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 efficiency is less than 10 GOp/s/W for single-frame runtime inference. We propose a flexible and efficient CNN accelerator architecture called NullHop that implements SOA CNNs useful for low-power and low-latency application scenarios. NullHop exploits the sparsity of neuron activations in CNNs to accelerate the computation and reduce memory requirements. The flexible architecture allows high utilization of available computing resources across kernel sizes ranging from 1x1 to 7x7. NullHop can process up to 128 input and 128 output feature maps per layer in a single pass. We implemented the proposed architecture on a Xilinx Zynq 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. Post-synthesis simulations using Mentor Modelsim in a 28nm process with a clock frequency of 500 MHz show that the VGG19 network achieves over 450 GOp/s. By exploiting sparsity, NullHop achieves an efficiency of 368%, maintains over 98% utilization of the MAC units, and achieves a power efficiency of over 3 TOp/s/W in a core area of 6.3 mm^2. As further proof of NullHop's usability, we interfaced its FPGA implementation with a neuromorphic event camera for real time interactive demonstrations.**

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

I. Introduction

Convolutional neural networks (CNNs) have emerged as one of the most popular approaches for solving a variety of large-scale machine vision tasks [1]–[3]. The conceptual simplicity of CNNs coupled with powerful supervised training techniques have made them 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.

CNN are trained, typically using backpropagation, to produce the correct output for a set of labeled examples. The network training is usually done on hardware platforms such as graphical processing units (GPUs) or highly-specialized server-oriented architectures as in [4].

Inference in state-of-art (**SOA**) trained CNNs is computationally expensive, typically requiring several billion multiply-accumulate (**MAC**) operations per image. Using a mobile processor or mobile GPU to run inference on a CNN can become unfeasibly expensive in a power-constrained mobile platform. For example, the NVIDIA Tegra X1 GPU platform which targets mobile automatic driver assistance (ADAS) applications, can process 640x360 color input frames at a rate of 15Hz through a computationally efficient semantic segmentation CNN [5]. Processing each frame through this CNN requires about 2 billion MAC operations, thus the GPU does around 60 billion operations per second (GOp/s), at 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 CNN at only 30 frames per second (FPS) if the network requires less than 2 GOp/frame.

An important development in CNN research relevant to hardware accelerators are methods for training CNNs that use low precision weights, activations, and sometimes backpropagated gradients [6]–[11]. Training a network which has low-precision parameters and which uses the Rectified Linear Unit (ReLU) activation function leads up to 50% increased sparsity in the activations. Several reported dedicated accelerators already exploit this sparsity [12]–[14].

Because sparse networks can be beneficial for saving computes and memory access during inference, we developed a CNN hardware accelerator called *NullHop* that exploits activation sparsity by two main features: The first feature is its ability to skip over zeros (zero-skipping) in the input CNN layers without any wasted clock cycles and redundant MACs. This method is different from [15], where a zero-skipping pattern is applied only in the computation of Fully-Connected (**FC**) layers. It is also different from [16] where the convolutions are accelerated by exploiting the sparsity in the convolution kernels rather than in the activations. The second feature is a novel compression scheme that is optimized for sparsely activated CNN layers. This compression reduces

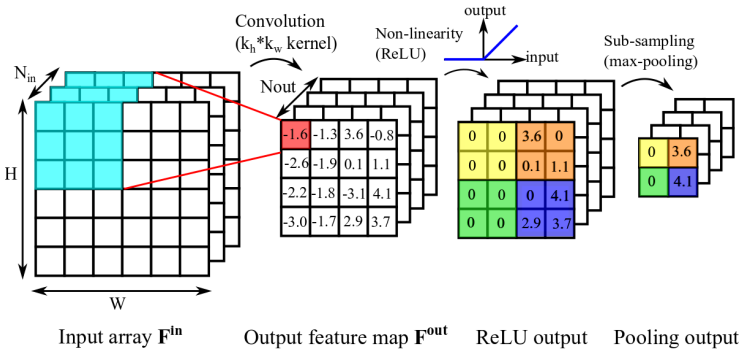


Fig. 1: The three main processing stages in a CNN.

external memory access and is more efficient than current run-length encoding schemes [12]. It is also operates directly on compressed representations, unlike the work of [13]. The processing pipeline operates directly on the compressed input representation therefore more input data can be stored in the accelerator memory. Finally, similar to the current SOA accelerators [12]–[14], [17]–[22], NullHop uses a configurable processing pipeline that maintains high efficiency across a range of CNN kernel sizes and numbers of feature maps.

II. CNN Principles of Operation

CNNs extract high-level features from input images using successive stages of convolutions, non-linearities, and sub-sampling operations. The three stages are shown in Fig. 1. A typical vision application starts with 3 input feature map channels, corresponding to the red, green, and blue color channels of the camera image. The convolution stage takes then as input a 3D array $\mathbf{F^{in}}$ with N_{in} 2D feature maps of size $H \times W$. Each filter in the filter bank, c_{ij}, is of size $k_w \times k_h$ and connects an input feature map F_i^{in} to an output feature map F_j^{out} as follows:

$$F_j^{out} = \sum_{i=0}^{N_{in}-1} c_{i,j} * F_i^{in} \tag{1}$$

The convolution output is also a 3D array, $\mathbf{F^{out}}$ composed of N_{out} feature maps of size $N_{out} \times (H - k_h + 1) \times (W - k_w + 1)$. A point-wise non-linearity is then applied to $\mathbf{F^{out}}$. In current SOA CNNs, the ReLU [23] is the most widely used non-linearity and is computed by $f(x) = max(0,x)$. In addition to being computationally cheap, using ReLUs empirically often yields better classification accuracy compared to saturating non-linearities, i.e, sigmoidal non-linearities [24]. 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 [25], where each pooling window is replaced by the maximum value in the window. An example of a 2x2 non-overlapping pooling stage is shown in Fig. 1.

A. Reduced Precision CNNs

When designing CNNs hardware accelerators, one major consideration is the energy consumption from the amount of memory access and the number of computes needed. Rounding a full precision pre-trained network to lower precision weights and activations will help to reduce hardware resources but the accuracy of the network is usually compromised. It is possible to increase the accuracy by also training deep networks using reduced bit precision methods as demonstrated in [6], [9], [26].

