in a cluster can receive kernel values from any kernel memory bank in the cluster. MAC blocks in a cluster are managed by different controllers, with each single controller managing one MAC block in each cluster. The pixel allocator block monitors the active controllers and dispatches a new pixel to whichever controller becomes idle. Each controller in turn dispatches the received pixel to the MAC blocks under its control. When there are 8 MAC blocks per cluster, all 8 controllers are active and the core is processing 8 input pixels in parallel.

The controllers issue read requests to the kernel memory banks to read the relevant kernel values. It might happen that two controllers need to access the same bank at the same time. A fair round-robin arbitration scheme serializes the clashing requests. Typically, this does not lead to idle cycles in the MAC blocks because the controllers try to pre-fetch kernel values and store them in the kernel buffers in the MACs several cycles before they are needed. Several cycles of memory contention can thus be absorbed while keeping the MAC blocks occupied. At any time, each cluster is processing the same set of input pixels as every other cluster. The clusters proceed in a lock-step manner where the patterns of memory contention are the same in each cluster. The clusters only differ in the contents of their kernel banks which store kernel values belonging to different output feature maps.

D. Pooling, ReLU, and Encoding Unit

The pooling, ReLU, and encoding (**PRE**) unit is responsible for the last processing steps in the computational pipeline. It receives the pixel pairs or partial pixel pairs from the MAC blocks and stores them in the PRE buffer. In the current implementation, its size is $2 \times N$ memory locations where 2 is the pooling dimension and N is the number of MAC blocks. The ReLU non-linearity can be applied and 2×2 pooling can be performed on the fly. An encoder is used to compress the output pixels according to the scheme described in Section III-A; and to stream out the SM segments and non-zero pixels.

If the number of MAC blocks per cluster is larger than 1, the MAC blocks output partial pixels. Before data can be further processed, the partial pixels need to be summed to get the final pixel values. The accumulation is performed within the PRE buffer, in a number of clock cycles that can vary from 1 (2 MACs per cluster) to 3 (8 MACs per cluster); during each accumulation cycle, two adjacent partial pixels are summed together and the result is stored back in the PRE buffer. Once the accumulation is over, the pixels are transferred to the output buffer, that can store up to N pixels.

The ReLU non-linearity is applied during the transfer of pixels from the PRE buffer to the output buffer. Each entry of the output buffer is initialized to zero if the ReLU non-linearity is enabled, otherwise with the most negative number that can be represented. When transferring pixels from the PRE buffer to the output buffer, a max operation is performed so that the value stored in the output buffer is the maximum of the incoming pixel and the original stored value.

When pooling is enabled, the maximum of the two values of each pixel pair and the current content of the output buffer is stored back into the output buffer. When the second pixel pair arrives from the MAC blocks, the same max operation is carried out. In this way the output buffer contains the results of the 2×2 max-pooling. When pooling is disabled, one row at a time is transferred from the PRE buffer to the output buffer. The encoder acts on the first K pixels of the output buffer, a number of pixels equal to the pixel bit width. In our 16 bit implementation, the encoder works on 16 pixels at a time. The encoder generates a SM segment from the first K pixels based on the position of the non-zero pixels. For each set of K pixels, during the first clock cycle the encoder streams out the SM and the first non-zero pixel; then, in each clock cycle two pixels can be sent out. When all the pixels of the current SM have been streamed out, the output buffer is shifted by K positions, a new SM is generated and the non-zero pixels streamed out. This is repeated until all the pixels in the output buffer are processed. The encoding scheme can be switched off: in this case all pixels are streamed out and no SM is generated. This option allows the host software to obtain the output activation in a easily accessible format, speeding up any eventual extra processing the CNN may require.

IV. DESIGN IMPLEMENTATION

The NullHop architecture was implemented using the following parameters:

- 16-bit precision fixed point kernels/activations
- 32-bit precision MAC units
- 32-bit input/output bus
- IDP memory: 512 KB
- Kernel memory: 576 KB
- Number of MAC: 128
- Max supported kernel size: 7
- Max number of rows input image[2]: 512
- Max number of columns input image[2]: 512
- Max number of feature maps[2]: 1024

A. Synthesis, Place and Route

The design has been synthesized using Globalfoundries 28nm technology with 1 V supply voltage. Fig. 9 shows the result of place and route. The post place and route core size is 5.8 mm^2 (core utilization = 75%) and the chip size including I/O pads is 7.5 mm^2 , with an estimated power consumption of 155 mW at the clock frequency of 500 MHz. The power consumption was estimated using a switching activity based technique (SAIF) with Synopsys Design Compiler. The result allows us to estimate the compute performance per Watt of the accelerator.

