Analysis of a Pipelined Architecture for Sparse DNNs on Embedded Systems

Adrián Alcolea Moreno, Javier Olivito, Javier Resano, and Hortensia Mecha

***Abstract*— Deep neural networks (DNNs) are increasing their presence in a wide range of applications, and their computationally intensive and memory-demanding nature poses challenges, especially for embedded systems. Pruning techniques turn DNN models into sparse by setting most weights to zero, offering optimization opportunities if specific support is included. We propose a novel pipelined architecture for DNNs that avoids all useless operations during the inference process. It has been implemented in a field-programmable gate array (FPGA), and the performance, energy efficiency, and area have been characterized. Exploiting sparsity yields remarkable speedups but also produces area overheads. We have evaluated this tradeoff in order to identify in which scenarios it is better to use that area to exploit sparsity, or to include more computational resources in a conventional DNN architecture. We have also explored different arithmetic bitwidths. Our sparse architecture is clearly superior on 32-bit arithmetic or highly sparse networks. However, on 8-bit arithmetic or networks with low sparsity it is more profitable to deploy a dense architecture with more arithmetic resources than including support for sparsity. We consider that FPGAs are the natural target for DNN sparse accelerators since they can be loaded at run-time with the best-fitting accelerator.**

***Index Terms*— Convolutional neural network (CNN), deep neural network (DNN), efficiency, embedded systems, field-programmable gate array (FPGA), sparsity.**

I. Introduction

DEEP neural networks (DNNs) have emerged as an outstanding model to solve complex problems in a wide variety of fields, such as computer vision, speech recognition, natural language processing, or audio recognition.

Convolutional neural networks (CNNs) are one of the most popular DNN models. Their core consists of several convolutional layers where the input, called activation or feature map, is convolved with a set of filters. The number of layers and filters, and the size of the activations turn out these models into computationally intensive and memory-demanding tasks. In computer vision, state-of-the-art CNNs require many millions of multiply-and-accumulate (MAC) operations to process a single image. High-performance general purpose processors (CPUs) and graphics processing units (GPUs) have been the natural target to execute these models on nonbattery-dependent systems. However, more energy-efficient architectures are needed for embedded systems.

Previous works have demonstrated that there are several strategies to reduce the computational and memory requirements of CNNs, with minimal impact on accuracy. Regarding the arithmetic, it is feasible to move from floating point to fixed point [1]–[3], and work with lower bitwidth [4]–[6]. Other works propose to speedup the MAC operations by using approximated outputs [7], [8]. Regarding the size of the models, pruning techniques force many parameters to zero, enabling data compression and reducing the number of useful MAC operations. This approach enables drastic reduction, both in model size and computational workload, of DNNs, which is essential for embedded systems.

Deep compression [4] constitutes a good reference for these techniques. Their authors achieve network size reductions from 35× to 49× while preserving accuracy in several popular DNNs (AlexNet, VGG, and LeNet) by using pruning, quantization, and compression. Another relevant reference is SqueezeNet [9]. In this study, the authors present a CNN with 50× fewer parameters than AlexNet with no loss in accuracy. Again, this result is achieved by using pruning and compression techniques.

Pruning techniques turn the CNNs into sparse models, and this sparsity can be exploited by including specific support to avoid useless operations, i.e., those ones where at least an operand is zero.

Although the benefits of these techniques are proved, they also introduce some challenges that must be carefully addressed in order to minimize their overhead. Compression requires support to manage the irregularities that arises in addressing and sparsity demands hardware to identify useful operations. Moreover, sparsity generates random memory-access patterns that can significantly degrade the performance of a parallel architecture due to the memory conflicts.

We have designed an accelerator that takes advantage of the optimization opportunities offered by sparsity in DNNs. Our accelerator works with compressed filters, performs only useful operations, and retrieves only useful values from memory by identifying those operations where both operands are different from zero. To this end, we included an additional

Manuscript received January 16, 2020; revised May 28, 2020; accepted June 14, 2020. This work was supported in part by Ministerio de Economía y Empresa (MINECO)/Agencia Estatal de Investigación (AEI)/European Regional Development Fund (ERDF) (EU) under Grant TIN2016-76635-C2-1-R, Grant TIN2017-87237-9, and Grant PID2019-105660RB-C21; and in part by the Aragón Government Research Group on under Grant ERDF 2014-2020. *(Corresponding author: Adrián Alcolea Moreno.)*

Adrián Alcolea Moreno, Javier Olivito, and Javier Resano are with the Group of Computer Architecture, University of Zaragoza, 50009 Zaragoza, Spain (e-mail: alcolea@unizar.es; jolivito@unizar.es; jresano@unizar.es).

Hortensia Mecha is with the Group of Dynamically Reconfigurable Hardware, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: horten@ucm.es).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2020.3005451

data structure, called indices tensor, consisting of a single bit per element that indicates whether that element is zero or not. This structure allows intelligent recognition of useful operations using simple bit operations, and compression of data by not storing values that are zero. Moreover, we have included specific support to reduce the memory conflicts, and we have explored the tradeoffs in performance, area, and energy efficiency.

The register-transfer level design of our accelerator has been written in VHDL and has been implemented in a Xilinx Zynq UltraScale+ FPGA. We believe that reconfigurable field-programmable gate arrays (FPGAs) are the natural target for our study since in these platforms it is possible to use the same hardware resources to implement a conventional CNN accelerator or an accelerator with specific support for sparsity. We have taken both performance and power measures running two popular CNNs: AlexNet and SqueezeNet. We selected these networks as benchmarks because pruned models are available for the community [10], [11]. In addition, we have used synthetic data to characterize the behavior of our accelerator for different sparsity ranges.

The rest of this article is organized as follows. Section II is an overview of the related work and Section III presents our contributions. Section IV describes the compression format used. Section V introduces the procedure to identify the useful operations. Section VI describes the initial architecture used as baseline, and Section VII explains the additional architectural support included to efficiently avoid the useless operations and reduce the memory conflicts. Section VIII describes the memory hierarchy and the data flow. Section IX analyzes the scalability of the proposed architecture. Finally, Section X evaluates the experimental results, and Section XI presents our final conclusions.

II. Related Work

A recent survey [12] analyzes the state of the art and future directions of DNN support in ASICs and FPGAs. This survey identifies the exploitation of sparse data as a powerful technique for reducing computational load and memory requirements in DNNs. Another survey [13] states that "there is an emerging need for the CNN-to-FPGA tools to support compressed and sparse." Although low or medium values of sparsity can be found in any model, pruning techniques are essential to build highly sparse models. Pruning was originally proposed in [14] and [15], and, recently, many other techniques has been proposed and studied [16]–[18]. These techniques consist in an iterative process that first identifies those weights that can be set to zero and then finetunes the remaining weights. Pruned models keep a similar accuracy than the original models with fewer meaningful parameters, allowing sparsity-based optimizations. However, an efficient management of sparsity must overcome the loss of regularity that arises in memory accesses. Otherwise, trying to exploit sparsity can even negatively impact on performance as explained in [19]. Cao *et al.* [20] proposed a technique to alleviate this by applying a bank-balanced pruning method designed to optimize the parallel execution of the pruned model.