In order to run reduced precision CNNs on the NullHop accelerator, we developed a custom branch of Caffe[1] called ADaPTION [27]. We use it to train networks from scratch as well as to fine-tune existing 32-bit floating-point networks to any specified fixed point precision for both weights and activations using the power2quant algorithm [26]. It also includes tools for estimating the required per-layer decimal point locations. We achieved VGG16 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 the number of bits also results in up to 50% increased sparsity of activations per layer as shown in Fig 2. The average sparsity in the activations increased from 57% in floating precision to a remarkable 82% in reduced precision. An existing library in Caffe called Ristretto [28] can be used to train and test CNNs with reduced precision weights and activations but the weights are quantized only using a fixed-point notation, whereas the activations are quantized using 16-bit floating-point notation. Activations are 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. While others have investigated ultra low-precision networks (e.g. binary), these networks are 10X to 100X slower to train and require even more feature maps to achieve the same accuracy [9]–[11]. Therefore, we targeted a more generally useful hardware solution by quantizing weights and activations to an intermediate 16-bit fixed-point precision.

III. Accelerator Architecture

Figure 3 shows the high-level schematic of the NullHop accelerator. The accelerator interface is composed of two separate 32-bit data bus for the input and output, an input configuration interface that can be used by a host microcontroller for configuring the system, and a control interface for clock, reset and bus handshake signals. The accelerator implements one convolutional stage followed by a ReLU transformation and then 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 convolutional stages one after another in a sequential 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-C.

The output feature maps produced by the current layer are streamed off-chip to the external memory. They are then

[1] https://github.com/NeuromorphicProcessorProject/ADaPTION

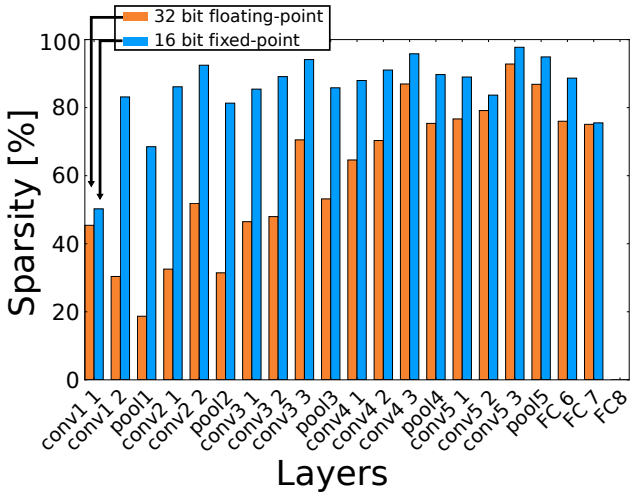


Fig. 2: Sparsity before (orange) and after (blue) activations are quantized to 16-bit fixed-point for VGG16 layers. Average over 1000 ImageNet images.

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 Processor (**IDP**) reads a small portion of the compressed input feature maps to generate pixels that are passed to the Compute Core Module (**CCM**). Only a few pixels, typically in one mini-column (of height $k_h + 1$) 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 forwards the pixels' positions (row, column, and input feature map index) to the CCM. The Pixel Allocator inside the CCM allocates each incoming pixel to a 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 (implemented in the PRE, Pooling-ReLU-Encoding module) before going to the pixel stream compression block. The compressed output feature maps are then sent off-chip.

A. Sparse Matrix Compression Scheme

NullHop uses a novel sparse matrix compression algorithm. This algorithm produces an average compression level that is higher than that obtained from using the method in [12] and is easier to decode than the Huffman coding used in [13]. The coding uses two elements: a Sparsity Map (**SM**) and a Non-Zero Value List (**NZVL**). The SM is a 3D mask, having the same number of entries as number of pixels in the feature maps. Each binary entry in the SM is 1 if the corresponding pixel is non-zero, and 0 otherwise:

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

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. 4. 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 (CIS) in bits is given by:

$$CIS = E(n + N(1 - S_p)) \quad (3)$$

where
- CIS input image size in bits
- E total number of pixels in input image
- S_p sparsity of input image (range 0-1)
- N input precision in bits
- n 1 bit

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

$$S_p > Th_p = 1/N \quad (4)$$

where Th_p is defined as the threshold sparsity. Solving (4) for different data bit precisions leads to Table I where a precision of 16 bits shows a threshold sparsity of 0.0625 which is low enough to guarantee compression in most CNN layers. Our compression algorithm shows better average compression performance than the run-length (RL) compression algorithm proposed by [12] as demonstrated for the VGG19 example in Fig. 5. In Fig. 5b it is also possible to see how the RL algorithm cannot reduce the size of the output feature maps for some layers, effectively increasing their size. Although our sparsity map algorithm produces an equivalent level of compression as the Huffman coding used by [13], our data structure permits an easier decoding during the computation, allowing the accelerator to operate directly on compressed representations without decompression.

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

Bit Precision	Threshold Sparsity
8	0.1250
12	0.0833
16	0.0625
24	0.0416
32	0.0312

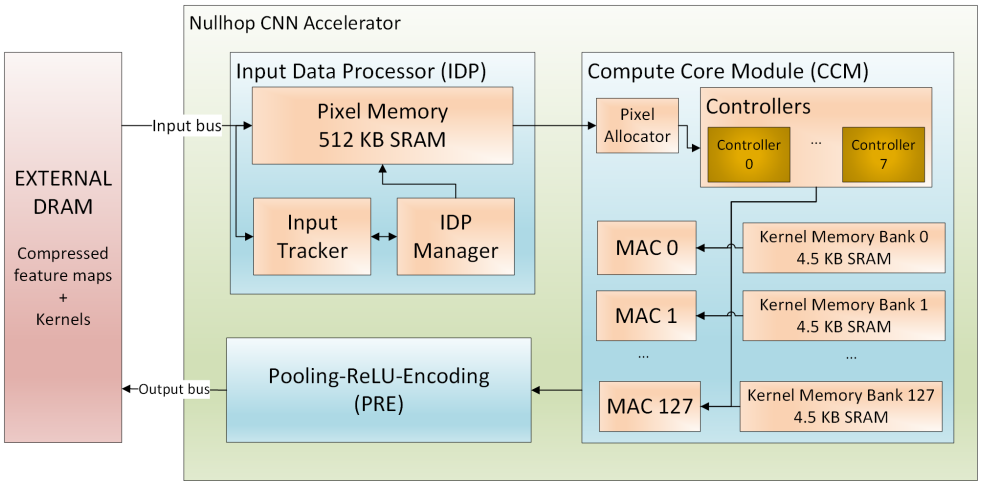


Fig. 3: High-level schematic of the proposed NullHop CNN accelerator.

