Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks

NEUROMORPHIC
Computing and Engineering

OPEN ACCESS

RECEIVED
24 December 2021

REVISED
20 June 2022

ACCEPTED FOR PUBLICATION
28 June 2022

PUBLISHED
14 July 2022

PAPER

Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks

Kevin Hunter, Lawrence Spracklen and Subutai Ahmad

Numenta, Redwood City, CA, United States of America
* Author to whom any correspondence should be addressed.

E-mail: khunter@numenta.com, lspracklen@numenta.com and sahmad@numenta.com

Keywords: sparsity, FPGA, deep learning, convolutional networks, neocortex, DNNs, ResNet

Supplementary material for this article is available online

Abstract

In principle, sparse neural networks should be significantly more efficient than traditional dense networks. Neurons in the brain exhibit two types of sparsity; they are sparsely interconnected and sparsely active. These two types of sparsity, called weight sparsity and activation sparsity, when combined, offer the potential to reduce the computational cost of neural networks by two orders of magnitude. Despite this potential, today's neural networks deliver only modest performance benefits using just weight sparsity, because traditional computing hardware cannot efficiently process sparse networks. In this article we introduce Complementary Sparsity, a novel technique that significantly improves the performance of dual sparse networks on existing hardware. We demonstrate that we can achieve high performance running weight-sparse networks, and we can multiply those speedups by incorporating activation sparsity. Using Complementary Sparsity, we show up to 100× improvement in throughput and energy efficiency performing inference on FPGAs. We analyze scalability and resource tradeoffs for a variety of kernels typical of commercial convolutional networks such as ResNet-50 and MobileNetV2. Our results with Complementary Sparsity suggest that weight plus activation sparsity can be a potent combination for efficiently scaling future AI models.

1. Introduction

In recent years, larger and more complex deep neural networks (DNNs) have led to significant advances in artificial intelligence (AI). However, the exponential growth of these models threatens forward progress. Training requires large numbers of GPUs or TPUs, and can take days or even weeks, resulting in large carbon footprints and spiraling cloud costs [69, 72]. Taking inspiration from neuroscience, sparsity has been proposed as a solution to this rapid growth in model size. In this article we demonstrate how to exploit sparsity to achieve two orders of magnitude performance improvements in deep learning systems.

Sparse networks either constrain the connectivity (weight sparsity) or activity (activation sparsity) of their neurons, significantly reducing both the size and computational complexity of the model. Typically, these techniques are applied in isolation to create *sparse–dense* networks. However, weight and activation sparsity are synergistic, and when deployed in combination, the computational savings are multiplicative. Consequently, *sparse–sparse* networks have the potential to reduce the computational complexity of the model by over two-orders of magnitude. For example, as illustrated in figure 1, when a network is 90% weight sparse, only one out of every ten weights is non-zero, facilitating a ten-fold reduction in compute. When a network is 90% activation sparse, only one out of every ten inputs is non-zero, similarly delivering a ten-fold reduction in compute. When applied in concert, the zero-values interplay, such that on average only one out of every 100 results will be non-zero, delivering a theoretical 100-fold savings.

However, with current implementations, the resulting speedups only represent a small fraction of these theoretical computational savings [21]. The irregular patterns of neuron interconnections and activity introduced by sparsity have proved difficult to exploit on modern hardware and impedes the implementation of

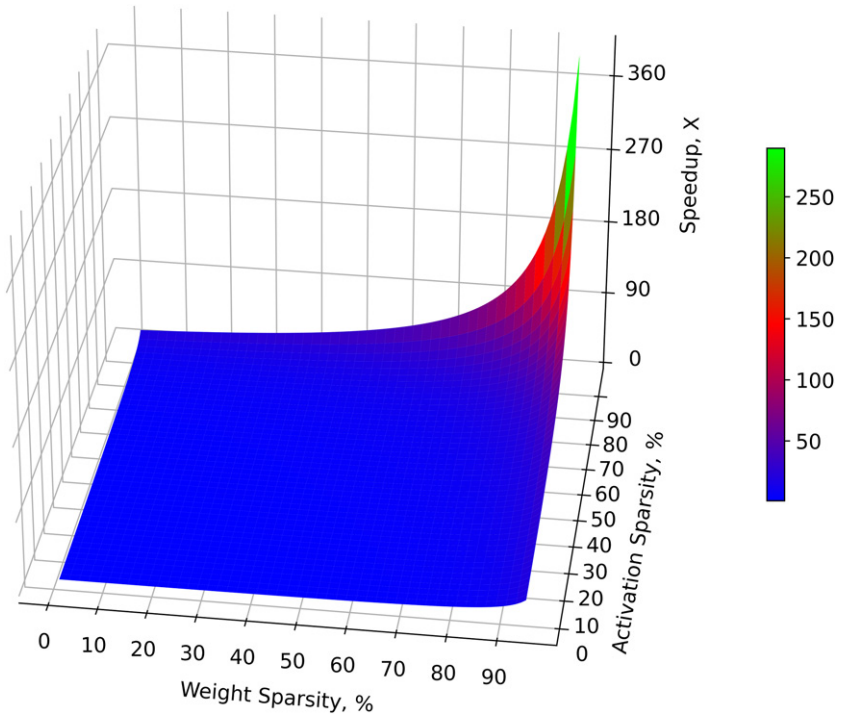


Figure 1. An illustration of the potential speedups that can be achieved with sparse networks (compared to their dense equivalents). When weight and activation sparsity are leveraged simultaneously (a sparse–sparse network), the benefits are multiplicative, enabling speedups that exceed two orders of magnitude.

efficient sparse–sparse networks. Hardware platforms with dedicated logic for exploiting sparsity have begun to appear [62], but the performance gains remain modest [52].

In this article we discuss Complementary Sparsity, a novel solution that inverts the sparsity problem. Rather than creating hardware to support unstructured sparse networks, we illustrate how sparsity can be structured to match the requirements of the target hardware. We demonstrate that this solution both creates highly efficient weight-sparse networks, and establishes viable sparse–sparse networks, yielding large multiplicative benefits.

We investigate the potential of Complementary Sparsity and sparse–sparse networks on FPGAs, due to their flexible architecture. This flexibility provides an ideal laboratory for investigating the trade-offs associated with different implementation approaches, and enables us to refine our understanding of sparse–sparse resource requirements. The resulting implementations not only provide a path to highly efficient sparse–sparse network inference on FPGAs, but also provide insights that can be leveraged as IP blocks in other architectures or ASICs, or adapted to fit a wide range of other compute architectures.

In this paper, we make four main contributions:

(a) We introduce Complementary Sparsity, a novel form of structured sparsity.
(b) We establish how Complementary Sparsity can enable the construction of efficient sparse–sparse networks.
(c) We discuss our sparse–sparse network implementation on a FPGA, demonstrating a 110× speedup over an optimized dense implementation.
(d) We demonstrate that leveraging activation sparsity reduces the hardware resource utilization associated with the core components of convolutional networks.

2. Sparsity in the brain, in deep learning, and in hardware

2.1. Sparsity in the brain

It is well known that the brain, specifically the neocortex, is highly sparse. This sparsity is instantiated a few different ways. First, the interconnectivity between neurons is sparse. Detailed anatomical studies show that cortical pyramidal neurons receive relatively few excitatory inputs from surrounding neurons [28, 50]. The percentage of local area connections appears to be less than 5% [28] compared to a fully connected dense network.

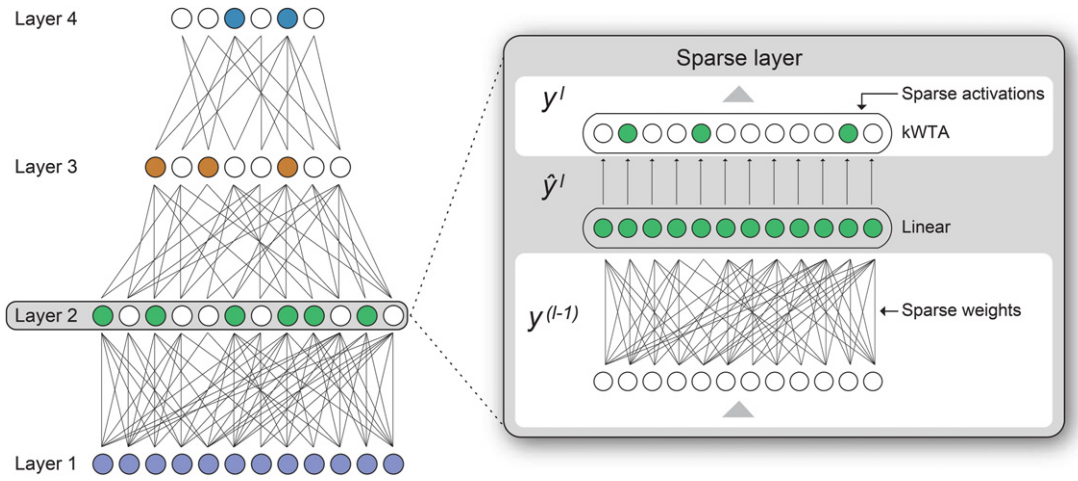


Figure 2. We focus on sparse networks with both sparse connectivity and sparse activations. Zeroes in the weight matrices enforce sparse connectivity. A *k*-WTA activation function ensures activations with fixed sparsity.

In addition to sparse connectivity, numerous studies show that only a small percentage of neurons become active in response to sensory stimuli [3, 6, 81]. On average less than 2% of neurons fire for any given input. This is true for all sensory modalities as well as areas that deal with language, abstract thought, planning, etc.

Recent experimental evidence suggests that local cortical networks are structured via specific networks of excitatory and inhibitory neurons [88, 93]. Inhibitory neurons are recurrently connected to excitatory neurons, encouraging competition that allows the most active neurons to 'win' [93]. These winner-take-all circuits are thought to give rise to sparse activations and are directly linked to the formation of sparse codes that match observed properties of V1 cells [37].

Sparsity leads to a number of useful properties. The brain is incredibly power efficient, a fact that has been directly linked to both activation sparsity [3, 43] and connection sparsity [60]. Sparsity has also been linked to the brain's ability to form useful representations [56, 57], make predictions [25, 51, 75], as well as detect surprise and anomalies. It seems evident that sparsity is ubiquitous in the neocortex and fundamental to its efficiency and functionality. Taking inspiration from these findings and their links to efficiency, in our implementation we employ both connection sparsity as well as activation sparsity through a competitive *k*-winner-take-all (*k*-WTA) circuit.

2.2. Sparsity in deep learning

The prevalence of sparsity in the brain stands in contrast to standard DNNs where sparsity is still a research area. The output of each layer in a DNN can be computed as a simple matrix multiplication; the inputs to the layer form either an activation vector (in the case of a single input), or a matrix (when a batch of inputs are being processed). In standard DNNs both the weight matrices and the activation vectors are dense.

Analogous to the neurology, it is possible to create two forms of sparsity in DNNs: sparse connections and sparse activations (figure 2). Absent connections are represented by 0's in the weight matrices, while inactive neurons are represented by 0's in the inputs to each layer. Recently, there has been an increase in research focused on creating networks that are both sparse and accurate. A variety of techniques have been proposed in the literature to achieve either form of sparsity, as summarized in the following sections. Note however that, unlike biology, it is rare for DNNs to combine both types of sparsity in the same network.