Some recent works have presented custom architectures for sparse DNNs. Cnvolutin was the first accelerator to partially avoid useless operations [21]. The authors proposed a technique to skip those operations where the value of the input activation is zero. Han *et al.* [22] presented "EIE: Efficient Inference Engine on Compressed Deep Neural Network." This engine manages compressed weights, and includes hardware support to compute only useful operations in fully connected layers. UCNN proposes a factorization technique to replace multiplications with additions, and takes advantage of filter sparsity [23], while SqueezeFlow [24] proposes a technique that transforms a sparse convolution into multiple effective and ineffective subconvolutions. After that the ineffective subconvolutions can be eliminated. Cambricon-X [25] and NullHop [26] are other accelerators for sparse CNNs. The first one exploits sparsity on filters but not on activations, whereas the second exploits sparsity on activations but not on the filters.

SCNN is a high-performance-oriented accelerator for CNNs [27]. It includes several processors, and each of them computes a convolution in parallel by computing the Cartesian product. This is a very powerful approach that allows data reuse and avoids any operation where an operand is zero. However, this architecture requires large buffers to store partial results, and crossbars to link the multipliers and the buffers. As a result, their support for sparsity increases the area of their chip by 34%, even with a 50% smaller on-chip activation memory than their dense baseline. In terms of performance, they achieve speedups from 2.19 to 3.52 for several popular CNNs.

Other relevant works are Eyeriss [28] and Eyeriss v2 [29] which are scalable architectures with hundreds of MAC units. Eyeriss proposes to identify when any of the operands is zero in order to gate the data path and save energy. Eyeriss v2 compress data in compressed sparse column (CSC) format and directly operates with the compressed data. With this approach they read only nonzero activation values. Then they look for the corresponding weights. If the weights are zero they do not carry out the computations in order to save energy. The only penalty is that this may generate some bubbles in the pipeline.

In summary, only three previous designs include support to avoid all the operations where at least one of the operands is zero: EIE, designed for fully connected layers, SCNN, designed for convolutional layers, and Eyeriss v2 that can deal with both of them. Although our design supports both fully connected and convolutional layers, we have mainly focused on convolutional layers because they are much more computationally intensive. Both SCNN and Eyeriss v2 are highly parallel systems designed for high performance and include a large amount of hardware resources whereas our design is designed for embedded systems, where hardware resources and power budget are very limited. Hence, our primary goal is efficiency. In fact, our architecture reaches almost peak performance in most situations, i.e., one useful operation per clock cycle in each MAC processor, whereas SCNN is very far from peak performance on highly sparse layers. For instance, when processing convolutional layers four and five in AlexNet, SCNN only reaches ∼25% peak performance [27] whereas for the same convolutions our design reaches ∼98%. Regarding Eyeriss v2, according to their results during the execution of AlexNet, one of our MAC processors carries out twice the number of operations per cycle than one of their MAC processors. This does not mean that our design is better; we just target different problems. They try to maximize the throughput using hundreds of MACs in parallel, whereas our design tries to maximize the performance of an embedded system with few MAC processors.

The evaluation methodology in all these previous works quantifies the gains in terms of performance and energy efficiency of their sparse architectures compared to dense architectures with the same arithmetic resources. This analysis is interesting, but it is not completely fair because the sparse architectures include more hardware resources than the corresponding dense architectures. In our experiments, we compare architectures with similar area. To this end, we include additional arithmetic resources in the dense architecture. With this approach, for a given scenario, we can identify whether it is better to use hardware resources to exploit sparsity or to use them to carry out more MAC operations in parallel.

Another limitation of the previous approaches is that they use in-house high-level simulators to gather the performance metrics. Of course, the authors have tried to develop accurate simulators, but it is impossible to know if they are 100% accurate since they have to model not only the accelerators, but also communications, and memory accesses. In our case, instead of using a simulator, we have implemented our design and we have measured our performance metrics during actual executions to guarantee that they are completely accurate.

III. Contributions

The main contributions of our work are as follows.

1) We have designed a sparse architecture that is able to avoid all useless operations and manage filter compression both for convolutional and fully connected layers. It also includes support to reduce the impact of the memory-bank conflicts due to the nonuniform memory access patterns. The design has been pipelined to improve the performance. Our architecture has been designed for embedded systems, and its objective is to maximize performance and reduce the energy consumption on systems with limited resources. It has been written in VHDL and is available for the community in a GitLab repository [30].
2) We propose a dense/sparse evaluation methodology that attempts to compare architectures with similar area resources. To this end, we have designed a dense architecture with parameterizable arithmetic resources and for each comparison we select the dense architecture most similar in area to the sparse design.
3) We have implemented both designs on an FPGA and taken performance and power consumption measurements. With this approach, we can identify the tradeoffs between sparse and dense architectures, and identify which architecture is better for a given scenario.

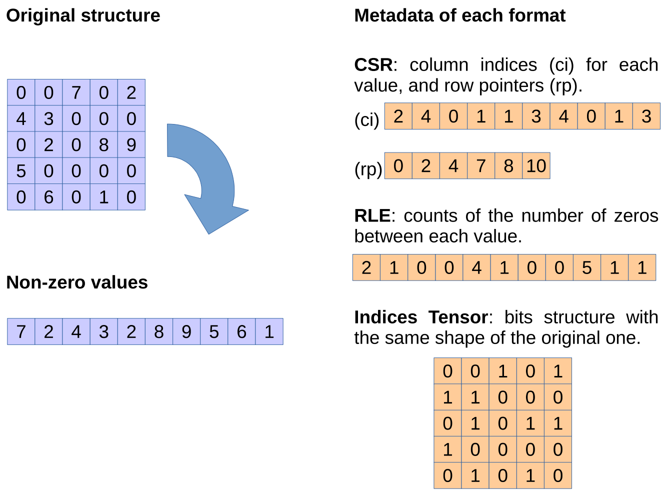


Fig. 1. Compression formats example.

IV. Data Compression

Some popular compression formats for sparse CNNs are run length encoding (RLE), compressed sparse row (CSR) or CSC [31]. The main idea of RLE is to store consecutive elements of the same value as a single value and the count of the repetitions, whereas CSC and CSR consist of two data sets: one that stores only those values that are not zero, and one that stores metadata to infer the remaining information and to calculate the addresses.

Instead of these formats, we propose to use a tensor with the same dimensions than the uncompressed data that store a single bit per element pointing out whether they are zero or not. Fig. 1 shows an example of these formats applied to matrices. For the comparison, we used the variation of RLE format proposed in [28], using 5 bits to codify the count of consecutive zeros. Regarding the CSC/CSR formats, they are symmetric structures, and using one or the other will be better depending on the selected matrix representation. In our case, CSR, using column indices and row pointers, is the one that reaches better results. Column indices store the column of each nonzero value, and row pointers point out the first nonzero value of each row. Its last value is the total number of nonzero elements.

Fig. 2 shows how these compression formats perform as a function of the sparsity. It includes two boundaries for each format. Upper and lower bound in CSC/CSR are calculated on matrices of 9×512 with 8-bit values and 1×16 with 32-bit values, respectively. We have chosen the indices tensor format for two reasons: first, it allows hardware-friendly identification of those operations that are useful (further discussed in Section V). Second, it yields higher compression ratios than the other formats for most scenarios and, in those where it under-performs, size becomes negligible.