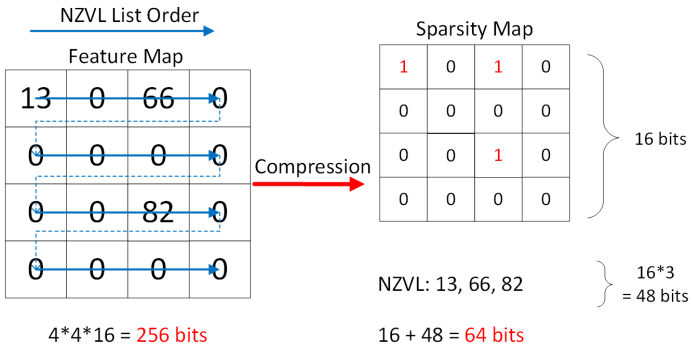


Fig. 4: Sparse compression scheme acting on a single sparse feature map. The non-zero values are stored as an ordered Non-zero Value List (NZVL). The order goes row-wise from top to bottom and from left to right in each row as shown on the feature map. The sparsity map (SM) is a binary mask with 1s at the positions of non-zero pixels, and 0s at the positions of zero pixels.

B. Pixel Memory and Decoding: Input Data Processing Unit

1) Input format: The 3D SM is split into 16-bit word segments that are streamed to the CNN accelerator interleaved with the corresponding NZVL pixels. The length of the SM segments is implementation dependent and in our case, it is equivalent to the chosen bit precision of the activation to simplify the hardware implementation.

The number of ones in a SM segment indicates the number of pixel values that will follow the SM segment as shown in Example 1 of Fig. 6. The position of the ones indicates the offsets of these pixels. The first word that 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, a SM segment can be all zeros as shown in Example 2 of Fig. 6. The SM segment will then be followed by another SM segment and this will continue until a 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 $p(i,x,y)$ be the pixel at position (x,y) of the i^{th} input feature map, F_i^{in}. The pixels are streamed into the accelerator as follows: $p(0,0,0), p(1,0,0), .., p(N_i,0,0), p(0,1,0), .., p(N_i,1,0), ..$

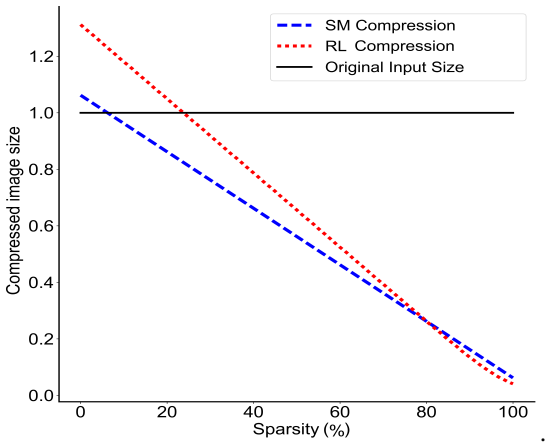
2) Decoding the compressed rows: The IDP is responsible for storing and decoding the compressed rows of the input feature maps. The module contains multiple SRAM banks and can start decoding the data from these banks while the input feature maps are still being loaded. The IDP maintains a pointer to the beginning of each row of the image stored inside the SRAM and uses these row starting addresses to decode the pixels in a sequential manner.

At each clock cycle, the IDP can read up to k_h+1 non-zero pixels from memory to be sent to the CCM module (Fig. 3). The pixels are read in a winding manner within a vertical stripe as shown in Fig. 7. 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 CCM module.

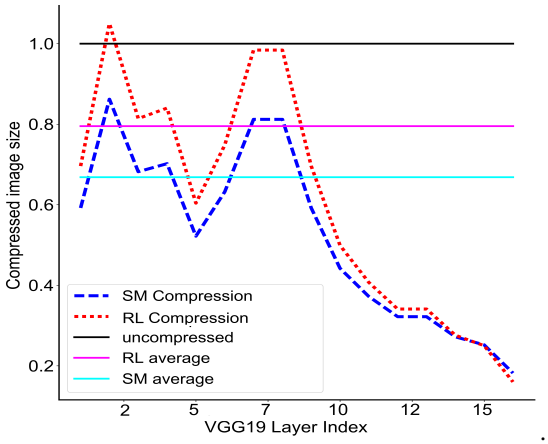
There are three main blocks inside the IDP module:

a) Pixel Memory: The Pixel Memory block 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 memory, followed by read requests received by the IDP Manager (described in Sec. III-B2c). The block also handles the communication between the IDP and CCM modules, sending pixel values to the processing unit and stalling IDP operations if CCM cannot receive more data.

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 starting position addresses received by the Pixel Memory in a small SRAM memory when a new row is stored; and providing them to the IDP Manager



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



(b) Comparison on 1000 runs of VGG19.

Fig. 5: Compression performance comparison between the Sparsity Map (SM) and Run-Length (RL) Compression algorithms.

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, which consists of a set of FSMs acting as control units for the IDP module. The number of FSMs is equal to the maximum kernel size supported plus 1; each control unit sends read requests to Pixel Memory for a specific row in the currently active (vertical) stripe. During processing, k_h+1 FSMs are enabled, while others are disabled. For example, for $k_h = 3$ there are $3+1 = 4$ enabled FSMs, each one reading a different row of the input. Each FSM stores in its internal registers, a SM and a memory pointer to the next address to be read, plus a register containing the spatial coordinates of the next pixel to be read.