B. FPGA Implementation

To validate the design, we have implemented it on a Xilinx Zynq 7100 System-On-Chip (SoC) FPGA, using the AXI4-Stream with Direct Memory Access (DMA) open source protocol to connect the SoC's FPGA with its ARM processor. The ARM CPU, running Petalinux as operating system (OS), is used as the controller for the accelerator. It manages the read and write operations between the DDR memory of the ARM computer to the BRAM of the accelerator. The processor also computes fully connected (FC) layers typically placed as the final layer of many CNNs. We included in the OS an embedded USB host controller module to interface it to an iniLabs DAVIS240C neuromorphic event-based camera [28] for the real-time demonstrations described in Sec. V. In this case the ARM processor also runs iniLabs cAER[3], an open source software to interface to DAVIS camera. Fig 10 shows the block diagram of our FPGA architecture, including modules MM2S and S2MM used to interface the accelerator with the AXI4-S bus.

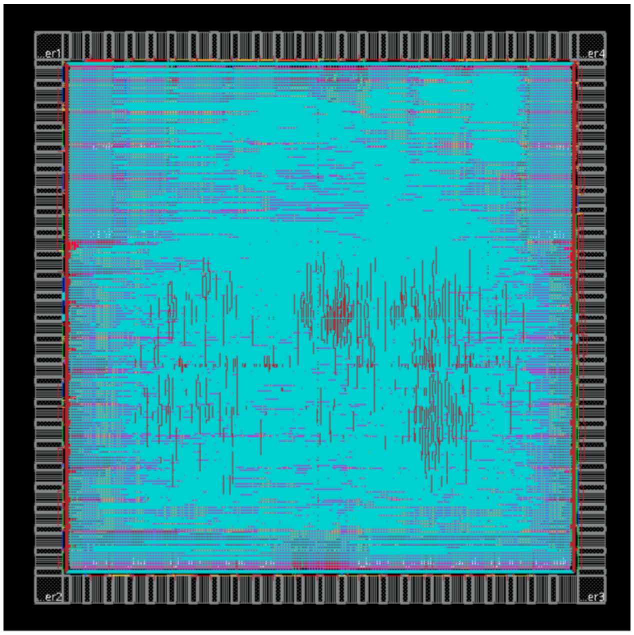


Fig. 9: Nullhop chip place and route

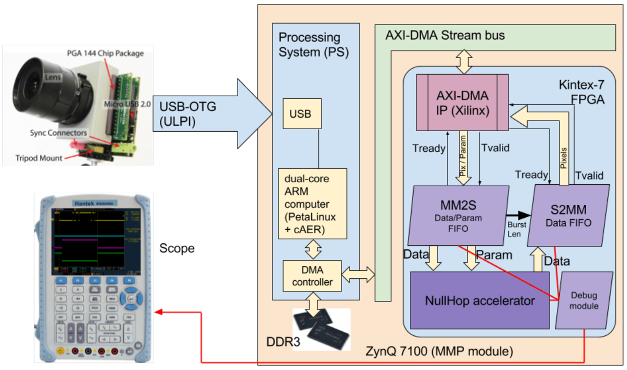


Fig. 10: System on Chip block diagram for testing scenario. The FPGA on the Zynq SoC hosts the NullHop accelerator plus glue-logic to interface to the Zynq ARM processor DDR memory through DMA

The design minimizes host memory manipulation by using DMA transfers from host memory to the accelerator and vice versa without need to reformat or process each layer output. For each layer, the ARM loads the layer configuration and the kernels. It then initiates interrupt-driven DMA transfer of the input and output. It is then free for other processing while the layer is computed. The development of this DMA functionality required considerable effort past the basic reference IP from Xilinx.

Our IC implementation targets a clock frequency above 500 MHz, but routing typically limits the FPGA implementation to much lower frequencies. For the Zynq 7100, the maximum clock frequency is 60 MHz after synthesizing and implementing the entire infrastructure composed of NullHop, plus AXI interfaces. Table II shows the resources required for the implementation.