In our architecture, filters are compressed in order to reduce the size of the models. It is also possible to compress the activation during inference, but it demands additional resources, as it is generated on the fly. Moreover, since the compression ratio is unknown at design time, memory resources should be allocated for the worst case, so the benefits are limited and

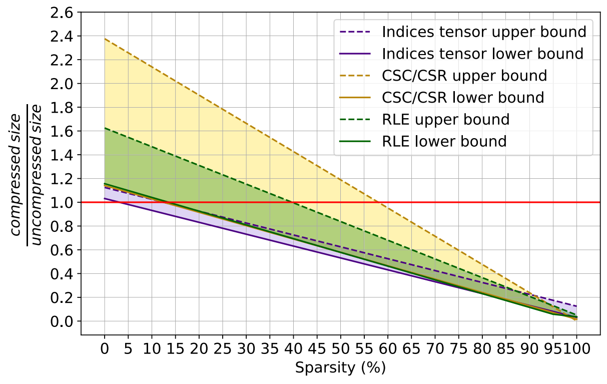


Fig. 2. Compression ratio comparisons.

do not justify the overhead. Our design leverages the indices tensor also for the activation in order to take full advantage of this compression format to identify useful operations, at the expense of a small overhead.

V. Identifying Useful Operations

Our design avoids any useless operation in order to reduce memory accesses and arithmetic computation. An efficient identification of those useful operations relies on the compression format discussed previously. First, sections of the indices tensor of the filter and the activation are fetched. Second, they are matched through a bitwise and operation to find all the pairs with two nonzero values. The main steps of this process are described in Algorithm 1. Activation_offset stores the position in the section where the current pair has been found. This information is used to calculate the address of the activation value needed. Filter_offset indicates the number of nonzero filter values that have been skipped since the last pair found. It is required to calculate the address of the filter value as filters are compressed and only nonzero values are stored. Finally, when the last pair of the fetched sections is processed, remaining_filter_offset keeps track of the number of remaining nonzero filter values in the section. This number must be taken into account when computing the next filter address.

Fig. 3 illustrates this process with a toy example. The algorithm iteratively looks for pairs of nonzero values located in the same section position. Notice that these nonzero values are marked as 1. In the first iteration, the first pair is found on the section element #3. Hence, activation_offset is 3. Filter_offset is 1 since one nonzero filter value has been skipped. Finally, processed elements (from #0 to #3) are masked. In the second iteration, the same procedure applies to the unmasked elements. Additionally, the last pair has been found, so remaining_filter_offset reports that an additional filter value must be skipped.

Fig. 4 depicts the hardware unit responsible for these operations. Bitwise unit carries out bitwise and operations on the filter and activation sections, and the masks generated by mask composer. They are both required to identify useful operations and compute the filter offsets. The priority encoder encodes the activation offset, i.e., the position where the current pair has been found. Finally, two tree adders return the filter offsets.

Algorithm 1 Identification of Useful Operations Within a Section

```
1:  fetch filter_section
2:  fetch activation_section
3:  initialize mask
4:  find_next_pair
5:  **if** pairs $\neq \emptyset$ **then**
6:      **while** unprocessed_pairs $\neq \emptyset$ **do**
7:          filter_offset $\leftarrow$ non-zero filter values skipped
8:          activation_offset $\leftarrow$ pair offset
9:          update mask
10:         **if** last_pair **then**
11:             remaining_filter_offset $\leftarrow$ remaining non-zero filter values
12:         **else**
13:             find_next_pair
14:         **end if**
15:     **end while**
16: **else**
17:     activation_offset $\leftarrow$ section_width
18:     remaining_filter_offset $\leftarrow$ remaining non-zero filter values
19: **end if**
```

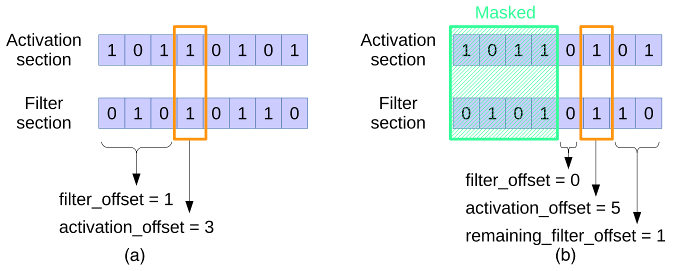


Fig. 3. Identification of useful operations. (a) Step one. (b) Step two.

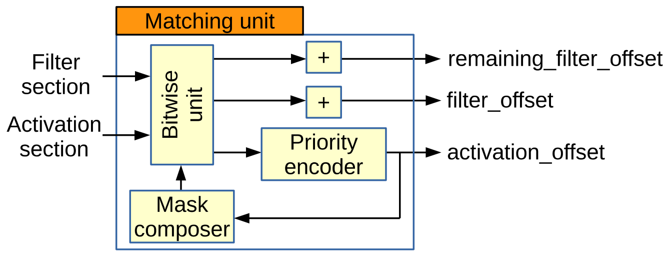


Fig. 4. Matching unit.

VI. Baseline: Dense Architecture

Assessing the benefits of exploiting sparsity in CNNs requires a baseline to compare with. We designed a dense accelerator for embedded systems able to exploit interfilter parallelism, through N processing units (PUs) computing N different filters, and intrafilter parallelism, through M multipliers per PU. It also includes support for filter compression according to the format discussed in section IV. Both input and output activations are stored on-chip through the whole inference process in order to reduce DRAM off-chip accesses, and filters are retrieved from DRAM on demand with a prefetch policy in order to hide fetch latency.

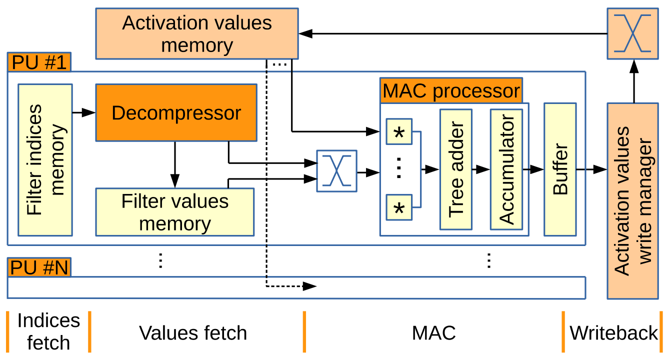


Fig. 5. Dense architecture.

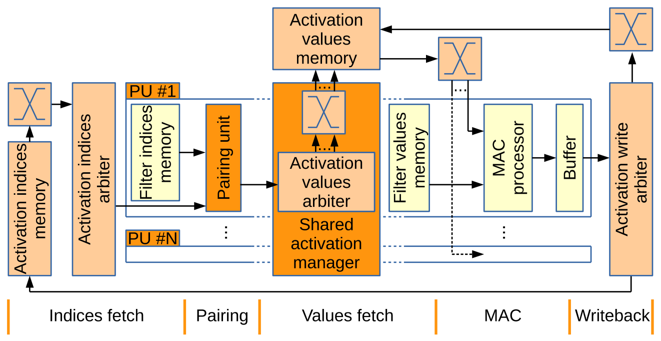


Fig. 6. Sparse architecture.