At the beginning of the layer, the IDP Manager reads the pointers to the first k_h+1 rows from the Input Tracker and stores them into each IDP FSM memory pointer register. The FSMs then send a read request to the Pixel Memory and store the read result into their SM registers. In the same clock

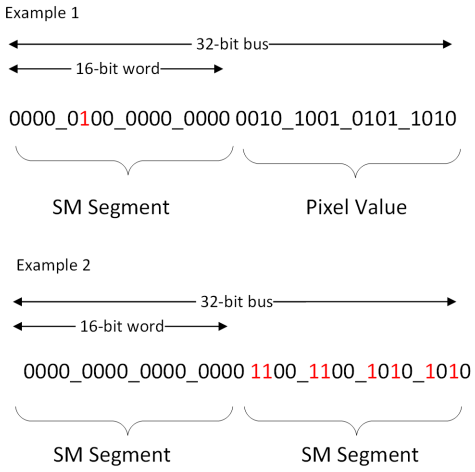


Fig. 6: Format of the words sent to the accelerator, using 16-bit words and 32-bit bus as an example. 16-bit SMs are interleaved with non-zero pixel values. If a SM has all-zero entries, it is followed by next SM.

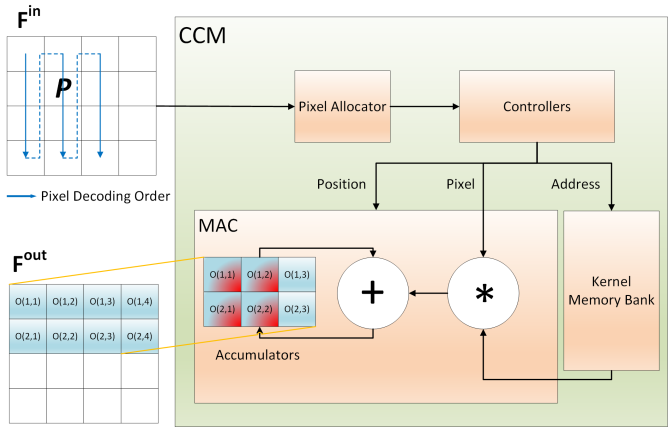
cycle, the position of the first non-zero entry is computed using the SM itself and the memory pointer register is incremented by one before the next read operation. Next, clock cycles are dedicated to the actual reading of pixels from memory: The FSMs first send a read request to the Pixel Memory for the current memory pointer (providing its spatial coordinates as extra information to be forwarded to CCM). They then increment the memory pointer by 1, and look 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 module iterates over SMs and pixels until the row ends, moving to the next feature map stripe when all FSMs have completed their task. No pixel with zero value is forwarded to the CCM module.

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

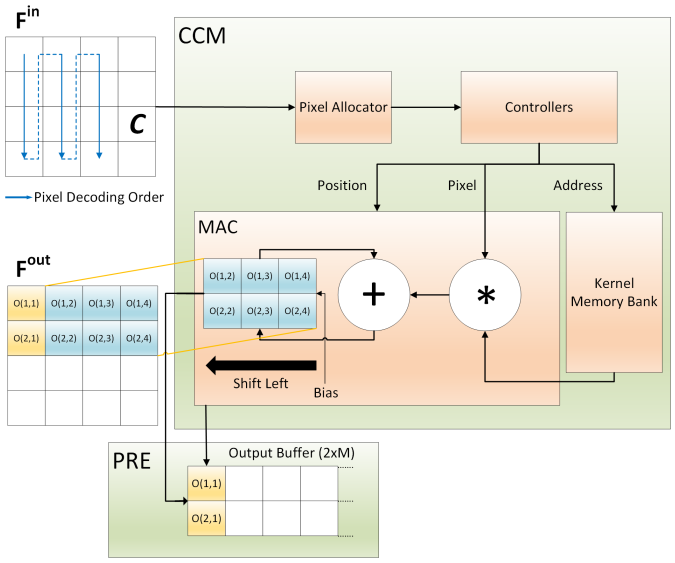
The CCM is the arithmetic unit of the architecture and computes the convolution output $\mathbf{F}^{\mathbf{out}}$. It is composed of the following blocks:

- A *Kernel Memory* with M SRAM banks. It stores the kernel weights necessary to compute the current layer.
- A *MAC block* containing M MACs, each composed of a multiplier, an adder, and $2 \times k_w$ accumulators.
- A *block* of *C Controllers*, each overseeing the weights read from the Kernel Memory and redirecting the weights and activations to a subset of MACs called *cluster*.
- A *Pixel Allocator* that oversees the redirection of the pixels received from the IDP to the different controllers.

a) Basic Operations: We now describe the simplest accelerator configuration where $N_{out} = M$. At the beginning of a layer computation, the weights are loaded in the Kernel Memory. Each kernel is stored in a different Kernel Memory Bank and assigned to a MAC block, and the size of the only existing cluster is M. The Pixel Allocator forwards pixels from



(a)



(b)

Fig. 7: Pixel processing scheme for an 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 results in the current output patch. (Sec. b) On receiving pixel C, the controller detects (based on the column index of C) that the computation of output pixels stored in the most-left entries of the accumulator array (pixels $o(1,1)$ and $o(2,1)$) is complete. Pixels $o(1,1)$ and $o(2,1)$ are shifted into an output buffer.

the IDP to the only active Controller. At every clock cycle, the Controller sends a single address, based on the coordinates x and y of the pixel itself, to all the Kernel Memory banks. Consequently, all MACs receive the same pixel coordinate and weights from different kernels, each computing the pixel value for a different output channel.

b) Kernel Read Scheme: NullHop computes $\mathbf{F^{out}}$ in a row-wise manner, with 2 output rows computed at the same time to enable on-fly 2x2 pooling. In a single MAC, pixel P is multiplied by all the weights of two rows of a kernel. Each multiplication result is added up in a different entry of the MAC accumulators according to the coordinates of P and of the weights. Fig. 7a illustrates this process for the case of 1 input feature map and when $k_w = k_h = 3$. P is multiplied by the weights in position $k_{0,0}$, $k_{0,1}$, $k_{1,0}$, $k_{1,1}$. Since it is close to the left image boundary, it is not necessary to multiply it with weights in position $k_{0,2}$ and $k_{1,2}$, since these multiplications do not contribute to any value of $\mathbf{F^{out}}$.