2.2.1. Weight sparsity

Research has shown that many DNNs are heavily overparameterized, and sparsity can be successfully applied to these networks [12, 42]. This technique of limiting neuron interconnectivity is referred to as *weight sparsity*.

As the sparsity of the weight matrices is increased the overall accuracy can drop. A variety of techniques have been developed to create networks that are both sparse and accurate [27, 53]. Most research has focused on the creation of: (a) sparse models by direct training; or, (b) sparse models from existing dense networks by removing (or 'pruning') the least important weights [27]. Within these two broad approaches exist a variety of different techniques, with varying degrees of sophistication and dynamism. Most simplistic are single-shot pruning algorithms [27], that remove all of the weights necessary to achieve the desired sparsity in one event. Iterative algorithms gradually increase the sparsity over the span of a

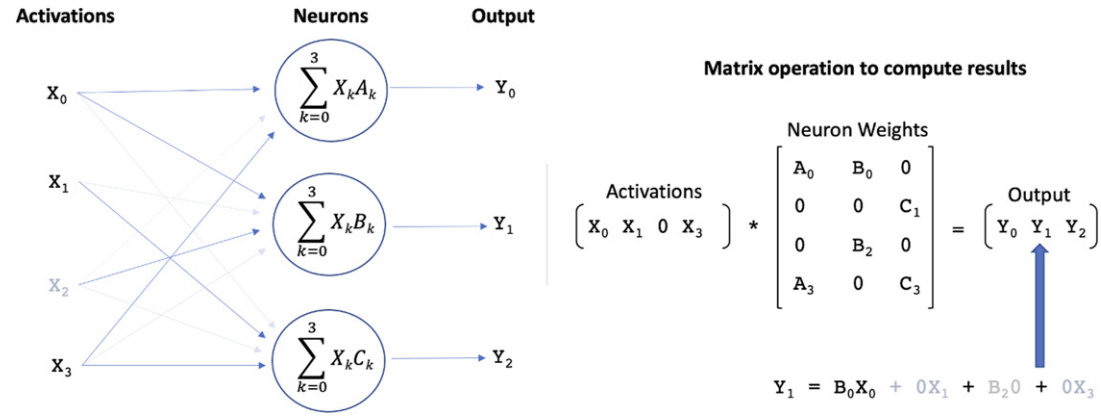


Figure 3. An illustration illustrating the simplification of matrix multiplication operations when both activations and weights are sparse. It is only necessary to compute a product if it contains a non-zero element in both the input activation *and* the weight matrix.

number of steps until the desired sparsity is achieved [27]. In addition to undertaking iterative pruning, algorithms can iteratively grow connections, working to ensure that the optimal set of interconnections is retained [17].

Pruning techniques primarily focused on reducing computational overheads are also in use [17, 53]. Different levels of sparsity in each layer allows sparsity to be focused in components of the model to deliver the most significant speedups, while smaller layers that only contribute minimally to the overall computational costs and parameter counts are protected. Novel pruning techniques and sparsity patterns that are tailored to hardware requirements have also focused on convolutional layers, where the multiplicity of channels provides opportunity for a variety of hardware friendly sparsity patterns [9].

2.2.2. Activation sparsity

In DNNs the outputs (activations) of each layer are generally dense, with between 50% to 100% of the neurons having non-zero activations. While less commonly discussed, activation sparsity can also be applied to DNNs [1, 40, 48, 59]. For activation sparsity, the determination of neurons to activate is typically performed by either explicitly selecting the top-k activations (frequently termed k-WTA) [1, 48] or by computing dataset specific activation thresholds for the neurons that, on average, reduce the number of activated neurons to the desired level [40]. It is also possible to compute sparse activations that are optimal from an information theoretic perspective by introducing regularizers or cost functions that penalize large values [42, 56, 67] and by performing locally iterative computations during inference [59, 64].

In this article we focus on networks using k-WTA [14, 46]. In these networks the ReLU activation function is replaced by an activation function where the output of each layer is constrained such that only the K most active neurons are allowed to be non-zero [1, 48]. Whereas ReLU allows all activations above 0 to propagate, k-WTA allows exactly the top K activations to propagate. From a hardware perspective, this is attractive because k-WTA can guarantee the exact sparsity level at each timestep. Although not optimal from a coding perspective, networks using k-WTA are surprisingly powerful and can provably approximate arbitrary non-linear functions [45]. In practice they are known to perform well on complex datasets [1, 47, 48].

2.3. Challenges of accelerating sparse networks

Removing weights and curtailing activations introduces zero-valued elements into the weight and activation matrices respectively. This reduces the number of multiply-accumulate (MAC) operations required for the matrix multiplication, as MAC operations can be eliminated if either the corresponding input or the corresponding weight is zero (figure 3). Accordingly, the theoretical computational savings associated with either weight sparsity or activation sparsity are directly proportional to the fraction of zeros. When both forms of sparsity are combined there is potential to yield significant multiplicative benefits (figure 1).

In practice it has proved extremely difficult to realize these performance benefits on current hardware architectures. Even for DNNs with high-degrees of weight sparsity, the performance gains observed are small. For example, on CPUs, even for weight sparse networks in which 95% of the neuron weights have been eliminated, the performance improvements observed are typically less than 4× [55]. In addition, there are almost no techniques that simultaneously exploit both weight and activation sparsity. In part due to these difficulties, sparse networks have not been widely deployed in commercial settings.

Modern hardware architectures thrive on processing dense, regular data structures, making the efficient processing of sparse matrices challenging. Sparse matrices are typically represented in a compressed form, where only the non-zero elements are retained, along with sufficient indexing information to locate the elements within the matrix. Given the processing overheads associated with these formats, they work best for extremely sparse matrices, 99% sparse or greater, such as the matrices used in high performance computing [5].

In DNNs, where the level of weight sparsity is lower, the overheads associated with these compressed formats significantly curtail the observed performance benefits. In addition, there are also overheads associated with determining which elements should be non-zero. For sparse activations, the non-zero elements are input dependent, and must be repeatedly recomputed during inference. There are overheads for generating an appropriate representation of the sparse activations. These overheads are not incurred when activations are dense, and represent a significant obstacle to achieving speedups from activation sparsity.

These challenges, and current hardware friendly solutions such as block and partitioned sparsity, are further discussed in supplementary section 1 (https://stacks.iop.org/NCE/2/034004/mmedia).

3. Complementary Sparsity

Directly processing a native representation of a sparse matrix is inefficient because of the presence of the zero-valued elements. Techniques such as block and partitioned sparsity (see supplementary material) help align the patterns of non-zero elements with hardware requirements, but are fundamentally at odds with creating highly sparse and accurate networks. Optimal performance requires large blocks and reduced partition sizes but this limits both the obtainable sparsity and the accuracy [41]. This in turn compromises these approaches from achieving the theoretical performance benefits of highly sparse networks.

We propose an alternate approach that inverts the sparsity problem by structuring sparse matrices such that they are almost indistinguishable from dense matrices. We achieve this by overlaying multiple sparse matrices to form a single dense structure. An optimal packing can be readily achieved if no two sparse matrices contain a non-zero element at precisely the same location. Given incoming activations, we perform an element-wise product with the incoming activations (a dense operation) and then recreate each individual sum.

We term this technique *Complementary Sparsity*. Complementary Sparsity introduces constraints upon the locations of non-zero elements but it does not dictate the relative positions of the non-zero elements, nor does it dictate the permissible sparsity levels. The technique can be applied to convolutional kernels by overlaying multiple 3D sparse tensors from a layer's 4D sparse weight tensor. Importantly, the technique provides a path to linear performance improvements as the number of non-zero elements decreases, even for very high levels of sparsity.

Figure 4(a) illustrates the use of Complementary Sparsity for convolutional kernels. In this example, each kernel is 80% sparse, and a set of five kernels with non-overlapping patterns is overlaid to form a single dense kernel. The number of sparse kernels that can be combined scales proportionally with their sparsity. The primary constraint is that the non-zero elements in each set should not collide with each other. Note that it is not necessary that all the weights in a layer are non-overlapping—the restriction applies only to each set being combined. Using our 80% example, if a convolutional layer contains 20 channels, there are four dense sets each containing five sparse kernels. The elements must be complementary within a set, but there are no restrictions across the four sets. Given this flexibility, in practice we have found that networks trained with the restrictions imposed by Complementary Sparsity do not compromise on accuracy when compared with unstructured sparsity.

Another important advantage of Complementary Sparsity is that it provides a path to facilitate both sparse weights and sparse activations. In the following subsections, we first describe the architecture of sparse–dense networks (i.e. networks with sparse weights and dense activations) and then describe the extension to sparse–sparse networks. Finally, we describe how these concepts can be implemented in an FPGA[1].

3.1. Complementary Sparsity and sparse–dense networks

The basic technique described above combines multiple sparse weight structures into a single dense entity, and natively supports sparse–dense networks, i.e. networks with dense activations and sparse weights. Partial results from each sparse entity must be kept separate and independently accumulated for final results. In sparse–dense networks processing is comprised of four distinct steps (figure 4(b)):

[1] Although the general notion of sparse weights is inspired by neuroscience, our specific pattern of complementary weights is primarily designed for efficiency on hardware.

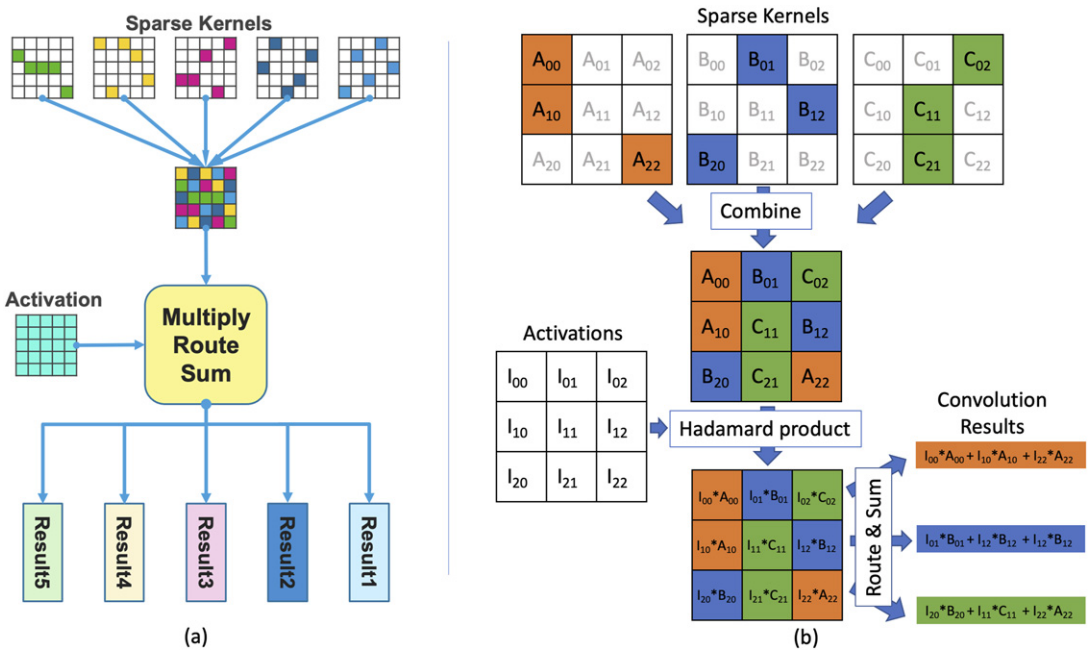


Figure 4. (a) Complementary Sparsity packs multiple sparse convolutional kernels into a single dense kernel for processing. A routing network then computes each individual sum. (b) An explicit worked out example of Complementary Sparsity, where the colors denote how weights for each kernel flow through the system. With 66% sparsity, three sparse kernels are combined into a single dense kernel. Each activation is then multiplied by a single weight value using a dense Hadamard product. This ensures there are no collisions between the product terms in the final convolution results.