---

[2] Determined by counter resolution; chosen for tested networks. No effect on throughput or area.

[3] https://inilabs.com/support/software/caer/

TABLE II: Resources used on NullHop Zynq 7100.

Resources	Logic	FF	BRAM	DSP
Full System	229K (83%)	107k (19%)	386 (51.1%)	128 (6.3%)

Power analysis for implemented NullHop for the Zynq 7100 has been estimated with Vivado assuming 0.5 switching rate for the logic. These estimations offer a static power consumption of 316 mW when FPGA clock is stopped and the ARM cores are idle, and a dynamic power consumption of 1.5 W for the ARM processor and 0.8 W for NullHop plus the AXI4-S logic. Table III shows this estimation distributed among the NullHop infrastructure The ARM consumes 63% of the total power, while 37% is used by the logic circuits and memory in the Kintex-7 embedded FPGA, where it is distributed between components as 27.27% for MACs, 4.14% for IDP, 1.75% for PRE, and 1.15% for AXIstream

In order to validate these Xilinx estimations we have measured the power consumption of the complete testing infrastructure running the example described in Sec. V-C at different stages. The base-board that hosts and power the AvNet 7100 SoC mini-module consumes 5.1 W without the 7100 mini-module plugged. The base-board with the mini-module plugged and its fan connected, but with any design in the FPGA and in reset mode, requires 6.95 W. After programming the FPGA, in an idle linux state (not running cAER), the system consumes 8.27 W. And running the experiment in a continuous iterative way, the needed power increase to 9 W. This implies an effective power consumption of about 750 mW, which is close to Xilinx estimations (777 mW for the FPGA).

V. APPLICATION EXAMPLE

We studied application scenarios, ranging from small custom-trained CNNs with a small number of classifier outputs to large widely used CNNs used in the ImageNet classification task to label an image with one out of a thousand possible labels (Tables IV-VIII).

A. *VGG16 and VGG19 networks*

The VGG16 and VGG19 networks [21] are large CNN architectures VII VIII used for classifying the ImageNet dataset. They require 31 GOp and 39 GOp, respectively and can be trained to achieve 68.3% and 71.3% top-1 accuracy using floating precision weights and states. For evaluating the performance of the accelerator we used randomly chosen images from the dataset. Because of their size, these two networks require particular arrangements to be run on the accelerator:

TABLE III: Power consumption estimation of NullHop on Zynq 7100.

Power (mW)	Dynamic	FPGA	Logic	BRAM	DSP	Routing
NullHop + AXI4-s + ARM	2339	804	20	396	2	67
NullHop	777	777	19	384	2	63
IDP	97	97	2	75	-	13
MACs + kernel memory	638	638	13	309	2	43
PRE	41	41	4	-	-	7

TABLE IV: Face Detector: Parameters of the 2 convolutional layers

Layer Number	Input feature maps	Output feature maps	Kernel Size	Input Width/Height	Pooling	Number of passes
1	1	16	5	36x36	Yes	1
2	16	16	3	16x16	Yes	1

TABLE V: RoshamboNet: Parameters of the 5 convolutional layers

Layer Number	Input feature maps	Output feature maps	Kernel Size	Input Width/Height	Pooling	Number of passes
1	1	16	5	64x64	Yes	1
2	16	32	3	30x30	Yes	1
3	32	64	3	14x14	Yes	1
4	64	128	3	6x6	Yes	1
5	128	128	1	2x2	Yes	1

TABLE VI: Giga1Net: Parameters of the 11 convolutional layers

Layer Number	Input feature maps	Output feature maps	Kernel Size	Input Width/Height	Pooling	Number of passes
1	3	16	1	224x224	Yes	1
2	16	16	7	112x112	Yes	1
3	16	32	7	54x54	Yes	1
4	32	64	5	24x24	Yes	1
5	64	64	5	22x22	Yes	1
6	64	64	5	20x20	Yes	1
7	64	128	3	18x18	Yes	1
8	128	128	3	18x18	Yes	1
9	128	128	3	18x18	Yes	1
10	128	128	3	18x18	Yes	1
11	128	128	3	18x18	Yes	1