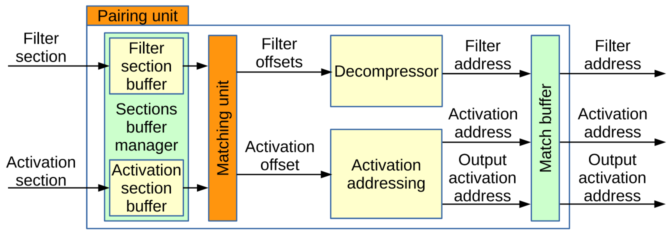


Fig. 7. Pairing unit.

Fig. 5 depicts the architecture of our dense accelerator. Each PU includes its own filter management and an MAC processor. The filter management includes support to decompress filters. The MAC processor is composed of a multiplier array, a tree adder to reduce the multipliers output, and an accumulator. Activation values memory is composed of two memories. When processing even layers one stores the input activation and the other one stores the output activation. When processing odd layers they swap their roles. This memory is shared among all the PUs, and there are no conflicts because all the PUs read the same data. In the figure, the pipeline is divided into four stages. These stages can also be internally pipelined in order to increase clock frequency if needed.

A. Stage 1: Indices Fetch

Filter indices for the next M operations are fetched at this stage. These indices are used at the next stage to fetch filter values.

B. Stage 2: Values Fetch

Filter and activation values are fetched at this stage. Activation values are fetched by a global controller as they are shared among all the PUs. Filter values demand specific addressing for each PU because of compression. Decompressor is the module responsible for addressing the filter memory based on the filter indices.

C. Stage 3: MAC

Operations are performed and buffered at this stage. Filter and activation values are retrieved, the MAC operation is performed on MAC processor, and the value is stored in a small buffer to avoid pipeline stalls because of writings. MAC processor is able to carry out M multiplications in parallel and reduce them in a single cycle through a tree adder.

D. Stage 4: Writeback

Output activation values are stored in the output activation memory. Activation values memory is parameterized with M banks, therefore, arbitration is required for those setups where $N > M$. We implemented a fixed-priority arbitration on each bank in order to keep hardware overhead as low as possible because pressure on this memory is very low since there are many computations between two consecutive writings.

VII. Sparse Architecture

Based on our dense design, we made architectural changes in order to include support for sparsity. Fig. 6 shows an architectural overview of our sparse design, which is divided into five stages. As in the dense architecture, each stage can be internally pipelined to increase clock frequency.

A. Stage 1: Indices Fetch

Filter and activation sections are fetched at this stage. Fetching filter indices is straightforward as they are stored in private memories. Activation indices are stored in a shared memory, and, therefore, access conflicts among PUs may arise. We included a multibank Activation indices memory, and fixed-priority arbitration for each bank. Fetching activation indices may become a bottleneck when dealing with very high sparsity ratios: the more sparsity the faster sections are processed, increasing pressure on memory. We found that including as many banks as PUs, in conjunction with a section buffer used to prefetch the next section in advance, causes minimal stalls while keeping hardware overhead low.

B. Stage 2: Pairing

Useful operations are identified and buffered at this stage following the procedure explained in Section V. Fig. 7 shows the architecture of Pairing unit. Sections buffer manager stores the sections under processing and the next ones to be processed. Matching unit processes both filter and activation sections and returns filter and activation offsets to compute their values addresses. Decompressor and activation addressing are responsible for computing the absolute memory addresses of filter and activation values, respectively. Finally, match buffer stores these addresses. This buffer is also useful to reduce stalls on accesses to activation values memory. This is further discussed in the next stage.

TABLE I
AVERAGE CONFLICT RATIO FOR SEVERAL CONFIGURATIONS

Requests	Banks	Conflict ratio
1x	1x	34%
1x	2x	19.5%
2x	1x	16%
2x	2x	2%
4x	1x	6%
4x	2x	0.1%

C. Stage 3: Values Fetch

Filter and activation values are fetched at this stage. Managing accesses to the shared activation memory in our dense design is straightforward, as every PU requests the same values at a given time. However, this no longer applies to our sparse design since accessing only those values that are not zero turn regular accesses into irregular ones, and therefore arbitration is required.

Activation values memory suffers from higher pressure than activation indices memory. Hence, we designed a more powerful arbitration scheme in order to prevent these accesses to become a bottleneck. This arbiter is able to explore several requests from each PU. These requests are stored in the match buffer. We have included support to process the requests out of order (multiplications can be safely reordered within a convolution step), making grants more likely at the expense of a low hardware overhead. Activation values arbiter grants each PU one of the requests, if possible, in a fixed order from PU #1 to #N, where PU #1 has the highest priority.

Fig. 8 illustrates how our arbitration works on a toy example with four PUs. Each PU requests two addresses (i.e., the depth of the match buffer is two). The activation memory is composed of four memory banks, so this memory supports up to four simultaneous accesses as long as they target different banks. The arbiter receives each pair of bank requests from each PU and grants one of them if possible. Thus, PU #1 is granted its first request (Bank #3). As a consequence, Bank #3 is masked for the remaining PUs. PU #2 requests Banks #3 and #1. Since Bank #3 is not available, the arbiter grants its second request (Bank #1). This procedure is repeated for the remaining PUs as shown in the figure.

In this example, the four PUs are granted, but this is not always possible. We empirically searched for the best trade-off between performance and hardware overhead exploring different configurations between the number of banks of the Activation values memory and the Match buffer size. As can be observed in Table I, we found that providing the Activation values memory with twice as banks as PUs, and setting the depth of the Match buffer to four, i.e., the arbiter explores up to four requests from each PU, memory access conflicts rarely occur.

As an additional optimization, we have included support to overlap the end of a convolution step with the beginning of the next one. While the requests corresponding to the last multiplications of the current convolution step are waiting to be granted, it is possible to store requests of the first multiplications of the next convolution step. Hence, when

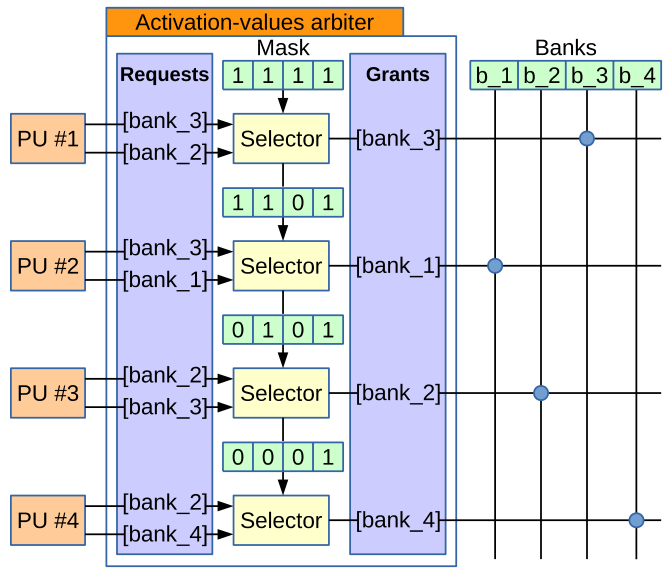


Fig. 8. Activation values arbiter workflow.

a convolution step finishes, the next one can have several requests ready for selection. This approach minimizes the memory accesses conflicts.