(a) **Combine**: multiple sparse weight structures are overlaid to form a single dense entity. This is done offline as a preprocessing step.

(b) **Multiply**: each element of the activation is multiplied by the corresponding weight elements in the dense entity (Hadamard product).

(c) **Route**: the appropriate element-wise products are routed separately for each output.

(d) **Sum**: routed products are aggregated and summed to form a separate result for each sparse entity.

The optimal techniques for implementing each component are dictated by the specifics of the target hardware. For example, in some cases, instead of routing the element-wise products, it may prove preferential to reorder the incoming activations.

Given that Complementary Sparsity reduces N sparse convolutions into a single dense operation, there is the potential for a linear N-fold performance improvement. The key challenge is to reduce the cost associated with routing and accumulating the packed results. Accordingly, of particular interest are techniques focused on minimizing the overheads associated with the routing of the Hadamard product terms. Implementation of arbitrary routing usually involves resource hungry crossbar modules, where footprint increases as the square of the number of inputs. However, for DNN inference operations, the locations of the non-zero elements have been determined during training, and remain static throughout inference. The required routing is both fixed and predetermined, ensuring efficiency by tailoring implementations to the specific requirements of the network.

To further minimize the overheads associated with the routing of the product terms, Complementary Sparsity can be combined with the other forms of structural sparsity. For example, in figure 4(a), each column in the kernel is a partition, with one non-zero element permitted per column. Similarly, complementary patterns with blocks of non-zero elements are also possible. Section 3.3.2 below describes our FPGA implementation of routing in more detail, and section 5 analyzes resource tradeoffs.

3.2. Complementary Sparsity and sparse–sparse networks

The above sparse–dense Complementary Sparsity technique can be extended to handle sparse–sparse networks, i.e. networks comprised of both sparse activations and sparse weights. As discussed in section 2.3, significant inefficiencies are traditionally associated with sparse–sparse computations due to the changing locations of non-zero elements in the activations. The overheads associated with pairing these non-zero activations with their respective non-zero weights degrades any performance gains associated with processing the mutually non-zero subset of elements.

Using Complementary Sparsity, the sparse–sparse problem is simplified to a problem with sparse activations and dense weights, eliminating the above overheads. As illustrated in figure 5(a), when the sparse weights are represented in a dense format, the incoming sparse activations are paired with the relevant weights. For each non-zero activation there exists a corresponding column of non-zero weight elements at a predefined location in the dense weight structure. Processing is comprised of the following five steps:

(a) **Combine**: multiple sparse weight structures are overlaid to form a single dense structure. This is done offline as a preprocessing step.
(b) **Select**: a k-WTA component is used to determine the top-k activations and their indices.
(c) **Multiply**: each non-zero activation is multiplied by the corresponding weight elements in the dense structure (Hadamard product).
(d) **Route**: the appropriate element-wise products are routed separately for each output.
(e) **Sum**: routed products are aggregated and summed to form a separate result for each sparse matrix.

Compared to sparse–dense, the extensions are in the second and third steps. The computation in the third step is reduced in proportion to the sparsity of the incoming activations. As before there is additional overhead imposed by routing. For sparse–sparse there is also additional overhead imposed by the k-WTA block. An efficient implementation of these components is critical to realizing an overall benefit, and are detailed below in sections 3.3.2 and 3.3.3.

3.3. Complementary Sparsity on FPGAs

In this section, we discuss our implementation of Complementary Sparsity on FPGAs, before presenting both performance and resource utilization results in sections 4 and 5. We focus our discussion on the sparse–sparse implementation of convolutional kernels, specifically the individual components of figure 5(a). The implementations are focused on inference operations. As discussed, the flexible architecture of FPGAs represents a model platform for exploring idealized circuit structures for Complementary Sparsity.

3.3.1. Sparse–sparse Hadamard product computation

For each of the K non-zero activations, its index is used to extract the relevant weights (figure 5(a)) which are then multiplied in an element-wise fashion (the 'multiply' step in section 3.2). The individual terms of the Hadamard products are then routed separately to compute the sums for each output channel.

The key to computing this efficiently is an offline preprocessing step that combines sets of sparse weight kernels into smaller sets of augmented dense structures, denoted as AWTs (i.e. the 'combine' step in section 3.2). We combine each complementary sparse kernels into a smaller number (L) of dense complementary sparse filter blocks (CSFB), as denoted in the middle of figure 5(b). These 3D tensors are then flattened into 1D columns and concatenated together horizontally into an AWT. In addition, in the AWT each non-zero weight value has a sparse kernel ID (KID) co-located with it. The KID flows through to each of the resulting product terms and is used for subsequent routing (described below in section 3.3.2).

Given this structure, one approach to computing the Hadamard product would be to serially access the AWT, once for each of the K non-zero activations. Instead, in our implementation we pre-load K instances of the AWT into a set of separate memories on the FPGA. The output port of the memory delivers one element from each of the L CSFBs in each AWT in parallel. All activation aligned weights can now be read out in parallel from this now multi-ported AWT. As a result, the Hadamard products for each column of the CFSB can be computed in a single cycle. (Figure 4 in the supplementary material describes the computation in detail.) The construction of the multi-ported AWT is illustrated in figure 5(b). Note that this is an offline process done once for each convolutional layer.

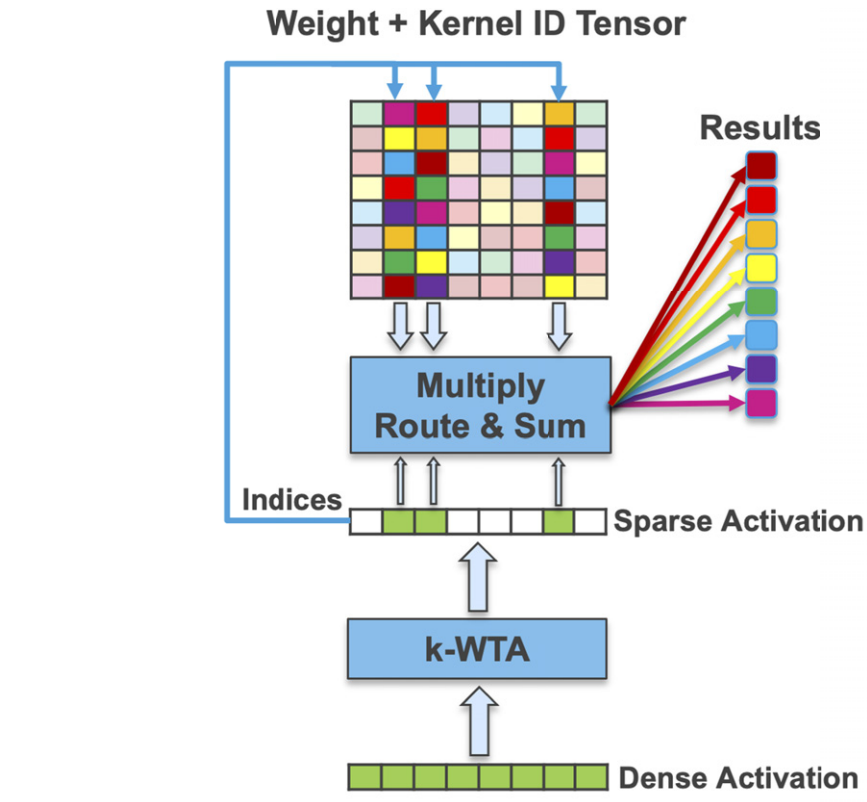
At inference time, the following formula generates the lookup address for the AWT, where (W_x, W_y) are the coordinates of columns in the CSFB, I_j is the index associated with jth non-zero activation value, and C_{in} is the number of channels in the input image to the layer:

$$\text{Address} = I_j + W_x * C_{\text{in}} + W_y * C_{\text{in}} * W \tag{1}$$

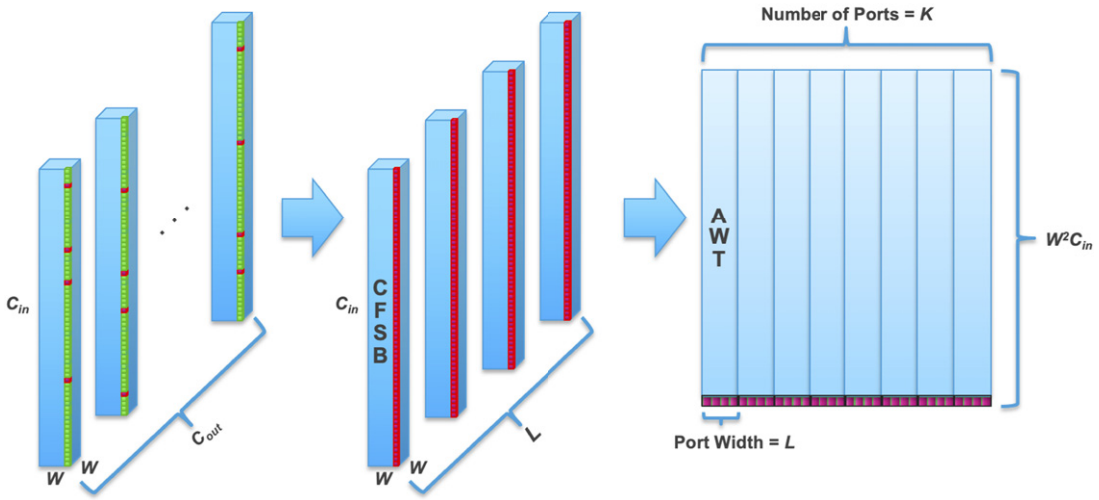
A key scaling issue with this scheme is the amount of memory consumed by the complete AWT structure. The total number of bits for the multi-ported AWT is:

$$B_{\text{M}} = C_{\text{in}} * W^2 * K * L * B_{\text{E}} \tag{2}$$

Here B_{E} is the size of each element and is the sum of the bit size of the weight element value, B_{W}, plus the bit size of the associated KID, B_{ID}. In our implementation we use eight-bit weights so $B_{\text{W}} = 8$. To determine B_{ID}, we need to calculate the number of sparse kernels, F, that can fit into a single CSFB. The non-zeros weights in each sparse kernel are distributed using partitioned weight sparsity along the C_{in} dimension (see supplementary



(a)



(b)

Figure 5. (a) High level architecture for handling sparse–sparse layers. Sparse activations are computed dynamically using *k*-WTA. The index of each non-zero activation is used to pair weights with each activation value for computing the final results. (b) The figure shows how the augmented weight tensor (AWT) structure is constructed offline. See text in section 3.3.1 for details.

material for an explanation of partitioned sparsity). With N non-zeros in each column of the sparse filter kernel, $F = C_{\text{in}}/N$. Therefore $B_{\text{ID}} = \lceil \log_2(F) \rceil$. If C_{out} is the number of output channels produced by the layer, the number of CSFBs, L, is equal to C_{out}/F. Plugging this into equation (2) yields:

$$B_{\text{M}} = W^2 * C_{\text{out}} * N * K * B_{\text{E}} \tag{3}$$

Note that the required memory decreases as activation sparsity is increased (decreasing K). Similarly the required memory decreases as the weight sparsity is increased (decreasing N). Therefore the memory savings with weight and activation sparsity are multiplicative. Overall we found that with sparse–sparse networks,