TABLE VII: VGG16: Parameters of the 13 convolutional layers

Layer Number	Input feature maps	Output feature maps	Kernel Size	Input Width/Height	Pooling	Number of passes
1	3	64	3	224x224	No	1
2	64	64	3	224x224	Yes	1
3	64	128	3	112x112	No	1
4	128	128	3	112x112	Yes	1
5	128	256	3	56x56	No	2
6-7	256	256	3	56x56	No	2
8	256	512	3	56x56	Yes	4
9	512	512	3	28x28	No	8
10	512	512	3	28x28	Yes	8
11	512	512	3	14x14	Yes	8
12	512	512	3	14x14	No	8
13	512	512	3	14x14	Yes	8

TABLE VIII: VGG19: Parameters of the 16 convolutional layers

Layer Number	Input feature maps	Output feature maps	Kernel Size	Input Width/Height	Pooling	Number of passes
1	3	64	3	224x224	No	1
2	64	64	3	224x224	Yes	1
3	64	128	3	112x112	No	1
4	128	128	3	112x112	Yes	1
5	128	256	3	56x56	No	2
6-7	256	256	3	56x56	No	2
8	256	256	3	56x56	Yes	2
9	256	512	3	28x28	No	4
10-11	512	512	3	28x28	No	8
12	512	512	3	28x28	Yes	8
13-15	512	512	3	14x14	No	8
16	512	512	3	14x14	Yes	8

Number of output feature maps Since there are 128 MAC blocks and at least one MAC block has to be dedicated to each output feature map, multiple passes through the input feature maps are needed in order to produce more than 128 output feature maps. This is the case for example for layers 5-8 of VGG19 in table VIII. The output feature maps are divided into multiple subsets with at most 128 maps per subset. Each subset is produced in one pass through the input feature maps. The kernels for the current subset of output feature maps are loaded into the accelerator at the beginning of each pass.

IDP memory size If the input feature maps cannot fit into the accelerator's SRAM and the output ones are more than 128, the input feature maps must be streamed in multiple times. . Multiple stream-in are less likely to occur in the zero-skipping mode since it is more likely that the compressed input feature maps can fit into the accelerator SRAM. The 512 KB of memory implemented in the IDP has been verified as sufficient to avoid multiple streaming for all tested input images.

Kernel memory size for one output feature map Each MAC block has access to 8kB of kernel memory, i.e, 4k kernel values. If there are more than 4k kernel values associated with one output feature map as in layers 10-16 of VGG19 in table VIII (where each output feature map has $512 \times 3 \times 3 = 4608$ kernel values, which is larger than 4096), then multiple MAC blocks and their kernel banks must be clustered together. Each cluster is responsible for producing one output feature map and must have enough kernel memory to store the kernels for one output feature map. For layers 10-16, the MAC blocks are clustered in groups of 2 to produce 64 output feature maps in each pass through the input feature maps.

B. Giga1Net

Giga1Net is a 1 GOp/frame (about 5 times more than [5]) CNN designed for stress testing our accelerator during development. It contains various computational scenarios, with both large (7x7) and small (1x1) kernels, layers with or without pooling and/or zero padding (Table VI). Rather than having high accuracy as the main goal, the purpose of the Giga1Net architecture is to identify inefficiencies in the hardware computational pipelines. Despite its relatively small size compared with the VGG CNNs, Giga1Net achieves 40% accuracy on the ImageNet dataset.

C. RoshamboNet

The RoshamboNet is a 5-layer, 20 MOp, 114k weight CNN architecture, described in Table V, trained to play the rock-scissors-paper game [29]. This network can classify input images of size 64x64 obtained from a DVS camera using the same training and feature extraction stage approach from [30]. The network outputs 4 classes: "rock", "scissors", "paper" or "background" from each feature vector. Each DVS frame is a 2D histogram of 2k DVS events. Thus the frame rate varies from less than 1Hz to over 400Hz depending on the hand speed.

As proof of concept, we connected to the Xilinx Zynq SoC, a robotic hand and a LED-based display, driven by a customized version of cAER running on the Zynq ARM processor. The system was presented at the NIPS 2016 Live Demonstration track [31]. It can play in real-time against a human opponent, recognizing the player symbol with over 99% accuracy and reacting in less than 10ms to create a convincing illusion of outguessing the opponent.