D. Stage 4: MAC

Useful operations are performed and buffered at this stage. Filter and activation values are retrieved, the MAC operation is performed on a MAC processor, and the value is stored in a small buffer to avoid pipeline stalls because of writings.

E. Stage 5: Writeback

This stage is similar to the corresponding stage in the dense architecture.

VIII. MEMORY REQUIREMENTS AND DATAFLOW

The size of the memories in our design is parameterizable, so it can be adjusted to the needs of each DNN. The memory hierarchy of our design includes the following storage elements.

1) *Off-Chip Memory:* It stores all the filters of the network and the image/s to be processed.
2) *On-Chip Memory:* It stores the input and output activations and the filters under processing in the following memories.
   a) *Filters Memory:* Each PU includes private resources to store both the filter for the ongoing convolution and the next one, which is preloaded to hide the fetch latency. As depicted in Figs. 5 and 6, the filter values and their indices are stored in two different memories, the Filter values memory and the Filter indices memory, respectively.
   b) *Activation Memory:* The Activation values memory and the Activation indices memory shown in Fig. 6 store the activation values and their indices in two shared memories. Both of them are multi-banked and are accessed through arbiters to prevent

conflicts. This memory element contains both the input and the output activation of the layer under processing.

Regarding the dataflow: first, the image is loaded into the activation memory, and one filter is loaded into the private filter memories of each PU. Once the inference begins, each PU will exploit reuse by convolving the filter across the whole input activation. This scheme reduces bandwidth with off-chip memory as filters are retrieved only once per inference. While convolving a filter, another one is retrieved from the off-chip memory in order to hide fetch latency. Each PU stores its ongoing convolution partial results in an accumulator register. No data is shared among PUs. Once the output of a layer has been stored, it becomes the input activation of the next layer.

IX. Scalability

Inference on CNNs is a parallel-friendly task. Many filters are convolved in each layer, and many operations are performed in each filter. Both interfilter and intrafilter parallelism are free of data dependencies, making it suitable for a custom hardware architecture to exploit it. This is indeed what we do in our dense architecture, where adding PUs exploits interfilter parallelism, and adding multipliers to each PU exploits intrafilter parallelism.

When exploiting sparsity, we have to deal with conflicts in the access to the activation data memory, which requires hardware hard to scale. Our design needs crossbars to access multibanked activations memories, and scaling them causes the performance-area tradeoff to plummet. Hence, our sparse architecture must be small in order to be efficient. For that reason, it is more suitable for embedded systems. For other contexts, like high resolution images, there are very good massively parallel architectures with hundreds of PEs, such as SCNN [27] and EyerissV2 [29], whose high throughput fits the needs of these problems.

Nonetheless, there are contexts where it is still possible to scale the design by including several instances of our architecture (i.e., cores) working in parallel on their own private activation memories. For example, if several images must be processed, they can be assigned to each of those cores. It is also possible to assign different regions of the activation to each core. In this case, there will be a small overlap among the activations, and therefore some additional control hardware would be necessary to share these overlapped data between cores.

X. Experimental Results

We implemented our sparse and dense designs on a Xilinx Zynq UltraScale+ ZCU104 evaluation board [32]. This platform includes a SoC with an FPGA tightly coupled with a CPU, a real-time processor, and a GPU. In our experiments, only the FPGA and the CPU were used. The FPGA, which hosts our accelerators, performs all the computations, and the CPU just manages the communications with the off-chip memory.

Power consumption measurements were taken with a Yokogawa WT210 digital power meter, a device accepted by standard performance evaluation corporation [33]. Our power meter records the total consumption of the evaluation board, which includes many unused elements. Hence, we have removed the static power consumption from our measures, and we have focused on the dynamic power consumed by our design, i.e., the average difference of power consumed by the board during idle and running states. To do this, we have measured the energy consumption of our designs for one hour in each state.

TABLE II
Experimental Setups With Eight PUs

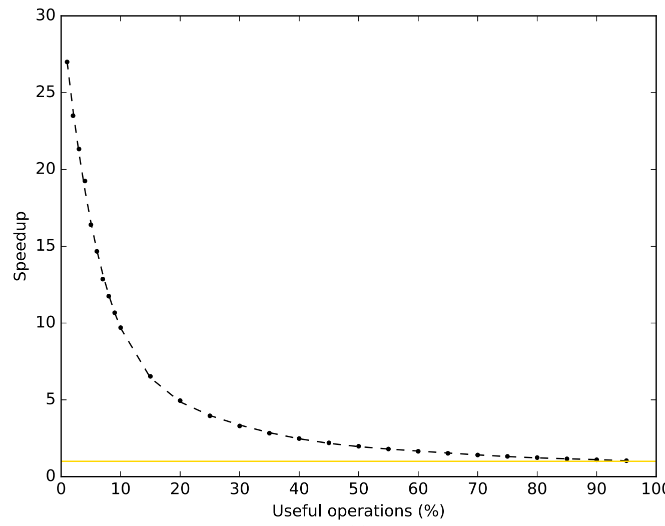
8-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	5.94	124 MHz	131 mW
dense	1	2.04	123 MHz	39 mW
dense eq.	8	5.41	123 MHz	206 mW
16-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	7.94	110 MHz	247 mW
dense	1	3.80	112 MHz	87 mW
dense eq.	4	7.58	115 MHz	472 mW
32-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	12.32	106 MHz	842 mW
dense	1	8.54	101 MHz	462 mW
dense eq.	2	13.02	101 MHz	1104 mW

The purpose of our experiments is to characterize the behavior of our designs in terms of performance, hardware resources, and energy efficiency, as a function of the useful operations (which depends directly on the sparsity) and the arithmetic bitwidth. To analyze the impact of the sparsity, we developed a set of synthetic benchmarks with a useful operations ratio ranging from 0 to 1, i.e., we started with a scenario where each multiplication include a zero as one of its operands, and we progressively reduced the number of zeros until reaching the opposite scenario, where all the operands are different from zero. To study the impact of the bitwidth we implemented three different versions of each design using 8, 16, and 32-bit fixed-point arithmetic. All the results in the figures of this section are normalized to the baseline selected in each case study.