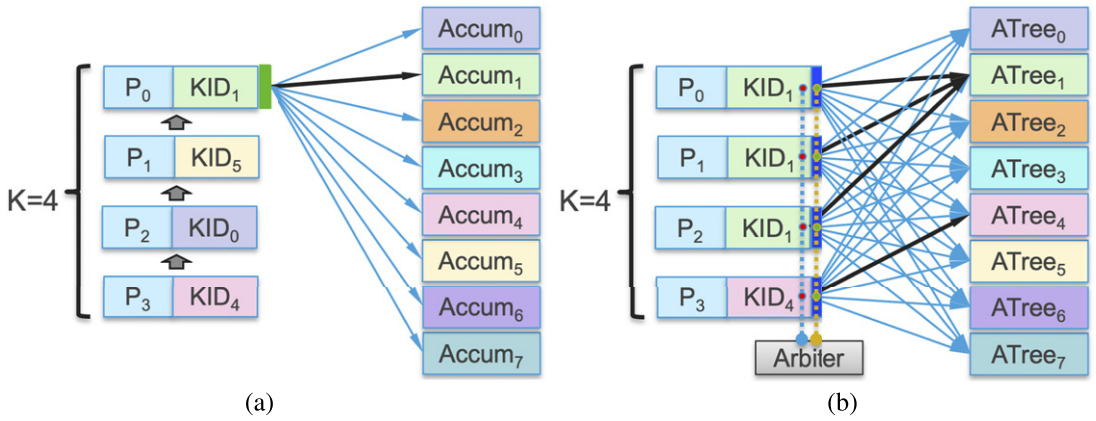


Figure 6. Sparse–sparse product term routing: (a) product terms serially routed to the appropriate accumulator, (b) product terms routed in parallel to the appropriate adder-tree. See text for details.

this approach of replicating weights enables far higher throughput with very favorable memory scaling (see section 5 for an in-depth study of resource scaling).

3.3.2. Sparse–sparse Hadamard product routing

A second critical component is the efficient routing of the products terms from the Hadamard operation. Once an activation has been multiplied by the retrieved weights, to complete the computation of the convolution, each resulting product term must be combined with the other product terms from the same kernel. The relevant products are identified using their sparse KID tag, which are copied from the sparse KID field of the associated augmented weight.

For K non-zero activation vectors, the retrieved weights may belong to a single sparse filter kernel (identical sparse KIDs), or might be distributed across several sparse filter kernels. Each of the K activations can be processed serially, in which case the results for each of the products can be simply routed via a multiplexor network to an designated accumulator, based upon its sparse KID. This is diagrammed in figure 6(a). The P_i represent the product terms along with their associated sparse KIDs, KID_j. The KID_j are used to successively index a single multiplexer to route the product term to the relevant accumulator Accum_j to be summed. The black arrow indicates the selection process. This operation is performed serially K times.

For greater performance, the products from all the activations can be processed in parallel. In this case, the product terms must be routed simultaneously to adder trees for summing, rather than to a single accumulator (figure 6(b)). Note that the active routes, marked by the black arrows, indicated that three product terms, P_0, P_1, and P_2 have identical sparse KIDs of 1, which lands them on ATree_1. All the adder trees need capacity to handle the possibility of all K product terms being routed to a single adder tree.

Routing of multiple product terms to non-conflicting inputs in an adder tree introduces additional complexity. Not only is it necessary to route based upon the sparse KID, but additional destination address bits are required to designate the specific input port of the adder in which the product term should land. This is resolved with an arbitration module, which supplies these additional address bits before the product is passed to a larger multiplexer network. This is indicated in figure 6(b) by a blue dotted line terminating on the Arbiter block.

The arbitration module generates the low order address bits from the set of sparse KIDs. The generated low order bits are concatenated to the sparse KIDs, represented by the yellow dotted line leaving the Arbiter and passing through the dark blue multiplexer blocks. The fine-grained fan-out to individual ports of the adder tree is not illustrated. Further arbitration module details can be found in supplementary section 5.

Sparsity partitioned in the channel dimension, as reflected in the range of sparse KIDs, reduces the bit size of these indices since we only need sufficient bits to identify the sparse kernel within the channel dimension, not the location within the $W^2 * C_{\text{in}}$ locations of a dense filter kernel. Small values for K, reflecting high activation sparsity, reduces the number of low order bits needed for adder tree input port assignment in the parallel implementation. Small values of N, reflecting high weight sparsity, also reduce the number of low order bits needed, since the number of product terms which can be directed towards a single adder tree is $\min(K, N)$.

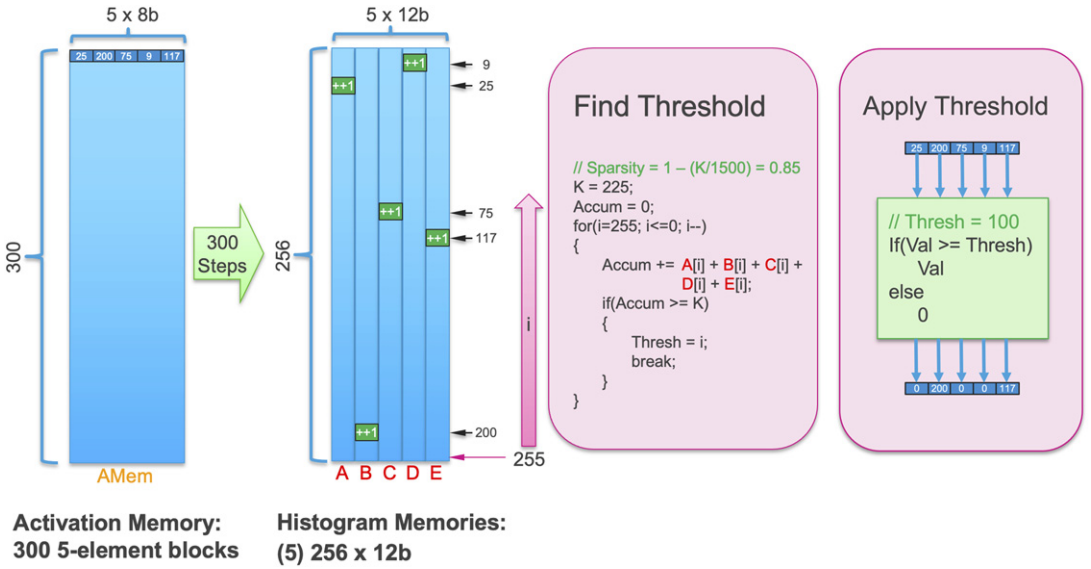


Figure 7. Parallel global *k*-WTA: performs a histogram based search of entire activation to determine the threshold yielding *K* non-zero elements. In this example, the 1500-element activation is stored as 300 five-element blocks in memory AMem. Each block is read out and the element values are used to address and increment counts in five separate memories, A–E. The counts are then cumulatively summed together into variable Accum, starting with the largest value location, until Accum reaches a total count of *K*, establishing a thresholding value. Values in AMem are then compared to thresh, and sent through to the next layer if they are greater than or equal to thresh, along with their indices in the original 1500 element vector.

3.3.3. Activation sparsity using k-WTA

For *k*-WTA, activation sparsity is induced by explicitly restricting the number of non-zero elements to the *K* largest values produced by a layer. Determining these top *K* values efficiently can represent a significant obstacle to the effective use of activation sparsity. The time and resources expended performing the sort operation erodes the performance benefits associated with leveraging the resulting sparsity in subsequent processing. Accordingly, an optimized *k*-WTA implementation is central to our FPGA implementation. We divide *k*-WTA implementations into two broad categories:

- **Global**: all elements of an activation are examined to determine the *K* largest. We use global *k*-WTA following linear layers.
- **Local**: the activation is partitioned into smaller units, and only the elements belonging to a partition are compared to each other. We use local *k*-WTA following convolutional layers, where the winner take all competition happens along the channel dimension.

For eight-bit activation values, our implementation of global *k*-WTA leverages a histogram-based approach. In our implementation, a 256-element array in memory is used to build the histogram, with each activation value being used to increment a count at a location addressed by that value. Once all of the activation values have been processed, the histogram array represents the distribution of the activation values. For a specified value of *K*, the histogram values can be read, largest first, to determine the appropriate minimum value cutoff; values above this threshold should be retained as part of the top-*k* and the remainder discarded. As a final step, the activation values are compared against the threshold and the winners passed to the next layer.

For improved performance, an implementation may process multiple activation elements in parallel. In this scenario, multiple histograms are built in parallel and then combined to determine the overall cutoff value. An example of this implementation is illustrated in figure 7, for 1500-activations, five-way parallelism, and activation sparsity of 85%.

For convolutional layers, activations have a natural partitioning in the channel dimension. We exploit that partitioning by applying local *k*-WTA to each channel independently. This partitioning provides efficiency benefits by reducing the number of elements that must be sorted. As in global *k*-WTA, the position of each result value produced by the convolutional layer must be tracked through the sorting process. This is achieved by appending an index to each data value entering the sorting function.

Sorting is performed in several stages, and we optimized the implementation based on the key observation that it is only necessary to find the top *K* values in each vector. The ordering of the low valued elements is immaterial, and, as *K* decreases with increasing activation sparsity, the cost of the sorting implementation falls accordingly. Note that activations of the preceding weight layer may arrive serially in parallel bursts or as

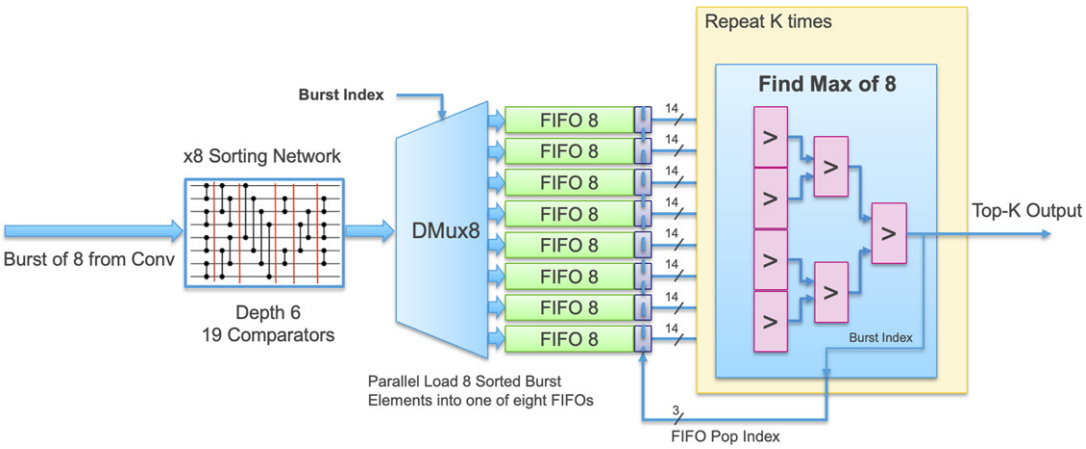


Figure 8. Structure of *k*-WTA when serially processing sparse convolutions. The input consists of *M* bursts, each consisting of *N* values (and indices). In this example, both *M* and *N* are eight. The sorting network is comprised of 19 comparators, arranged into depth six layers. The depth of the comparator tree to find the maximum of eight values is 3 ($\log_2(8)$).