Fig. 7b shows what happens when a pixel lies in a different column than the previous one. In this example, the Controller detects that there is a column index increase. Because the IDP sends the pixel along vertical stripes, the values stored in the left-most entries of the MAC accumulators are the convolution results for the previous column. The Controller asserts a shift flag to the MAC accumulators, which are shifted left and sent to the output buffer inside the PRE module (described in Sec. III-D). The right-most entries of the accumulators are initialized with the value of the bias, ready for the computation of next output column. This process repeats: As the input decoding moves to a new column, the accumulators are shifted one position to the left until all the pixels in a double row of $\mathbf{F^{out}}$ are produced.

c) Different Input Feature Maps: When the number of output feature maps of a layer is different from M, the function of the Pixel Allocator and the Controllers changes to redistribute the workload over multiple MACs.

1) $N_{out} < M$: Since there are more MACs than output feature maps, it is impossible to perform a 1-to-1 mapping. Instead, NullHop splits the computation of a single output channel over $v = M \div N_{out}$ MACs. The Pixel Allocator sends, in parallel, pixels to v Controllers. Each controller is in charge of issuing commands to a *cluster* composed of N_{out} MACs and a Kernel Memory Bank. Every MAC stores partial convolution results that will be summed up in the PRE module to produce $\mathbf{F^{out}}$.

2) $N_{out} > M$: In this case, it is impossible to assign an output feature map to every MAC block. To handle this situation, NullHop splits the computation over multiple passes. In each pass, we process a subset of the output feature maps. First, we load the first M kernels and compute $F^{out}_{(0:M-1)}$. Then, we load a second set of M kernels and compute $F^{out}_{(2M:2M-1)}$. The procedure is repeated until all the N_{out} channels are processed. If N_{out} is not a multiple of M, the system distributes the kernels evenly among passes and computes them as described for $N_{out} < M$.

3) $N_{out}\ [KBytes] > Kernel\ Memory\ Bank\ [KBytes]$: In some cases, it is possible that M kernels cannot be allocated to M memory banks because they would not fit in the SRAM banks (e.g. in case of 7x7 convolutions with M input channels). Instead of increasing the memory size to cover also these corner cases, NullHop splits the kernels over multiple memory banks and passes similar to what was described in 1) and 2).

D. Pooling, ReLU, and Encoding Unit

The pooling, ReLU, and Encoding module (**PRE**) is responsible for the final processing steps of the computational pipeline. It receives the convolution results shifted out from the MACs in the CMM and stores them in the PRE buffer. In the current implementation, the buffer size is equal to

$2 * M$ memory entries where 2 is the pooling dimension. 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 Sec. 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 produce partial convolution results. Before the data can be further processed, these partial results need to be summed to produce the $\mathbf{F}^{\mathbf{out}}$ values. The accumulation is performed within the PRE buffer, in $log(size(cluster)) + 1$ clock cycles. During each accumulation cycle, two adjacent partial values are summed together and the result is stored back in the PRE buffer. Once the accumulation is over, the results are transferred to the output buffer which can store up to M 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 three values of the 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 CCM, 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 B pixels of the output buffer, a number of pixels equal to the bit precision of the chip. In our 16-bit implementation, the encoder works on 16 pixels at a time. The encoder generates a SM segment from the first B pixels based on the position of the non-zero pixels. For each set of B pixels, the encoder streams out the SM and the first non-zero pixel during the first clock cycle; then, two pixels are sent out on every subsequent clock cycle. When all the pixels of the current SM have been streamed out, the output buffer is shifted by B 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

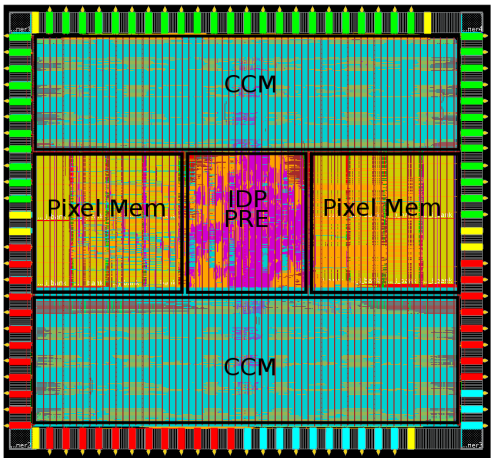


Fig. 8: NullHop chip place and route. Pads for power (yellow), input data and input configuration bus (green), output data bus (red) and control signals (blue) are highlighted.

- Number of controllers: 8
- Max supported kernel size: 7
- Max number of rows in input image[2]: 512
- Max number of columns in input image[2]: 512
- Max number of feature maps[2]: 1024

A. Synthesis, Place and Route

We synthesized NullHop with Synopsys Design Compiler using Globalfoundries 28nm technology. From this manufacturing process, we used the library with the smallest available standard cells, including all the different threshold voltages except for the lowest. This choice was to reduce area and power for embedded system deployment. Fig. 8 shows the results of the place and route, including the position of the I/O pads. The post place-and-route core size is 6.3 mm^2 (core utilization = 70%) and the chip size including I/O pads is 8.1 mm^2, with an estimated power consumption of 155 mW at 1 V supply voltage and a clock frequency of 500 MHz. The power consumption was estimated using switching activity with Synopsys Design Compiler. The result allows us to estimate the compute performance per Watt of the accelerator as reported in Table XI.