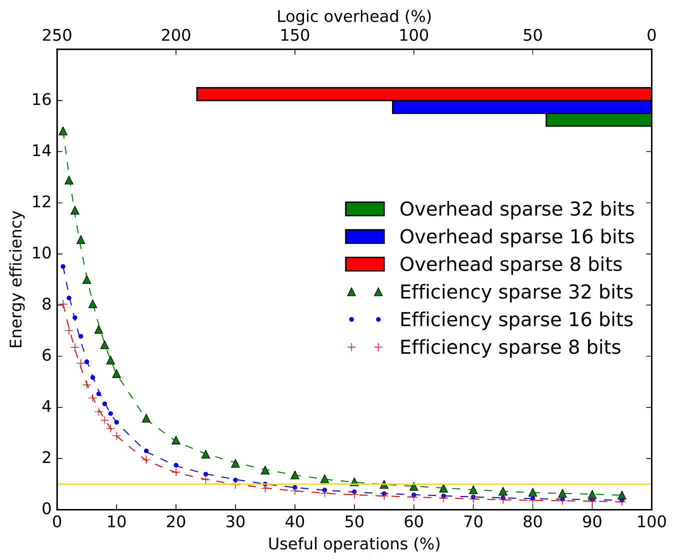
Our experimental setup is divided into three model designs.

a) *Sparse:* Sparse architecture with one multiplier per PU.
b) *Dense Base:* Dense architecture with one multiplier per PU.
c) *Dense Equivalent:* Dense architecture where the number of multipliers per PU is selected in such a way that the dense design is as similar as possible in area to the sparse design.

Table II details the design parameters, hardware resources utilization, maximum frequency, and the dynamic power consumption of each setup. In our experiments all the setups were clocked to 100 MHz. FPGAs include DSP blocks that can be used to execute MACs, and synthesis tools map these operations into them whereas the remaining functionality is implemented using look-up tables (LUTs). This makes impossible to compare ones with each other, as MACs are implemented with a different technology than the rest of the

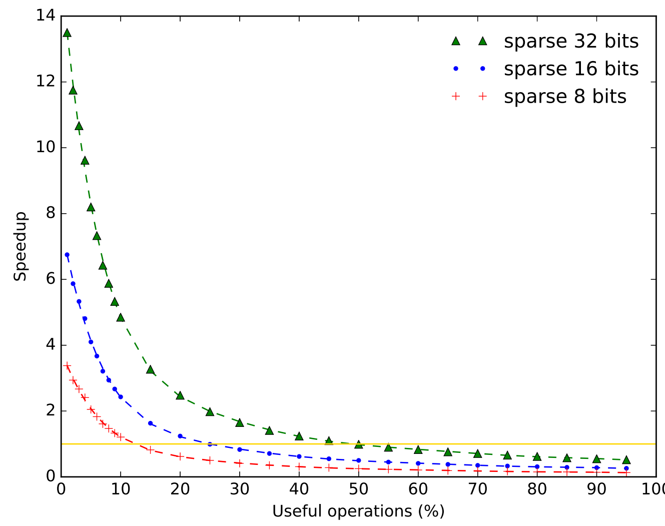


(a)

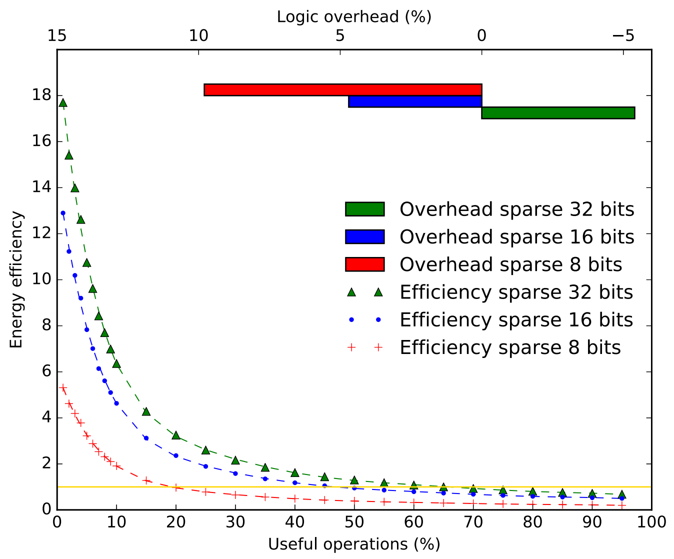


(b)

Fig. 9. Speedup and energy efficiency of sparse experimental setups normalized to dense base. (a) Speedup. (b) Energy efficiency.



(a)



(b)

Fig. 10. Speedup and energy efficiency of sparse experimental setups normalized to dense equivalent. (a) Speedup. (b) Energy efficiency.

logic. Hence, we disabled DSP mapping in order to map also the MAC processors into LUTs.

We first compared sparse and dense base models, which include the same arithmetic resources (number of PUs, and multipliers per PU). Hence, both designs exploit the same degree of parallelism. We want to assess the benefits in terms of performance and energy efficiency of including support for sparsity, and quantify the hardware overhead introduced.

Fig. 9(a) depicts the speedup as a function of the useful operations. As all the setups are clocked to 100 MHz, the speedup grows in inverse proportion to the percentage of useful operations since the reason for this speedup is the number of operations avoided, therefore the three different arithmetics analyzed (8, 16, and 32 bits) achieve the same speedup.

Fig. 9(b) depicts the energy efficiency as a function of the useful operations. In the top of the figure the overhead in terms of logic resources is presented. Unlike speedup, the energy efficiency and the area overhead depend on the arithmetic bitwidth. The reason is that arithmetic computations require less logic resources and consume less energy for low bitwidth, whereas the area and energy consumption due to the support included for the sparsity remains the same.

Gains in performance and energy efficiency are remarkable for highly sparse scenarios. However, achieving these results involves a logic overhead ranging from 50% to almost 200%. Hence, the question is: what if we provide our dense design with similar hardware resources? Comparison between sparse and dense equivalent models answers this question. Setups in dense equivalent design balance hardware resources by including more multipliers for each PU. Scaling by exploiting

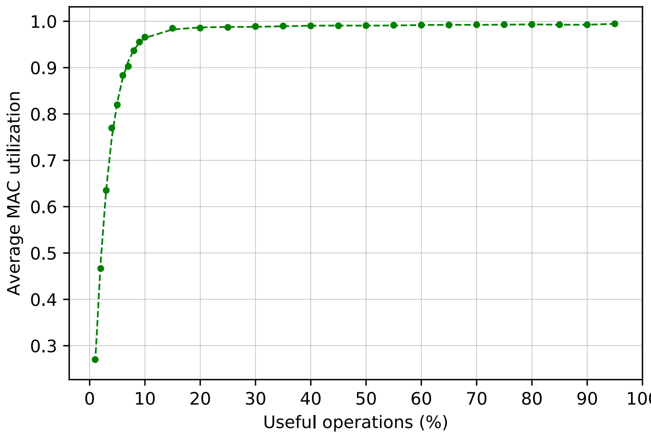


Fig. 11. Average MAC utilization.

the intrafilter parallelism is feasible due to the size of the filters in actual CNNs, and demands little overhead (tree adders to reduce the multiplications, and a little more complex addressing control). Our objective is to identify whether it is worthy to move from a dense to a sparse architecture for a given scenario. For that, we are going to compare our sparse design with a dense design with similar hardware resources, i.e., similar area. As can be seen in Table II, for 32-bit arithmetic the dense equivalent version includes twice the number of multipliers than the sparse version, for 16-bit it includes 4× multipliers, and for 8-bit it includes 8×. These differences are due to the different sizes of the multipliers for each arithmetic bitwidth.