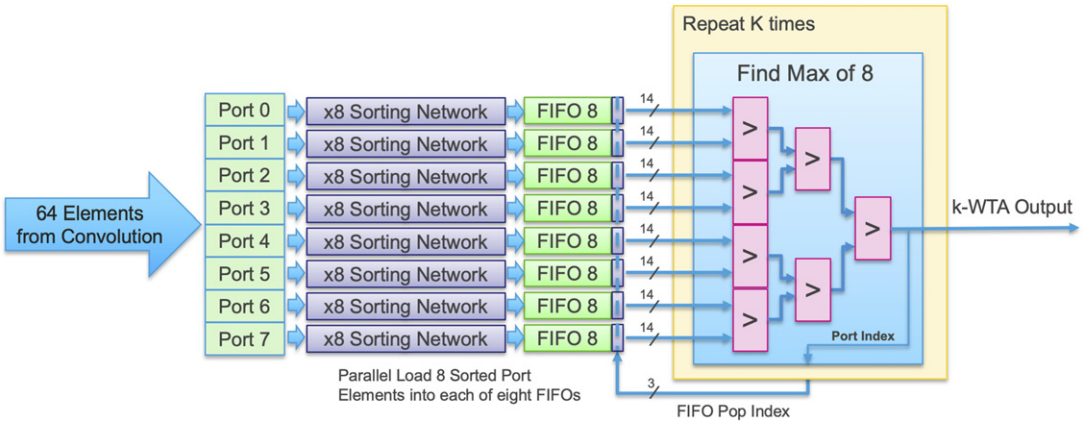


Figure 9. Structure of *k*-WTA when parallel processing complementary sparse convolutions.

a single parallel vector depending on how they are computed. To avoid bottlenecks, the performance of the *k*-WTA implementation should be matched to the performance of the convolutional operator.

In the serial burst implementation (figure 8), the input consists of *M* bursts, each consisting of *N* values (and indices). Each burst is first sent through a sorting network [38], which orders the burst by value, largest value first. The sorted burst is then loaded into one of a set of *M* FIFOs via a multiplexor. Once all the FIFOs have been loaded, a vector composed from the *M* top-of-FIFO values is then passed through a $\log_2(M)$ stage comparator tree, in order to determine the maximum value in the vector. The maximum value is retained, and its associated indexing information (which indicates in which FIFO the value was located) is used to pop that element from the appropriate FIFO, exposing the FIFO's next largest element. This process is repeated *K* times; at which point the output vector has been filled with the top *K* elements and is passed to the next processing layer.

If the input from the convolutional layer arrives in parallel instead of in bursts, higher performance can be achieved by removing the demultiplexer (DMux8) shown in figure 8 and replicating the sorting network to directly feed each FIFO, as shown in figure 9.

In summary, there are a number of ways to implement *k*-WTA efficiently. With an appropriate implementation choice, we find that overall the *k*-WTA is a relatively small percentage of overall resource usage (see section 5.3).

4. Results on an end to end speech network

In this section we discuss the application of Complementary Sparsity to an end to end speech recognition system. We trained a convolutional network [61] to recognize one-word speech commands using the Google

Table 1. Architecture of the CNN network trained on GSC data.

Layer	Channels	Kernel size	Stride	Output shape
Input	—	—	—	$32 \times 32 \times 1$
Conv-1	64	$5 \times 5 \times 1$	1	$28 \times 28 \times 64$
MaxPool-1	—	$2 \times 2 \times 1$	2	$14 \times 14 \times 64$
Conv-2	64	$5 \times 5 \times 64$	1	$10 \times 10 \times 64$
MaxPool-2	—	$2 \times 2 \times 1$	2	$5 \times 5 \times 64$
Flatten	—	—	—	1600×1
Linear-1	1500	1600×1	—	1500×1
Output	12	1500×1	—	12×1

Speech Commands (GSC) dataset [79]. We implemented dense and sparse versions of the network on both large and small FPGA platforms. Our goal was to study the impact of Complementary Sparsity on full system throughput (the number of words processed per second) and understand trade-offs in resources, memory consumption and energy usage.

GSC consists of 65 000 one-second long utterances of keywords spoken by thousands of individuals. The task, to recognize the spoken word from the audio signal, is designed for embedded smart home applications that respond to speech commands. State of the art convolutional networks on this dataset achieve accuracies (before quantization) of 96%–97% using ten categories [65, 70].

Our base dense GSC network is a standard convolutional network composed of two convolutional layers, a linear hidden layer plus an output layer, as described in table 1. We also trained a sparse network with identical layer sizes but with both sparse weights and sparse activations. Our sparse network follows the structure and training described in [1]. To enforce sparse weights we used a static binary mask that dictates the locations of the non-zero elements and meets requirements of Complementary Sparsity. The ReLU activation function was replaced by a (k-WTA) [1, 47] activation function (see section 2.2.2 and figure 2).

The baseline dense version of the network contained 2522 128 parameters, while the sparse network contained 127 696 non-zero weights, or about 95% sparse. The activations in the sparse network range from 88% to 90% sparsity (i.e. 10%–12% of the neurons are 'winners'), depending on the layer. Both dense and sparse models were trained on the GSC data set, achieving comparable accuracies (see [1] for details). In our implementation, the accuracies of the sparse and dense networks are between 96.4% and 96.9%. Both activations and weights are quantized to eight-bits.

4.1. FPGA implementation

We implemented the baseline dense GSC network using the Xilinx™ software 'Vitis AI' [86]. Vitis AI is the preferred solution for deploying deep learning networks on Xilinx FPGA platforms. Convolution and linear layers in Vitis AI invoke hand-optimized processing elements (PEs) implemented using RTL. A software compiler converts a given network, including parameters and weights, into schedules of calls to these PEs.

We implemented our sparse GSC networks using the Xilinx Vivado HLS toolset [83, 84]. Although HLS uses a C++ compiler (with Xilinx specific pragmas) and does not produce hand-optimized designs, it represents a faster design path. There was sufficient flexibility in the toolset to implement our sparse designs. We note however that the results for our sparse networks below would likely be improved using hand-optimized designs.

We created two pipelined implementations of our sparse network using HLS. The *sparse–dense* implementation leveraged weight sparsity in Conv-2 and the linear layer, ignoring sparse activations. The *sparse–sparse* implementation leveraged both sparse activations and sparse weights (as described in section 3.3). In the sparse–dense implementation, the Conv-1 layer was left as fully dense as its profile was small relative to the other pipeline stages. In the sparse–sparse implementation the other stages became faster and Conv-1 became a bottleneck. As such we implemented Conv-1 using a sparse–dense strategy (the input to the network is dense, hence sparse–sparse is not an option for Conv-1).

4.2. Benchmark description

The performance of the three different CNN implementations were tested on two different Xilinx FPGA platforms. The first, the Alveo™ U250 [85], is a high-end card targeted at data centers, while the second, the UtraScale+™ ZU3EG [87], is a smaller system targeted at embedded applications. Compared to the ZU3EG, the U250 has 11× the number of system logic cells, about 56× the internal memory, and consumes 9× more power.

For each CNN network on each FPGA two different experiments were undertaken:

Table 2. Throughput of single sparse and dense networks on the U250 and ZU3EG platforms, measured in words processed per second. The dense network did not fit on the ZU3EG due to its limited resources. All sparse implementations, regardless of platform, were significantly faster than the dense network running on the U250. The sparse–sparse implementation was consistently 2× to 3× faster than the sparse–dense implementation.

FPGA platform	Network implementation	Throughput	Speedup
U250	Dense	3049	1.0
	Sparse–dense	35 714	11.71
	Sparse–sparse	102 564	33.63
ZU3EG	Dense	0	—
	Sparse–dense	21 053	N/A
	Sparse–sparse	45 455	N/A

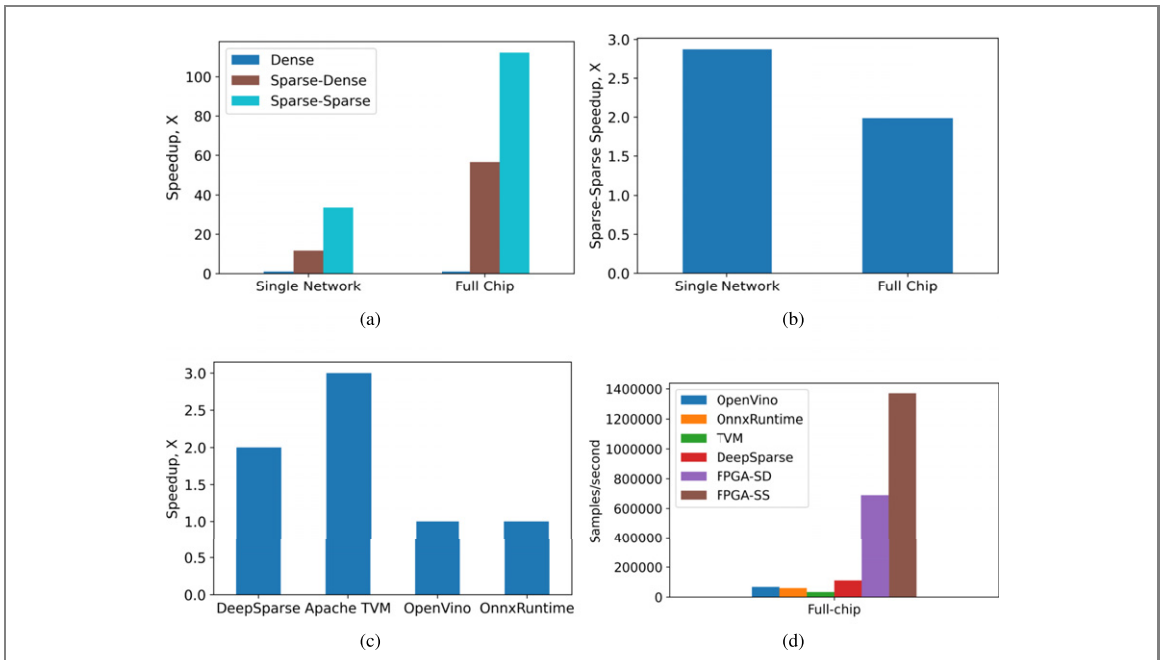


Figure 10. Performance comparisons between sparse and dense networks: (a) sparse network performance on the U250 relative to dense, (b) sparse–sparse network performance, relative to sparse–dense, on the U250, (c) sparse–dense network performance on CPUs using common inference runtimes (relative to a dense network on the same runtime), (d) sparse network performance on CPUs and FPGAs.

(a) **Single network performance**: a single network is a pipelined implementation of one GSC network, processing a single stream of speech commands.

(b) **Full chip performance**: multiple network instances are placed on the FPGA until the entire FPGA's resources are exhausted, or the design cannot be routed by the software. Multiple input streams are distributed across the instances, and the inference throughput delivered by the entire chip is reported.

In both experiments, the input data is a repeating sequence of 50 000 pre-processed audio samples. We chose overall throughput, measured as the total number of input words processed per second, as the primary performance metric.