D. Face Detector CNN

The face detector is a very small CNN designed to recognize whether a face is present or absent in an image obtained from the DAVIS camera. The DVS events and APS frames are processed by the ARM cAER thread to generate 36x36 input images, again using the method of [30]. The CNN was trained on a dataset of 1800k frames collected from public face datasets and labeled DAVIS frames. The CNN architecture described in Table IV requires 1.98 MOp for classifying a single frame.

VI. RESULTS

The networks described in Sec. V have all been run on both the Mentor QuestaSim HDL simulator with post place-and-route delays and on our Xilinx Zynq platform. Results are summarized in Tables IX and X.

A. VGG19 and VGG16

When Nullhop processes big and deep networks with a high level of sparsity, it achieves more GOp/s than the ideal maximum (128 GOp/s, equal to the number of MAC units times their clock frequency) because of the skipping of operations, achieving an efficiency greater than 100%. VGG19 and VGG16, with respectively 471.64 GOp/s - 368.47% efficiency and 420.83 GOp/s - 328.8 % efficiency illustrate this effect of high sparsity.

When processing each layer, there is an initial loading phase where the kernels are loaded into the the accelerator's kernel memory followed by the first k input rows of the feature maps. After this initial loading phase, the controllers are activated to process the input pixels while the rest of the input feature maps are loaded in parallel. For all layers except the first one, the utilization of the 128 MAC blocks outside the initial loading phase was consistently above 99%, that is, in more than 99% of clock cycles, each MAC block was carrying out a multiplication. The zero-skipping pipeline thus efficiently utilizes the compute resources even when faced with unpredictable sparsity patterns. This pipeline, though data-dependent, achieves a MAC utilization that is on par with dense processing pipelines that carry out all the MAC operations described in Eq. 1 in a data-independent manner.

The first convolutional layer in the VGG networks is special since the accelerator performance is limited by the bandwidth of the output bus from the accelerator to the external DRAM: each output pixel has a computational cost of $3 \times 3 \times 3 = 27$ MAC operations. The 128 MAC blocks can thus dispatch on average $128/27 = 4.7$ output pixels per cycle. However, the output bus can transmit at most 2 non-zero pixels per cycle. Despite the sparsity of the output, the output bus still must slightly throttle the compute pipeline to be able to transmit the pixels. Thus, the MAC utilization of the first layer is 60%.

RTL performances at 500 MHz show how the accelerator is able to process these networks at about 13 FPS

TABLE IX: RTL simulations results (convolutional layers only)

Network	GOp/frame	ms/frame	frame/s	GOp/s	Efficiency	GOp/s/W	MAC Utilization	MAC Utilization (no kernel loading time)
VGG19	39.07	82.72	12.1	471.64	368.5%	3042.84	74.19%	97.87%
VGG16	30.69	72.94	13.71	420.83	328.8%	2715.00	78.34%	98.14%
Giga1Net	1.040	4.16	240.07	249.68	195.1%	1610.79	67.31%	87.40%
RoshamboNet	0.018	0.2	4219.40	75.95	59.4%	490.00	33.58%	65.80%
Face detector	0.002	0.0264	37864.46	75.73	59.2%	488.57	40.90%	51.05%

TABLE X: FPGA results (convolutions + fully connected + control overhead)

Network	GOp/frame	ms/frame	frame/s	GOp/s	ms/frame (CNN Only)	Efficiency (CNN Only)
VGG19	39.017	2439	0.410	16.10	1819	143.40%
VGG16	30.693	2269	0.441	17.196	1506	136.45%
Giga1Net	1.040	115.8	8.64	8.99	81.8	99.41%
RoshamboNet	0.018	5.49	182.15	3.28	5.98	22.23%
Face detector	0.002	3.289	304.05	0.61	0.57	23.31%

TABLE XI: Nullhop main features summary

Technology	ASIC: GF 28nm FPGA: Xilinx Zynq 7100
Chip Size	7.5mm^2
Core Size	5.8mm^2
#MAC	128
Clock Rate	ASIC: 500 MHz FPGA: 60 MHz
Max. Effective Throughput	ASIC: 471 GOp/s FPGA: 17.19 GOp/s
Max. Effective Efficiency	ASIC: 368% FPGA: 143%
Max. GOp/s/W	ASIC: 3042 GOp/s/W FPGA: 28.8 GOp/s/W
Arithmetic Precision	16-bit fixed point multipliers 32-bit fixed point adders

(frames/second), value high enough for real time processing The FPGA implementation, despite lower frequency and the overhead due to using the ARM as external controller, is able to process VGG16 CNN in about 1.5s, aligned with the state of the art.