B. FPGA Implementation

To validate the design, we 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 FPGA fabric with its ARM processor. The ARM CPU runs Petalinux as the operating system (OS) to control the accelerator. It manages the read and write operations between the DDR memory of the ARM computer and the BRAM of the accelerator. The processor also computes the FC layers following the convolutional layers. We included in the OS an embedded USB host controller module to interface it to

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

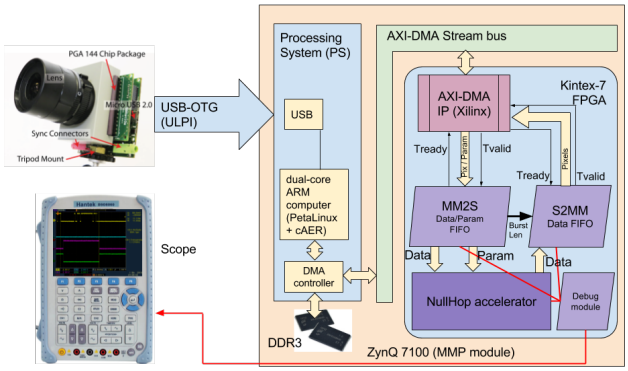


Fig. 9: System-on-Chip (SoC) block diagram for the testing scenario. The FPGA on the Zynq SoC hosts the NullHop accelerator plus glue logic for interfacing to the Zynq ARM processor DDR memory using the DMA.

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%)

an iniLabs DAVIS240C neuromorphic event-based camera [29] for the real- time demonstrations described in Sec. V. For this use, the ARM processor also runs iniLabs cAER[3], an open source framework to interface to the DAVIS camera. Fig 9 shows the block diagram of our FPGA architecture, including the MM2S (*Memory To Stream*) and S2MM (*Stream to Memory*) modules used to interface the accelerator with the AXI4-S bus.

The design minimizes host memory manipulation by using DMA transfers from host memory to the accelerator and vice versa without requiring each layer output to be reformated or processed. 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 the complete functionality required at least 12 months of effort beyond the basic reference IP from Xilinx.

Our IC implementation targets a clock frequency above 500 MHz, but routing 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 which includes NullHop and the AXI interfaces. Table II shows the required resources.

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

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
CCM	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

Power analysis for the implemented NullHop on the Zynq 7100 was estimated with Vivado assuming half the nodes switch state on each clock cycle for the logic. These estimates show a static power consumption of 316 mW when the 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 the breakdown of the power estimate across the blocks. The ARM consumes 63% of the total power, while the logic circuits and memory on the Kintex-7 embedded FPGA consume the remaining 37%. The power breakdown for the FPGA blocks is estimated as 27.27% for CCM, 4.14% for IDP, 1.75% for PRE, and 1.15% for AXIstream.

To validate the Xilinx estimations, we measured the power consumption of the different stages of the complete testing infrastructure when running the example described in Sec. V-C. The base-board that powers the AvNet 7100 SoC mini-module consumes 5.1 W. After adding the 7100 and the fan, but with no design in the FPGA and in reset mode, it consumes 6.95 W. After programming the FPGA, in an idle linux state not running cAER, the system consumes 8.27 W. Continuously running RoShamBo increases the power to 9 W. Thus the incremental power is about 750 mW, close to Vivado's estimate of 777 mW.

V. Application Example

We studied several CNNs, ranging from small custom CNNs on a classification task with a small number of labels to widely-used large CNNs used to classify ImageNet, a large dataset with 1000 classes.

A. VGG16 and VGG19 networks

The VGG16 and VGG19 networks [30] are large CNN architectures used for classifying the ImageNet dataset. For 224x224x4 input images, they require 31 GOp/frame and 39 GOp/frame respectively, and can be trained to achieve 68.3% and 71.3% Top-1 accuracy using our ADaPTION tool. For evaluating the hardware performance of NullHop reported in Sec. VI, we used randomly chosen images from the dataset. Because of the image size, these two networks require particular arrangements so that they can be implemented on NullHop as discussed next:

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	No	1
5	64	64	5	22x22	No	1
6	64	64	5	20x20	No	1
7	64	128	3	18x18	No	1
8	128	128	3	18x18	No	1
9	128	128	3	18x18	No	1
10	128	128	3	18x18	No	1
11	128	128	3	18x18	Yes	1

a) Number of output feature maps: Since there are 128 MAC blocks and at least one MAC block must 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 which has 256 output feature maps. 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 NullHop at the beginning of each pass.

b) IDP memory size: If the input feature maps cannot fit into NullHop's SRAM and there are more than 128 output feature maps, the input feature maps must be streamed in multiple times. The probability of multiple passes is lower 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 module was verified as sufficient to avoid multiple streaming for all tested input images.

c) Kernel memory size per 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 (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 for one pass through the input feature maps.

B. Giga1Net

Giga1Net is a 1 GOp/frame (about two-thirds the size of [1]) that we designed for stressing our accelerator. It contains various network layer configurations (Table VI). Rather than achieving high accuracy as the main goal, the purpose of the Giga1Net architecture is to identify inefficiencies that arise in the hardware pipeline from the use of multiple combinations of different kernel sizes (1x1, 3x3, 5x5, 7x7), different number of output feature maps (from 16 to 128), the presence or absence of the pooling operation and zero padding.

C. RoshamboNet

RoshamboNet is a 5-layer, 20 MOp, 114k weight CNN architecture, described in Table V, trained to play the rock-scissors-paper game [31]. This network can classify input images of size 64x64 obtained from the DVS of a DAVIS camera using the same training and feature extraction stage approach from [32]. The network outputs 4 classes: "rock", "scissors", "paper" or "background" from each feature vector. Each frame is a 2D histogram of 2k DVS events. Because of the asynchronous nature of the DVS output, the frame rate varies from <1 Hz to over 400 Hz depending on the speed of the moving hand.

We connected NullHop to a DAVIS camera, a robot hand, and an LED display, driven by a customized version of cAER running on the Zynq ARM processor. It was demonstrated at NIPS 2016 Live Demonstration track, in software form at ISCAS [31], [33], and four more public science events in 2017 and 2018. It beats human opponents by recognizing the player's 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 small CNN designed to recognize whether a face is present or absent in an image obtained from the DAVIS camera. The DVS events are accumulated into 36x36 input images, again using the method of [32]. The network 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 and on our Xilinx Zynq platform. Results are summarized in Tables VII and VIII.