4.3. Single network results

Table 2 shows the results of running a single network instance on the U250 and ZU3EG platforms. On the U250, the sparse–dense implementation achieves over 11.7× the throughput of the dense implementation, while the sparse–sparse implementation outperforms both the dense and the sparse–dense implementations by 33.6× and 2.8× respectively (figures 10(a) and (b)).

The dense implementation did not fit on the smaller ZU3EG platform due to the limited resources available. Both sparse implementations were able to compile and run successfully on the platform due to their smaller size and lower resource requirements. The sparse–sparse implementation was about 2.1× faster than the sparse–dense implementation. Interestingly, the sparse–dense implementation on the ZU3EG platform

Table 3. Full-chip throughput of sparse and dense networks on the U250, measured in words processed per second. The relatively compact footprint of the sparse networks allowed the compiler to fit a larger number of networks per chip. The sparse–sparse implementation was over 100× faster than the dense implementation.

FPGA platform	Network implementation	Total networks	Throughput	Speedup
U250	Dense	4	12 195	1.0
	Sparse–dense	24	689 655	56.5
	Sparse–sparse	20	1369 863	112.3

was still 6.9× faster than the dense implementation on the more powerful U250. This demonstrates the performance benefits associated with sparse networks, and also the potential for sparse networks to open up new applications in embedded scenarios that were previously impossible.

4.4. Full chip results

Table 3 shows the full-chip throughput results for the U250. The numbers illustrate the performance benefits of sparse networks. In the experiments on the U250, the sparse–sense and sparse–sparse implementations outperformed the dense implementation by 56.5× and 112.3× respectively (figure 10(a)). The increased performance delta between the dense and sparse implementations can be attributed to the relative compactness of sparsity allowing significantly more sparse networks to be accommodated on the chip (e.g. 20 sparse–sparse networks versus four dense networks). This results in the observed increase in aggregate throughput. Only one copy of each sparse network could fit on the ZU3EG, thus overall throughput on this platform was identical to that in table 2.

Note that the 20× replication count achieved for the sparse–sparse implementation is lower than the 24× replication achieved for the sparse–dense. The added complexity of handling sparse activation indices (see section 3.3.2) increases the FPGA resources required to support the network. Nevertheless, the additional performance benefits associated with exploiting activation sparsity more than outweigh the resource costs, almost doubling the aggregate throughput.

4.5. Comparisons with CPU inference engines

In this section we report performance gains of our sparse GSC network on a variety of widely available inference runtimes. The CPU in these experiments is a 3.0 GHz 24-core Intel Xeon 8275CL processor. Figure 10(c) demonstrates the speedup of the sparse–dense network on these runtimes (relative to the dense network on the same engine) observed for our GSC CNN network. Most strikingly, both the well-known ONNX Runtime [58] and OpenVino [30] runtimes fail to exploit sparsity. For the other runtime engines the sparse networks outperform the dense network, with Neural Magic's Deepsparse [54] and the Apache TVM providing a 2× and 3× speedup, respectively. The observed performance gains are relatively modest, considering there is a 20× reduction in the number of non-zero weights.

In figure 10(d), the absolute performance for the sparse networks on the CPU and FPGA are compared. The results show significant speedups from sparsity on an FPGA with absolute performance over 10× that currently achievable on a CPU system. None of these runtime engines exploit both sparsity in activations and weights.

4.6. Power efficiency

In addition to improved inference performance, reduced power consumption is becoming increasingly critical [69, 72]. Table 4 shows the absolute and relative power efficiency for inference operations. Due to the significant resource reductions associated with sparse networks, not only has the total throughput improved, but the power consumed per inference operation has also dropped considerably. It is common to improve throughput at the expense of increasing energy consumption [72]. Our results demonstrate that sparsity avoids many unnecessary operations altogether, simultaneously improving throughput and power efficiency.

5. Resource tradeoffs analysis

In the previous section, we discussed end-to-end throughput results for a full network. It became clear during implementation that a key consideration is the resource usage required to implement the routing and *k*-WTA components. In this section we implemented a series of controlled experiments to analyze these resource tradeoffs in isolation.

Table 4. Power efficiency of sparse networks on the U250 and ZE3EG FPGAs in comparison with the dense network baseline. We estimate power efficiency using a word/sec/watt metric based on worst-case (i.e. total system power of each platform).

FPGA platform	System power (W)	Network type	Number of networks	Words sec/watt	Relative efficiency, %
U250	225	Dense	4	54	100
		Sparse–dense	1	158	292
		Sparse–dense	24	3065	5675
		Sparse–sparse	1	455	842
		Sparse–sparse	20	6088	11 274
ZU3EG	24	Dense	0	0	0
		Sparse–dense	1	877	1624
		Sparse–sparse	1	1893	3505

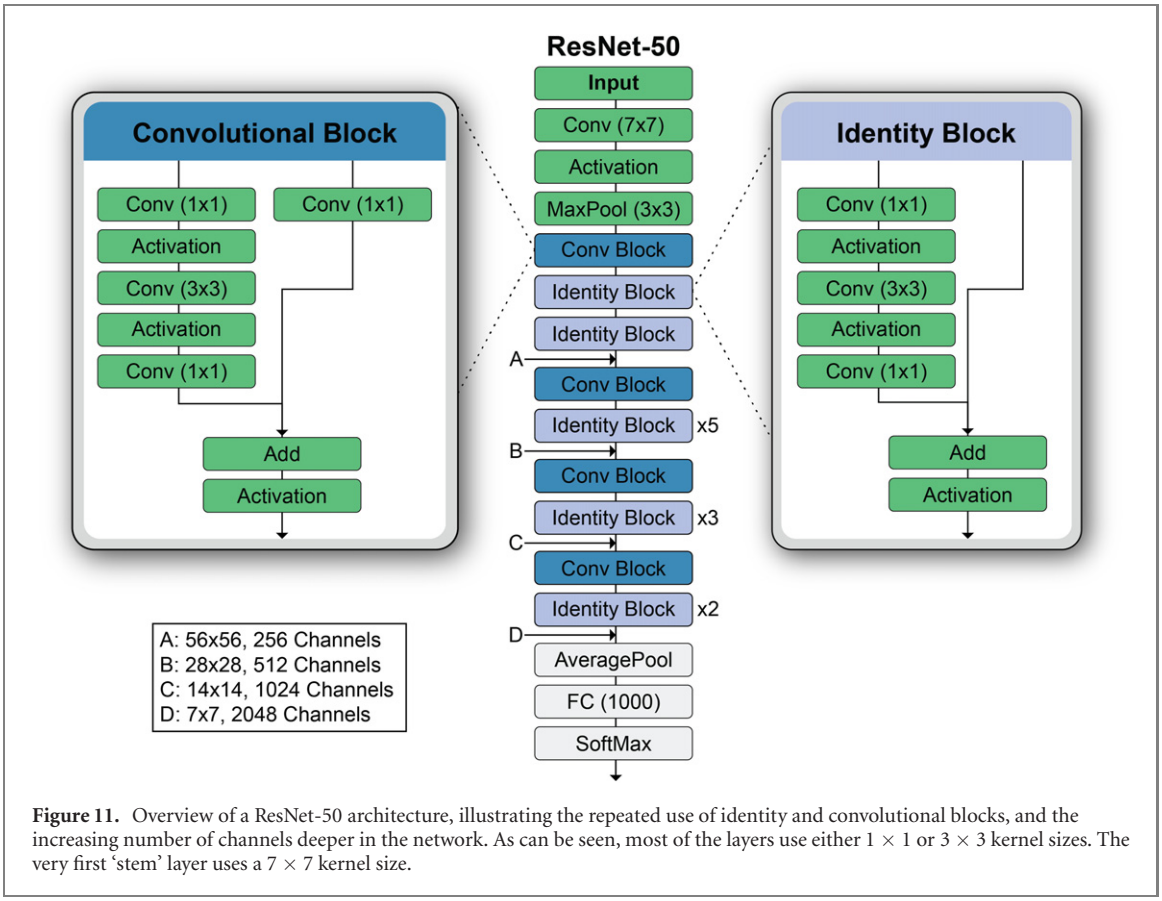


Figure 11. Overview of a ResNet-50 architecture, illustrating the repeated use of identity and convolutional blocks, and the increasing number of channels deeper in the network. As can be seen, most of the layers use either 1 × 1 or 3 × 3 kernel sizes. The very first 'stem' layer uses a 7 × 7 kernel size.

In GSC, the convolutional layers employed 5 × 5 kernels. In these experiments we focus on two other structures, 1 × 1 and 3 × 3 kernel types. These kernel types are typical of a number of common networks structures, such as the ResNet-50 (figure 11), ResNeXt, and MobileNetV2 networks [26, 66, 82]. We investigate the resource savings achievable via a combination of activation and weight sparsity applied to these convolutional layer types. The key questions revolve around how the FPGA resource requirements scale with weight sparsity, and how this changes as we add in activation sparsity.

5.1. Experiment setup

To investigate whether Complementary Sparsity could be applied generally, we developed the component shown in figure 5(a) as a set of general-purpose parameterized blocks. For the *k*-WTA block, *K* is defined per instance at compilation time. Three convolutional blocks were developed: a sparse–dense 7 × 7 convolutional block, and separate blocks for 1 × 1 and 3 × 3 sparse–sparse convolutions. The parameterization of these blocks included: boundary padding size, stride, weight sparsity, and memory bandwidth, as well as input activation sparsity for the 1 × 1 and 3 × 3 blocks.

When implementing components on an FPGA there is a great deal of flexibility in choosing how resources are allocated. There is significant latitude to trade serial processing for parallel processing by allocating sufficient resources to every stage. This in turn makes it challenging to explore both resource utilization and throughput in a controlled manner. In these implementations, we targeted a fixed throughput for all components in order to focus on resources. Our throughput target was chosen to be aggressive without leading to exploding resources. The primary target stipulated that a 1×1 [64:64][2] convolution should be computed in a single cycle. For a 1×1 [64:64] convolution, when weights and activations are dense, 4096 multiplications and 4096 additions are required to carry out the computation per spatial location. For a sparse–sparse computation ($N = 4$ and $K = 8$), this requirement is reduced to 32 multiplications and 32 additions, making this aggressive target feasible. Our 3×3 [64:64] convolution used nine 1×1 convolutions, taking about nine cycles. The k-WTA layer had a target of one cycle. As sparsity levels varied the compiler automatically allocated the hardware resources to achieve this target, allowing a controlled investigation of resource impact. We removed bandwidth as a confounding parameter by allocating sufficient memory to meet the target (but see section 5.5 below for an analysis of bandwidth).

The parameterization and above setup facilitated a systematic analysis of Complementary Sparsity for a variety of convolutional layers, primarily as a function of weight and activation sparsities. Our goal was to gain an improved understanding of the resource consumption and its scaling with degree of sparsity. Since extending sparse architectures to dense configurations is not meaningful, we confine our analysis to k-WTA activation sparsity $\geqslant 50\%$, and weight sparsity $\geqslant 50\%$. These represent reasonable break points between sparse and dense implementations.

5.2. Resource utilization of sparse–sparse convolution kernels

In figures 12(a)–(c), and 13(a)–(c), we present the resource utilization observed for the convolutional layers when activation sparsity is increased. In each experiment, we hold the weight sparsity constant, increase the activation sparsity and report the reduction in resource utilization. Our FPGA implementations of convolutional layers consume a variety of FPGA resources, including lookup tables (LUTs), flip flops (FFs), and memory blocks ultraRAMs (URAMs). We found that, for all investigated levels of weight sparsity, increasing the activation sparsity delivered a significant reduction in the resource utilization across all FPGA resources. For example, looking a figure 12(a), for a weight sparsity of four non-zeros out of 64-elements (i.e. $\frac{60}{64} = 93.75\%$ sparse), as the activation sparsity is increased from $\frac{16}{64}$ to $\frac{8}{64}$ and $\frac{4}{64}$, the number of LUTs required for the implementation of the 1×1 convolution is reduced by $2.7\times$ and $4.1\times$, respectively.