Results in terms of performance are shown in Fig. 10(a). The results show that the benefits of providing support for sparsity have decreased. Our sparse design working on 8-bit arithmetic is only worthy when the useful operations are below 10%. On 16-bit, the threshold grows up to 25%, and on 32-bits, the threshold is at 50%. Numbers of energy efficiency are more favorable to the sparse architecture [see Fig. 10(b)]. The benefits show up when the useful operations are below 20%, 60%, and 70%, respectively. Hence, for low-precision arithmetic, it is only profitable to include support for sparsity in aggressively pruned models.

One of the goals of our design is to maximize the utilization of arithmetic resources. Fig. 11 shows that MAC utilization is virtually 100% even for networks with a very low useful operations ratio. It only plummets when the useful operations is under ∼5% because, in that situation, frequently the 32-bit matching unit cannot find any useful operation in the fetched sections. Even so, this is not a undesirable scenario for our architecture. In fact, when no useful operations are found our architecture is indeed skipping 32 operations in one cycle.

A. AlexNet and SqueezeNet

The previous results have been obtained using synthetic data generated randomly for a given value for a given percentage of useful operations. However, we also wanted to try our accelerator with representative pruned models and data sets. Although many works have demonstrated the possibilities of pruning, deep learning frameworks still do not include support in their main branches. However, Han *et al.* [4] shared two pruned models, AlexNet and SqueezeNet, on GitHub [10], [11]. These are very good benchmarks for our design since they are popular models that use the representative data set ImageNet [34], and SqueezeNet is especially interesting for Embedded Systems. Tables III and IV detail the useful operations ratio per layer of these two pruned networks and the MAC utilization of our sparse design for each layer. Overall, the useful operations ratio on AlexNet and SqueezeNet are 18.4% and 32.0%, respectively.

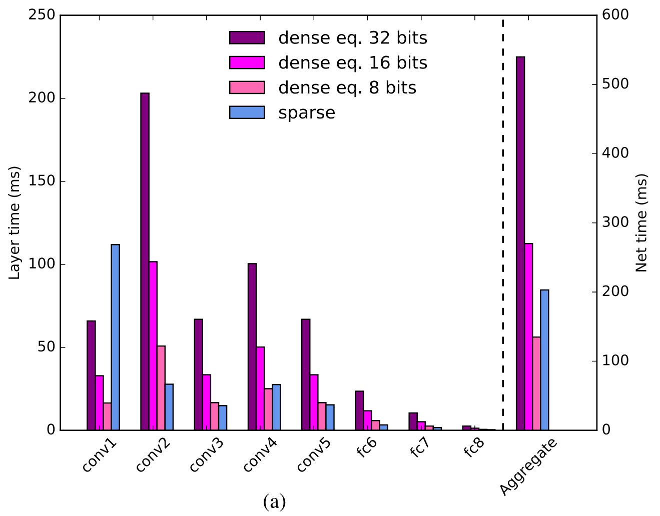
TABLE III
USEFUL OPERATIONS PER LAYER IN ALEXNET

	Useful operations	MAC utilization
conv1	84.3%	99.3%
conv2	6.7%	98.9%
conv3	10.9%	98.2%
conv4	13.5%	98.4%
conv5	11.3%	97.6%
fc6	3.6%	51.9%
fc7	5.8%	70.3%
fc8	7.6%	90.8%

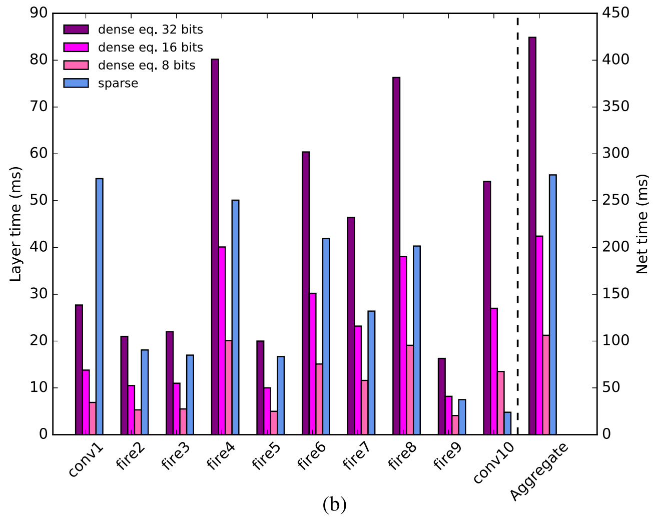
TABLE IV
USEFUL OPERATIONS AND PERCENTAGE OF MAC UTILIZATION PER LAYER IN SQUEEZENET

	Useful operations	MAC utilization
conv1	98.5%	99.6%
fire2	41.2%	95.5%
fire3	37.4%	96.8%
fire4	30.5%	97.8%
fire5	40.9%	98.1%
fire6	34.0%	98.1%
fire7	28.0%	98.2%
fire8	25.8%	97.9%
fire9	26.6%	98.5%
conv10	2.3%	51.9%

Fig. 12(a) and (b) depicts the execution time per layer of AlexNet and SqueezeNet, respectively. We have compared our sparse design working with 8-bit, 16-bit, and 32-bit arithmetic with their area equivalent dense designs, i.e., dense equivalent model with 8×, 4×, and 2× multipliers, respectively. Results vary among layers because of their different useful operations ratios, as shown in Tables III and IV. When executing AlexNet, our sparse design outperforms its dense equivalent by 2.66× on 32-bit arithmetic and 1.33× on 16-bit. On 8-bit, the dense equivalent design with 8× multipliers is superior by 1.50×. When executing SqueezeNet, our sparse design outperforms its dense equivalent by 1.53× on 32-bit arithmetic whereas the dense equivalent design is superior on 16- and 8-bit by 1.31× and 2.61×, respectively. Given these execution times, our accelerator is able to process 227 × 227 images in 203 ms for AlexNet and in 265.2 ms for SqueezeNet, yielding a throughput of 3.8 and 4.9 images/s, respectively. These numbers are suitable for many embedded applications. They may not look impressive at a first glance, but, in fact, they are very close to the peak performance. For instance, processing an image with the pruned version of AlexNet [11] involves 863 740 448 multiplications. Since most of them include a zero as an operand, with our approach that the number can be reduced to just 159 031 554 useful multiplications.



(a)



(b)

Fig. 12. Execution time on AlexNet and SqueezeNet. (a) Execution time on AlexNet. (b) Execution time on SqueezeNet.

The minimum execution time using eight multipliers clocked at 100 MHz and assuming peak performance is 198.79 ms, just a bit lower than the 203 ms that our sparse architecture needs.

XI. Conclusion

We propose a sparse architecture for DNNs that avoids those operations with zero as one of the operands and keeps almost peak utilization of arithmetic resources, even in highly sparse scenarios. The architecture includes support to deal with compressed filters, identify the useful operations, and reduce the memory access conflicts generated due to the nonuniform memory accesses. It has also been pipelined to improve the performance.