A. VGG19 and VGG16

When big and deep networks with a high level of sparsity are computed on NullHop, it achieves more GOp/s than the ideal maximum (128 GOp/s, equal to the number of MAC units times their clock frequency) because operations are skipped, thereby 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

TABLE VII: 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 VIII: 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 IX: NullHop main features summary

Technology	ASIC: GF 28nm FPGA: Xilinx Zynq 7100
Chip Size	8.1mm^2
Core Size	6.3mm^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

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 (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 that the accelerator is able to process these networks at about 13 FPS (frames/second), a value which is high enough for real time processing. The FPGA implementation, despite the lower frequency and the overhead due to using the ARM as the external controller, is able to process the VGG16 CNN in about 1.5s, a number that is aligned with state of the art.

B. Giga1Net

The first layer of Giga1Net is composed of 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 map. Since 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 the RoshamboNet and Face Detector are small CNNs for which the Nullhop pipeline is pushed to its limits in terms of flexibility, representing the lower limit of 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, NullHop achieves faster than real time performance at only 50 MHz clock frequency.

D. 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 an LPDDR3 memory model [34], and ran each network as fast as possible. Table X shows the results. For small networks, the number of operations necessary to compute the convolutions is low and the accelerator spends most of the time performing memory transfer to load and store kernels and activations. The result is that both RoshamboNet and FaceNet have higher memory power consumption than the

larger VGG16 and VGG19 networks. In the case of VGG16, the overall memory transfer is an average of 42 MB/frame. The only available comparison is Eyeriss [12], which reports 341 MB for a batch of 3 VGG16 images. In batch mode, Eyeriss transfers about 113 MB/frame, which is almost 3 times more I/O than Nullhop.

TABLE X: Power estimation using LPDDR3

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

VII. Discussion

A. Comparison with Prior Work

Table XI 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 other architectures suffer significant discrepancies between the theoretical peak performances and the effective ones, with exception of [16], NullHop is able to achieve an efficiency consistently higher than 100% (and up to 368% for a large CNN), because of its zero-skipping pipeline and high MAC utilization. It confirms the validity of the concept of sparse computation introduced for fully-connected layers in [15] and, for the first time, takes it to the convolutional layers that constitute the bulk of CNN computation. This advantage does not have any drawback since all other main specifications – core size, frequency and number of MAC units – are similar or better than other proposed solutions. The only ASIC with higher throughput is Cambricon-X [16], which achieves a remarkable 544 GOp/s in only 6.3 mm^2, but implements twice the number of multipliers and runs at twice the frequency of NullHop. However, Cambricon-X is about 3 times less power efficient than NullHop.

Both implementations in [22] and [20] have a higher power efficiency than Nullhop, but provide consistently lower performances (<350 GOp/s) using more MAC units. They also require a larger area (16 mm^2), but this is justified by their support for Recurrent Neural Networks and variable bit precision.

B. Conclusion

Sec. VI-A showed that the utilization of MACs in NullHop is consistently above 99% if not limited by the input or output bandwidth. NullHop allows sparse computation with high utilization comparable to pipelines operating on dense representations. Thus, it achieves a speedup directly proportional to a CNN's sparsity since it does not waste cycles on zero input pixels, while preserving a high level of CNN architecture flexibility. NullHop has quite high precision of 16-bit weights and states, making it easy to train CNNs with acceptable accuracy compared with the difficult CNN hyperparameter tuning and extended training time required by super-reduced precision networks, e.g. [11]. Even so, a dedicated IC implementation of NullHop would achieve state-of-art power efficiency of 3 TOp/s/W.

Sec. III-A showed that the NullHop sparsity map compression achieves a higher compression ratio than run-length encoding schemes that were previously used to compress input feature maps and allows direct operation on their compressed representation. This compression reduces I/O power consumption that is a crucial component of system power consumption and computing time, making NullHop suited for embedded systems applications.

NullHop sprang from the fundamental neural organizing principle of sparse computation resulting from threshold-linear neural activation [23], [24], [36]. Only later did it become clear that exploiting sparsity also allows large reductions in power consumption, memory access, and compute time, all of which are key to deployment for real world applications. The NullHop data encoding and processing pipeline are optimized to handle spatially-sparse data representations with a flexible range of input and output feature maps. NullHop was developed specifically for the case of CNNs. However, its compression scheme can also be used to encode sparse data in other types of networks. Exploiting *temporal* sparsity can greatly reduce recurrent neural network memory access [37], [38]. Future architectures will certainly achieve even higher efficiency by combining these spatial and temporal sparsity principles with compression and controlled precision, even without departing from the immense practical advantages of synchronous logic.

Acknowledgments

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

References

[1] 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.

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

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

[4] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA2017*.

[5] 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

[6] 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.

[7] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: ImageNet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016.

[8] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

TABLE XI: Comparison with prior work

Architecture	Core Size [mm^2]	Technology	Maximum Frequency [MHz]	Theoretical Peak Performance [GOp/s][1]	Effective Performance [GOp/s][2]	Efficiency[3]	Effective Power Efficiency[4] [GOp/s/W]
NullHop[a]	6.3	GF 28nm	500	128	471	368%	1571 (3042[+])
NullHop[a1]	6.3	GF 28nm	500	128	420	328%	1634 (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[+]
ShiDianNao[e]	4.86	65nm	1000	128	-	-	-
Moons et al.[f]	2.4	40nm LP	204	102	71	69%	940[+]
Envision[g]	1.87	UTBB 28nm	200	102	76	74%	1000[+]
Cambricon-X	6.38	TSMC 65nm	1000	512	544	106%	571
DNPU[l]	16	65nm	200	300	270	90%	4200[m,+]
Yin et al.[n]	19.36	TSMC 65nm	200	409.6	368.4	90%	1027
UNPU[o]	16	65nm	200	-[p]	345.6	-	3080[+]

[1] Computed as number of MAC/SOP units times clock frequency
[2] Measured performance
[3] Effective GOp/s divided by theoretical peak performance
[4] Including memory power consumption. Marker [+] indicates core-only power consumption or not specified.
[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 [12]
[d] Custom CNN - Full system [35]
[e] Multiple Custom CNN - Convolutional layers only [18]
[f] AlexNet - Convolutional layers only [13]
[g] VGG16 - Convolutional layers only [14]
[l] Value computed using the reported fps for 16-bit AlexNet [22]
[m] Device working at low frequency (100 Mhz) [22]
[n] AlexNet - Full system [21]
[o] Unspecified 16-bit precision network [20]
[p] Serial multiplier implementation [20]

[9] 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.