It is apparent that, across all levels of weight sparsity investigated, as the level of activation sparsity is increased, the complexity of the implementations of the convolutional layers is reduced. The degree to which the resource utilization decreased with increased sparsity is dictated by the resource type and the degree of weight sparsity. For instance, there is a clear linear relationship between URAMs consumed and the degree of activation sparsity. This is true for all levels of weight sparsity investigated.

LUTs are used for both routing and implementing multipliers in this design. The LUT count is reduced as a function of weight sparsity due to the decreased number of multipliers, while conversely there is greater routing complexity with increased weight sparsity. This latter effect is due to managing larger numbers of consolidated sparse weight kernels. Despite these competing factors, the overall LUT count decreases significantly with weight sparsity.

For FF utilization, the resource savings are more muted at higher weight sparsities. For FFs, which are primarily used for high bandwidth local storage, there is a baseline quantity for holding input and output values. The especially muted 3×3 convolution FF resource utilization results illustrated in figure 13(b) reflect the fact that FFs are also used to buffer intermediate results; the results of the nine internal 1×1 operations are serially accumulated. Therefore the FF utilization scaling as a function of weight sparsity is on top of these relatively static baselines. However, in many instances we demonstrate a super-linear reduction in resource utilization. This is rate of reduction is observed because a number of the elements in the implementations of the convolutions scale non-linearly with the number of non-zero activations; resulting in significant resource savings as K is decreased.

Figures 14(a)–(c) and 15(a)–(c) show the impact of varying weight sparsity for a fixed activation sparsity. In these results, the resource savings are sub-linear. Increases in weight sparsity result in decreases in the number of multiplies, but as discussed, routing overheads limit these reductions. However, for LUTs, FFs and URAMs, at any given activation sparsity, increasing the weight sparsity reduces the resource consumed by the implementation.

[2] Our notation [a:b] refers to an input channel count of a and an output channel count of b.

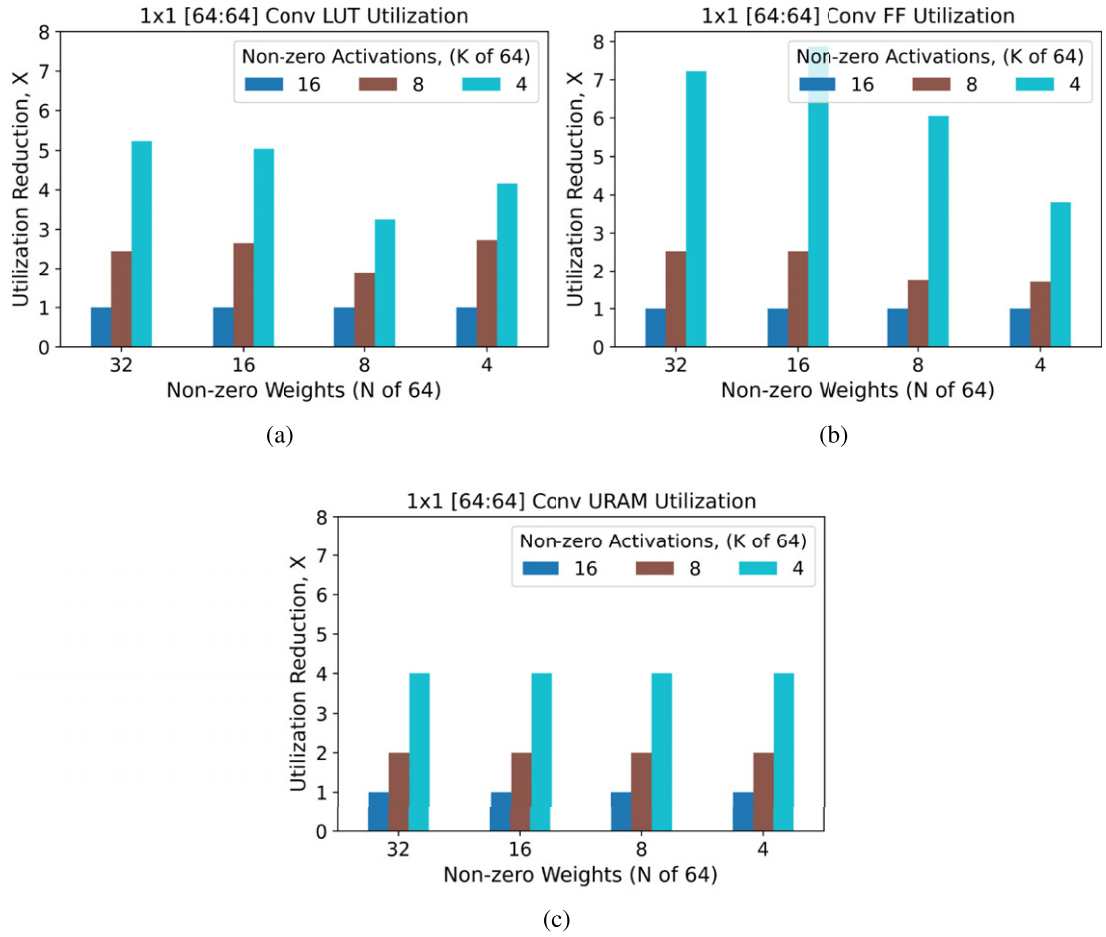


Figure 12. Impact of activation sparsity on resource utilization for 1×1 [64:64] convolution operations for different degrees of weight sparsity for: (a) LUTs, (b) FFs and (c) URAMs (K and N indicate the number of non-zero elements, reduction in utilization relative to $K = 16$).

In summary, for our FPGA implementations using Complementary Sparsity, increasing sparsity (weight, activation or both) results in more resource efficient implementations, while continuing to meet the stipulated throughput metric.

5.3. Resource utilization of *k*-WTA

We also investigated the resource impact of our k-WTA implementations. Here too increasing activation sparsity resulted in the consumption of fewer hardware resources, as illustrated in figure 16. The resource utilization was found to decrease almost linearly with the degree of sparsity. This represents an important synergy, with the convolutional kernel implementations benefiting from increased levels of activation sparsity, and the cost of providing the sparse activations decreasing as the level of activation sparsity is increased.

In figures 17(b) and (a), the combined resource utilization for sparse–sparse convolutions and their associated k-WTA components is shown. For both the 1×1 and 3×3 convolutions, the costs associated with the k-WTA implement is small compared with the costs associated with the convolutions, especially for the 3×3 convolution, where the implementation cost of the convolution is increased, but the k-WTA cost remains constant.

5.4. Sparsity in the network stem

In addition to the convolutional kernels that form the convolutional and identity blocks in ResNet-50, the network contains an initial 'stem'. This stem performs a $7 \times 7 \times 3$ (RGB color values) convolution on the input image [17]. In many sparse implementations, this first convolutional layer is left as a standard dense operation, because it represents a small part of the overall implementation profile. However, the implementation of this initial convolution can both require significant hardware resources and dictate overall network throughput. Although the overall latency of the entire network pipeline will shrink when recast as a sparse implementation,

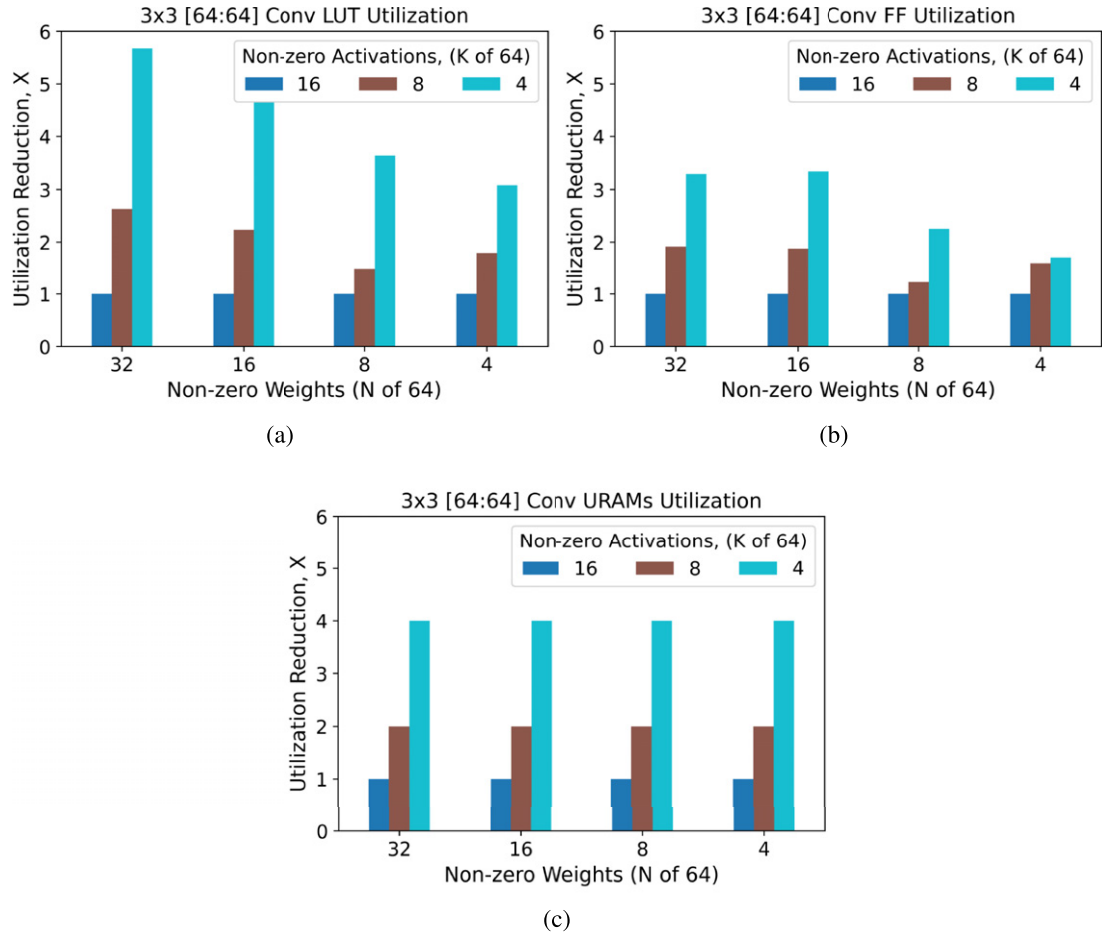


Figure 13. Impact of activation sparsity on resource utilization for 3×3 [64:64] convolution operations for different degrees of weight sparsity for: (a) LUTs, (b) FFs, (c) URAMs (K and N indicate the number of non-zero elements, reduction in utilization relative to $K = 16$).

the throughput benefits will be capped by this first layer, making an efficient sparse implementation highly desirable.