B. Giga1Net

The first layer of Giga1Net is composed by sixteen 1x1x3 kernels, requiring 3 MAC operations for each output pixel. When processing 16 output feature maps, Nullhop clusters 8 multipliers to work together on a single output mapSince each 1x1 kernel in the first layer requires only 3 MAC operations, 5 MAC units are idle during the computation. Furthermore, the write of the output pixels on the bus is again a bottleneck, since now the MACs are producing 16 output pixels per clock cycle. As result, the MAC utilization in the first layer is 10%. For successive layers, the Nullhop pipeline is able to adapt better to the architecture of the network, achieving about 250 GOp/s - 195% efficiency.

The FPGA implementation suffers from the low compute performance of the ARM CPU which computes the fully-connected layers. Despite that, the system is able to classify frames at more than 8 FPS running at only 60 MHz, with an efficiency for the full system of almost 100%.

C. RoshamboNet and Face Detector

Both RoshamboNet and the Face detector are small CNNs for which the Nullhop pipeline is pushed to its limits in terms of flexibility, representing the lower limit of the NullHop efficiency. For such small networks, the main performance limitation lies in the I/O bandwidth: Nullhop MACs are forced to be idle for about 50% of the time while they wait for kernels or data to be loaded or previous results to be streamed out. Despite this bottleneck, the accelerator is able to achieve above real time performances at both 50 and 500 MHz respectively.

D. Comparison with Prior Work

Table XII provides a summary of NullHop results and comparison to prior CNN accelerators, showing how in ASIC simulations the system is able to achieve state-of-the-art performance. The most relevant result obtained is in terms of efficiency: while all other architectures suffer significant discrepancies between the theoretical peak performances and the effective ones, Nullhop is able to achieve an efficiency consistently higher than 100% (and up to 368% for a large CNN), thanks to its zero-skipping pipeline and high MAC utilization.

E. Memory Power Consumption Estimation

To estimate the total power consumption of an ASIC implementation of our system including external DRAM memory access, we collected statistics about data movement during computation. We used a DRAM memory access energy of 21 pJ/bit reported for a LPDDR3 memory model [32]. Table XIII shows results for the different test cases. Since all simulations were run at the highest possible frame rate, smaller networks, where the data I/O phase is dominant, consume more power than larger ones.

VII. CONCLUSIONS

Due to the many algorithmic benefits of sparse representations, it is important to develop computing architectures that natively exploit sparsity to reduce power consumption, memory resources utilization, and compute time. In this paper, we have developed a processing architecture whose native data encoding scheme and processing pipeline are optimized to handle sparse data representations and a flexible range of input and output feature map numbers. Our processing pipeline was developed specifically for the case of CNNs. However, the

TABLE XII: Comparison with prior work

Architecture	Core Size [mm^2]	Technology	Maximum Frequency [MHz]	Theoretical Performances [GOp/s]	Effective Performances [GOp/s] [i]	Efficiency	Effective Power Efficiency [GOp/s/W]
Nullhop[a]	5.8	GF 28nm	500	128	471	368%	3042
Nullhop[a1]	5.8	GF 28nm	500	128	420	328%	2715
Nullhop-FPGA[b]	-	-	60	15	16.1	106.95%	28.8
Nullhop-FPGA[b1]	-	-	60	15	17.2	114.24%	27.4
Eyeriss[c]	12.25	TSMC 65nm	250	84	27	32%	115
Origami[d]	3.09	UMC 65nm	500	196	145	74%	437
NeuFlow[e]	12.5	IBM 45nm	400	320	294	91%	490
ShiDianNao[f]	4.86	65nm	1000	128	-	-	-
Moors et al.[g]	2.4	40nm LP	204	102	71	69%	940
Envision[h]	1.87	UTBB 28nm	200	102	76	74%	1000

[a] VGG19 - Convolutional layers only
[a1] VGG16 - Convolutional layers only
[b] VGG19 - Full system
[b1] VGG16 - Full system
[c] VGG16 with batch size = 3 - Full system [14]
[d] Custom CNN - Full system [13]
[e] Custom CNN - Convolutional layers only [11]
[f] Multiple Custom CNN - Convolutional layers only [12]
[g AlexNet - Convolutional layers only [16]
[h] VGG16 - Convolutional layers only [17]
[i] Computed as number of MAC/SOP units times clock frequency

TABLE XIII: Power estimation using LPDDR3

Network	LPDDR3 Power [mW]	Total Power [mW]	Effective Performances [GOp/s/W]
VGG19	114	269	1751
VGG16	102	257	1634
Giga1Net	129	284	878
RoshamboNet	219	374	203
FaceNet	159	314	250

compression scheme used could also be used to encode sparse data in other types of networks.