[10] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," *arXiv:1511.06393*, Nov. 2015. [Online]. Available: http://arxiv.org/abs/1511.06393

[11] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "WRPN: Wide Reduced-Precision Networks," *arXiv:1709.01134 [cs]*, Sep. 2017, arXiv: 1709.01134. [Online]. Available: http://arxiv.org/abs/1709.01134

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

[13] 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-Sept, pp. 1–2.

[14] 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.

[15] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *arXiv: 1602.01528*, 2016.

[16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2016-Dec. IEEE, oct 2016, pp. 1–12.

[17] 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*.

[18] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao," *ISCA 2015*.

[19] 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.

[20] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: a 50.6 TTOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *2018 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, feb 2018.

[21] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei, "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *2017 IEEE Symposium on VLSI Circuits*.

[22] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, feb 2017, pp. 240–241.

[23] 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.

[24] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, Jun. 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.html

[25] 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.

[26] 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.

[27] M. B. Milde, D. Neil, A. Aimar, T. Delbruck, and G. Indiveri, "ADaPTION: Toolbox and Benchmark for Training Convolutional Neural Networks with Reduced Numerical Precision Weights and Activation," *arXiv:1711.04713*, 2017.

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

[29] 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, 2014.

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

[31] 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.

[32] 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)*.

[33] 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

[34] 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.

[35] 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*, 2015.

[36] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, pp. 947–951, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1038/35016072

[37] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu, "Delta Networks for Optimized Recurrent Network Computation," in *PMLR*, Jul. 2017, pp. 2584–2593. [Online]. Available: http://proceedings.mlr.press/v70/neil17a.html

[38] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A Power-efficient RNN Accelerator," in *FPGA 18*, Monterey, CA, Feb. 2018.

Alessandro Aimar received the B.Sc. degree in Physical Engineering from Politecnico di Torino (Italy) and the M.Sc. degree in Nanotechnologies from a joint program of Politecnico di Torino, INP Grenoble (France) and EPFL (Switzerland). After working as engineer at Imagination Technologies (UK) he joined the Institute of Neuroinformatics for his PhD. In 2017 he founded Synthara Technologies, a startup focused on deep learning and neuromorphic hardware.

Hesham Mostafa obtained his masters in electrical engineering from the Technical University of Munich in 2010, and his PhD in Neuroinformatics from the Institute of Neuroinformatics in 2016. He is currently a post-doc at the integrated systems neuroengineering lab at the Institute of Neural Computation at UC San Diego.

Enrico Calabrese obtained the B.Sc. degree in physics from Universita' Degli Studi di Ferrara, and the joint M.Sc. degree in micro and nanotechnologies for integrated systems from Politecnico di Torino, Grenoble INP and EPFL. He is currently a PhD candidate at the Institute of Neuroinformatics. His research interests include Deep Learning algorithms for artificial vision and attention.

Antonio Rios-Navarro received the B.S. degree in computer science engineering, the M.S. degree in computer engineering, and the PhD degree in neuromorphic engineering from the University of Seville, Seville, Spain, in 2010, 2011, and 2017, respectively. He is currently a post-doc at the Computer Architecture and Technology department, University of Seville. His current research interests include neuromorphic systems, real-time spikes signal processing, FPGA design and Deep Learning.

Ricardo Tapiador-Morales received the B.S. degree in Computer Sciences in 2015 and the M.S. degree in Computer Engineering in 2016, from the University of Seville, Spain.He is currently a PhD student of Computer Architecture and Technology department,University of Seville. His research interests include neuromorphic engineering, convolutional neural networks, FPGA design and embedded systems.

Iulia-Alexandra Lungu received her BSc in Bioinformatics from the University Claude Bernard Lyon and her MSc in Computational Neuroscience from the Technical University Berlin. She is now pursuing her PhD degree at the Inst. of Neuroinformatics, where she works with talented hardware designers to develop efficient and powerful algorithms and hardware solutions for AI. Her main interests revolve around incremental and reinforcement learning.

Moritz B. Milde received the B.Sc. degree in biomimetics from Westphalian University of Applied Sciences and the M.Sc. in neurobiology from Bielefeld University. He is currently a PhD candidate at the Institute of Neuroinformatics, where he focuses on event-driven vision-based scene understanding for robotic navigation.

Federico Corradi (M13) received the B.Sc. degree in physics from Universita Degli Studi di Parma, Italy, the M.Sc. degree (cum laude) in physics from La Sapienza University, Rome, Italy, and the Ph.D. degrees in Natural Sciences from the University of Zurich, and in neuroscience from Neuroscience Center Zurich, Switzerland. He is currently with imec in Holland.

Alejandro Linares-Barranco (M04-SM17) received the B.S. degree in computer engineering, the M.S. degree in industrial computer engineering, and the Ph.D. degree in computer engineering (specialized in computer interfaces for neuromorphic systems) from the University of Seville in 1998, 2002, and 2003, respectively. He is Associate Professor in the University of Seville since 2009 and head of the Architecture and Computer Tech. Dept.

Shih-Chii Liu (M02SM07) studied electrical engineering as an undergraduate at the Massachusetts Institute of Technology and received the Ph.D. degree in the computation and neural systems program from Caltech in 1997. She worked at various companies including Gould American Microsystems, LSI Logic, and Rockwell International Research Labs. Currently, she is a group leader at the Institute of Neuroinformatics.

Tobi Delbruck (M'99SM'06F'13) received the B.Sc. degree in physics from UC San Diego in 1986 and PhD degree from Caltech in 1993. He is currently a Professor of Physics and Electrical Engineering at ETH Zurich with the Institute of Neuroinformatics, where he has been since 1998. His group with SC Liu focuses on neuromorphic sensory processing and efficient deep learning.