We have carried out comparisons between similar-in-area dense and sparse architectures in order to identify in which scenarios including support for sparsity is superior to providing a dense architecture with additional arithmetic resources. Sparsity is, as expected, the key parameter to decide whether to move from dense to sparse architectures, but arithmetic bitwidth also plays a major role. The hardware cost of MAC units does not scale linearly with the bitwidth; therefore, the overhead ratio of a sparse architecture is much larger on low precision. Our results show that the benefits of exploiting sparsity are clear on 32-bit arithmetic, whereas on 8-bit it is hard to profitably exploit sparsity given the sparsity of current state-of-the-art CNNs. Recent works show consensus on using arithmetic of at least 16-bit [21], [22], [25]–[27]. For this particular arithmetic bitwidth, adding support for sparsity improves energy efficiency as long as the useful operations are under 50%, and also performance, when the useful operations are under 25%. In other scenarios it is better to use the logic resources to include more arithmetic resources than to include support for sparsity.

We consider that FPGAs are currently the natural target for sparse accelerators, instead of ASICs as suggested in previous works. The benefits of including support for sparsity depend on the particular sparsity and arithmetic precision of the DNN to process, and FPGAs can be seamlessly adapted by loading the best-fitting accelerator for each profile.

Developing pruning techniques for DNNs is currently a very active research topic and we expect that many aggressively pruned models are likely to be available soon. The inclusion of support for sparsity will be needed in order to take advantage of this powerful optimization opportunity.

References

[1] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," 2016, *arXiv:1604.03168*. [Online]. Available: http://arxiv.org/abs/1604.03168

[2] 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," 2017, *arXiv:1711.04713*. [Online]. Available: http://arxiv.org/abs/1711.04713

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.

[5] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016, *arXiv:1612.01064*. [Online]. Available: http://arxiv.org/abs/1612.01064

[6] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*. [Online]. Available: http://arxiv.org/abs/1702.03044

[7] A. Mirzaeian, H. Homayoun, and A. Sasan, "NESTA: Hamming weight compression-based neural Proc. Engine," 2019, *arXiv:1910.00700*. [Online]. Available: http://arxiv.org/abs/1910.00700

[8] A. Mirzaeian, H. Homayoun, and A. Sasan, "TCD-NPE: A reconfigurable and efficient neural processing engine, powered by novel temporal-carry-deferring MACs," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2019, pp. 1–8.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: http://arxiv.org/abs/1602.07360

[10] S. Han. (2016). *Squeezenet Deep Compression—GitHub Repository*. Accessed: Dec. 10, 2018. [Online]. Available: https://github.com/songhan/SqueezeNet-Deep-Compression

[11] S. Han. (2016). *Deep Compression AlexNet—Github Repository*. Accessed: Dec. 10, 2018. [Online]. Available: https://github.com/songhan/Deep-Compression-AlexNet

[12] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, vol. 32, pp. 1109–1139, Oct. 2018.

[13] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, Jun. 2018.

[14] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, D. S. Touretzky, Ed. San Mateo, CA, USA: Morgan Kaufmann, 1990, pp. 598–605. [Online]. Available: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[15] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Proc. Adv. Neural Inf. Process. Syst.*, S. J. Hanson, J. D. Cowan, and C. L. Giles, Eds. San Mateo, CA, USA: Morgan Kaufmann, 1993, pp. 164–171. [Online]. Available: http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.pdf

[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2015, pp. 1135–1143, pp. 1–9. [Online]. Available: http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf

[17] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5687–5695.

[18] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2017, *arXiv:1710.01878*. [Online]. Available: http://arxiv.org/abs/1710.01878

[19] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 548–560.

[20] S. Cao *et al.*, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. ACM/SIGDA ISFPGA*. New York, NY, USA: Association Computing Machinery, 2019, pp. 63–72.

[21] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.

[22] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254.

[23] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*. Piscataway, NJ, USA: IEEE Press, Jun. 2018, pp. 674–687.

[24] J. Li *et al.*, "SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules," *IEEE Trans. Comput.*, vol. 68, no. 11, pp. 1663–1677, Nov. 2019.

[25] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

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

[27] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.

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

[29] Y.-H. Chen, T.-J. Yang, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

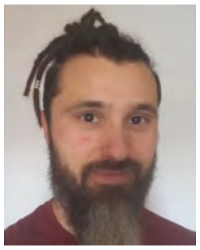
[30] A. Alcolea, J. Olivito, and J. Resano. (2018). *Pipelined Architecture for Sparse DNNs*. [Online]. Available: https://gitlab.unizar.es/jolivito/pipelined_architecture_for_sparse_DNNs

[31] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, Univ. California, Oakland, CA, USA.

[32] Xilinx. *Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit, SoC and FPGA Platform*. Accessed: Jun. 2020. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/zcu104.html

[33] Yokogawa. *Wt210/Wt230 Digital Power Meters*. Accessed: Jun. 2020. [Online]. Available: https://tmi.yokogawa.com/solutions/products/power-analyzers/wt210wt230-digital-power-meters

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255. [Online]. Available: http://www.image-net.org



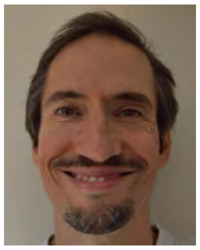
Adrián Alcolea Moreno graduated in social work and computer engineering from the University of Zaragoza, Zaragoza, Spain, in 2010 and 2017, respectively. He received the master's degree in international peace, conflict and development studies from Jaume I University, Castellón de la Plana, Spain, in 2012. He is currently working toward the Ph.D. degree at the Computer Architecture Research Group, University of Zaragoza.

He is working with the Computer Architecture Research Group, University of Zaragoza, on the development of hardware support for deep neural networks.

Javier Olivito received the M.S. and Ph.D. degrees in computer engineering from the University of Zaragoza, Zaragoza, Spain, in 2011 and 2017, respectively.

He is currently an FPGA Engineer with the Research and Development Department, Telnet Redes Inteligentes, Zaragoza. His research interests include hardware/software co-design, acceleration of artificial intelligence algorithms, and dynamic reconfiguration.

Dr. Olivito FPGA designs have received several international awards, including the First Prize at the Design Competition of the IEEE International Conference on Field-Programmable Technology in 2009, 2010, and 2014.



Javier Resano received the Ph.D. degree in computer architecture from the Universidad Complutense de Madrid, Madrid, Spain, in 2005.

He is currently an Associate Professor of Computer Engineering with the University of Zaragoza, Zaragoza, Spain, where he is a member of the GaZ Research Group (Computer Architecture Group) and the Aragón Institute for Engineering Research (I3A). His research interests include hardware/software co-design, task scheduling techniques for heterogeneous multiprocessor systems, FPGA design, and efficient computing for remote-sensing systems using machine-learning techniques.

Dr. Resano has received several international awards for his digital-logic designs.

Hortensia Mecha was born in 1967. She received the degree in applied physics and the Ph.D. degree in physics from the Universidad Complutense de Madrid (UCM), Madrid, Spain, in 1990 and 1996, respectively.

She is currently a Professor of Computer Architecture and Technology with the Computer Architecture and Automation Department, UCM. She is also working with the GHADIR Research Group on dynamically reconfigurable architectures. Her research interest includes several aspects of the computer-aided design of integrated circuits, with a particular emphasis on automated synthesis, reconfigurable computing, and fault tolerance.