Our compression scheme achieves a higher compression ratio than run-length encoding schemes that were previously used to compress input feature maps and allows us to operate directly on their compressed representation, as shown in Sec. III-A. This compression reduces I/O power consumption that is a crucial component of overall system power consumption as well as computational time.

We have shown in Sec. VI-A that NullHop's utilization of compute resources is consistently above 99% assuming that computation is not limited by the input or output bandwidth. NullHop allows sparse computation with high utilization comparable to pipelines operating on dense representations. Thus, NullHop achieves a speedup directly proportional to a CNN's sparsity since it does not waste cycles on zero input pixels.

ACKNOWLEDGMENTS

This work was supported by Samsung Advanced Inst. of Technology (SAIT), the University of Zurich and ETH Zurich.

REFERENCES

[1] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review,*, vol. 65, no. 6, pp. 386–408, Nov 1958.

[2] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[3] D. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 886–893.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *ArXiv e-prints*, September 2014.

[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[9] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[10] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "ENet: A deep neural network architecture for real-time semantic segmentation." [Online]. Available: http://arxiv.org/abs/1606.02147

[11] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "Neuflow: Dataflow vision processing system-on-a-chip," in *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, pp. 1044–1047. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6292202

[12] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao," *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pp. 92–104, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2749469.2750389

[13] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 199–204.

[14] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 262–263.

[15] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 A 1.42 TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 264–265.

[16] B. Moons and M. Verhelst, "A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*, vol. 2016-Septe, pp. 1–2, 2016.

[17] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247. [Online]. Available: http://ieeexplore.ieee.org/document/7870353/

[18] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[19] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks." in *Aistats*, vol. 15, no. 106, 2011, p. 275.

[20] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations

in convolutional architectures for object recognition," in *International Conference on Artificial Neural Networks*. Springer, 2010, pp. 92–101.

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

[22] E. Stromatias, D. Neil, M. Pfeiffer, F. Galluppi, S. B. Furber, and S.-C. Liu, "Robustness of spiking Deep Belief Networks to noise and reduced bit precision of neuro-inspired hardware platforms," *Frontiers in Neuroscience*, vol. 9, Jul. 2015. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4496577/

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

[24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[25] L. K. Muller and G. Indiveri, "Rounding methods for neural networks with low resolution synaptic weights," *arXiv preprint arXiv:1504.05767*, 2015.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[27] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[28] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck, "A 240×180 130 dB 3 μs latency global shutter spatiotemporal vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 10, pp. 2333–2341, 2014.

[29] I.-A. Lungu, F. Corradi, and T. Delbruck, "Live Demonstration: Convolutional Neural Network Driven by Dynamic Vision Sensor Playing RoShamBo," in *2017 IEEE Symposium on Circuits and Systems (ISCAS 2017)*, Baltimore, MD, USA, 2017.

[30] D. P. Moeys, F. Corradi, E. Kerr, P. Vance, G. Das, D. Neil, D. Kerr, and T. Delbruck, "Steering a predator robot using a mixed frame/event-driven convolutional neural network," in *2016 IEEE Conf. on Event Based Control Communication and Signal Processing (EBCCSP 2016)*, vol. in press.

[31] A. Aimar, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, F. Corradi, A. Linares-Barranco, and T. Delbruck. Roshambo NullHop CNN accelerator driven by DVS sensor, NIPS 2016. [Online]. Available: https://www.youtube.com/watch?v=KeLnZZYLexE

[32] M. Schaffner, F. K. Gürkaynak, A. Smolic, and L. Benini, "DRAM or no-DRAM?: Exploring Linear Solver Architectures for Image Domain Warping in 28 nm CMOS," *Proceedings of DATE*, pp. 707–712, 2015.