Complementary Sparsity can be successfully applied to this stem convolution, but, because the inputs to this first layer are dense images, a sparse–sparse implementation is not feasible. However, a weight-sparse implementation on an FPGA provides a considerable performance benefit: in our implementations, by increasing the weight sparsity (from $N = 9$ to $N = 5$) by $1.8\times$, we increased throughput by $1.6\times$. In this layer we chose to implement Complementary Sparsity in the spatial dimensions. We also imposed block sparsity constraints, with the three-element input dimension being treated as a block, either fully non-zero or completely zero.

The first layer of most DNNs process a dense input data stream and will only be able to exploit weight sparsity in a sparse–dense configuration. If the rest of the DNN is implemented as sparse–sparse layers those layers will see large performance gains. As an unexpected result, in pipelined implementations we find that the first layer's throughput will often dictate the maximum throughput of the network. To increase overall throughput, in FPGAs it is possible to increase the parallelism of the first layer, such that its sparse–dense layer latency is less than or equal to the highest latency sparse–sparse layer. This additional resource cost is made up by the resource gains achieved in the rest of the network. As a general rule, we find that the large gains achieved by a sparse–sparse implementation warrant careful profiling of the rest of the system as unexpected bottlenecks can emerge.

5.5. Sparse–sparse memory bandwidth considerations

Memory represents a scarce resource, and its efficient utilization is a key contributor to the success of sparse–sparse implementations. Two factors dictate memory utilization on the FPGA. The first is simply dictated by the capacity required to retain the relevant weight elements. Second is the requirement for sufficient memory bandwidth to extract all needed weight elements on a per-cycle basis. This bandwidth requirement

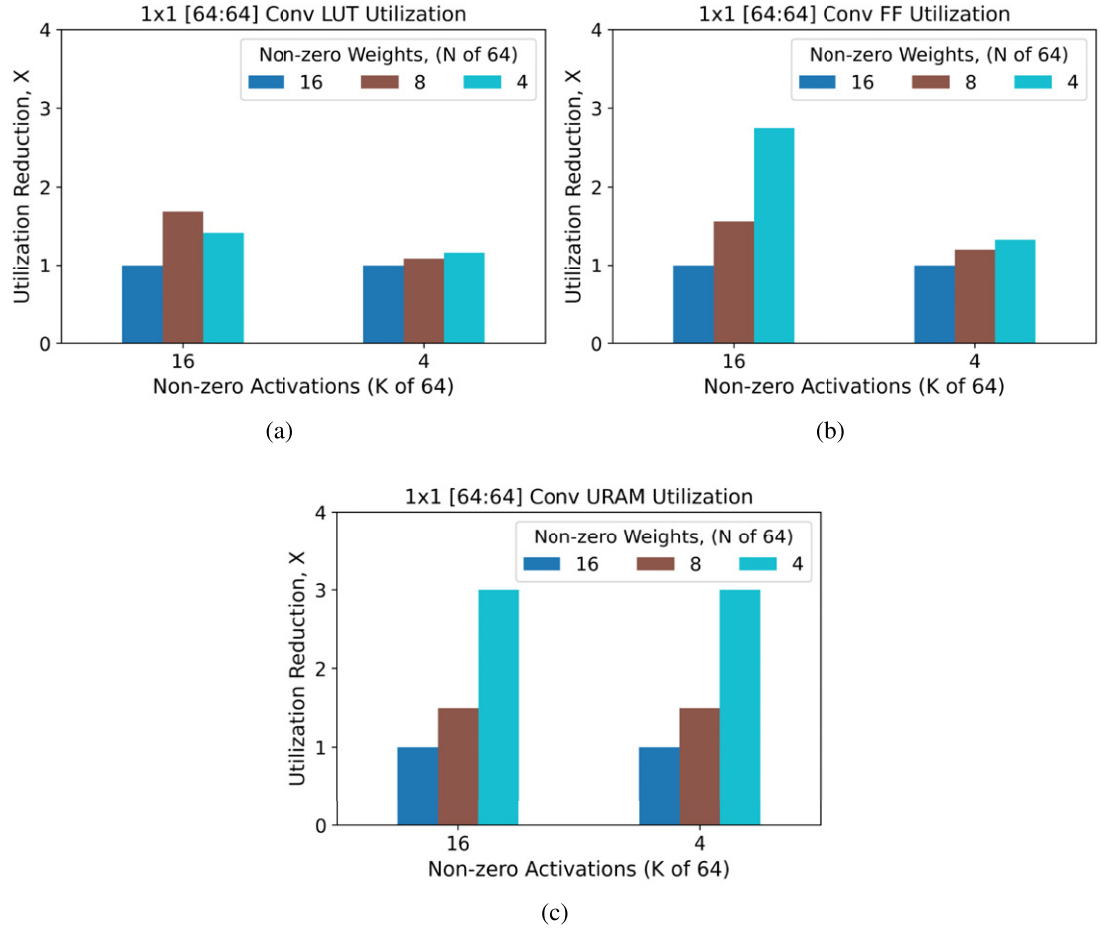


Figure 14. Impact of weight sparsity on resource utilization for 1×1 [64:64] convolution for different degrees of activation sparsity for: (a) LUTs, (b) FFs and (c) URAMs (K and N indicates the number of non-zero elements, reduction in utilization relative to $K = 16$).

is dictated both by the number of weights that need to be fetched for each activation and the number of activations that are processed in parallel:

(a) **Weight sparsity**: for each non-zero activation, the elements from the corresponding compacted dense weight kernels are processed in parallel. As weight sparsity is increased (i.e. smaller N), the width of the port required to support the parallel read of all the associated data decreases linearly.

(b) **Activation sparsity**: processing for each non-zero activation requires an independent lookup. If activations are processed in parallel, each operation requires its own memory port. As activation sparsity is increased (i.e. smaller K), the number of memory ports falls linearly.

On FPGAs, memory bandwidth requirements are served by numerous relatively small tightly coupled memories (TCM) that are implemented as static RAMs. On Xilinx platforms, URAMs are dual-ported, with a port width of 72 bits and a capacity of 288 Kbits (4096 locations). In our implementation, memory requirements for the first few stages of a network such as ResNet-50 will be driven by bandwidth rather than capacity. In order to achieve the stipulated throughput target, weights must be distributed across a larger number of URAMs than would be dictated by storage capacity requirements alone. The memory bandwidth required to support the computation of a 1×1 [64:64] convolution in a single cycle (i.e. fully parallelize [64:64] channel dot products) necessitates multiple dual-ported URAMs. As a result, the storage capacity of each URAM unit is relatively underutilized.

In summary, in this experiment where we employ a high degree of parallel computation to make an aggressive but fixed throughput target, sufficient local memory bandwidth is key. The pattern of access is not predictable, due to the dynamic selections of the k-WTA module. Although this disrupts location access coherence, the rate of access is predictable. This rate is a combined function of both weight and activation sparsity. Compared to an equivalent fully parallel dense network, sparse–sparse networks deliver significant reductions in both the number and capacity of FPGA TCMs, and the associated bandwidth.

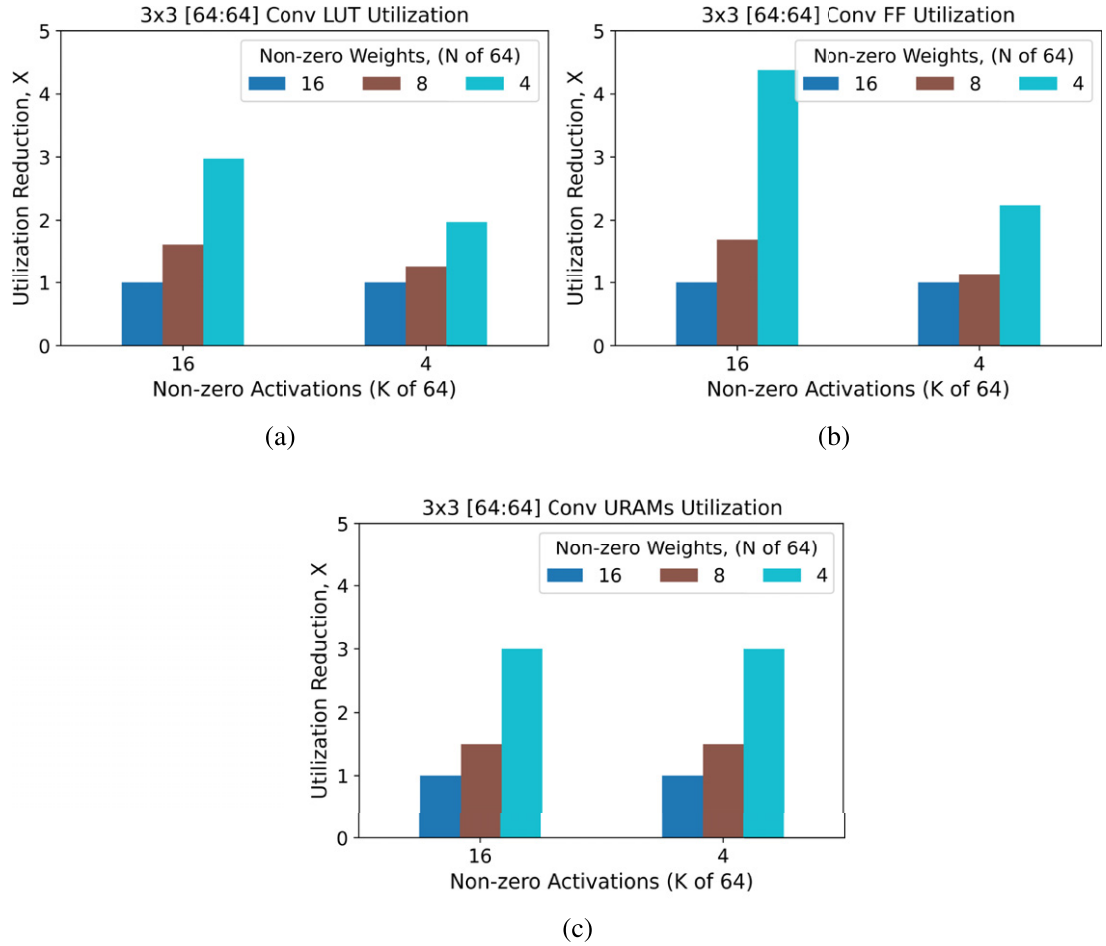


Figure 15. Impact of weight sparsity on resource utilization for 3×3 [64:64] convolution operations for different degrees of activation sparsity for: (a) LUTs, (b) FFs, (c) URAMs (K and N indicate the number of non-zero elements, reduction in utilization relative to $N = 16$).

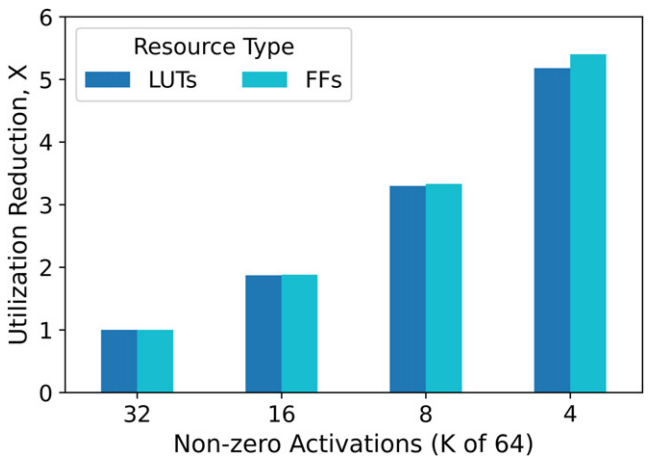
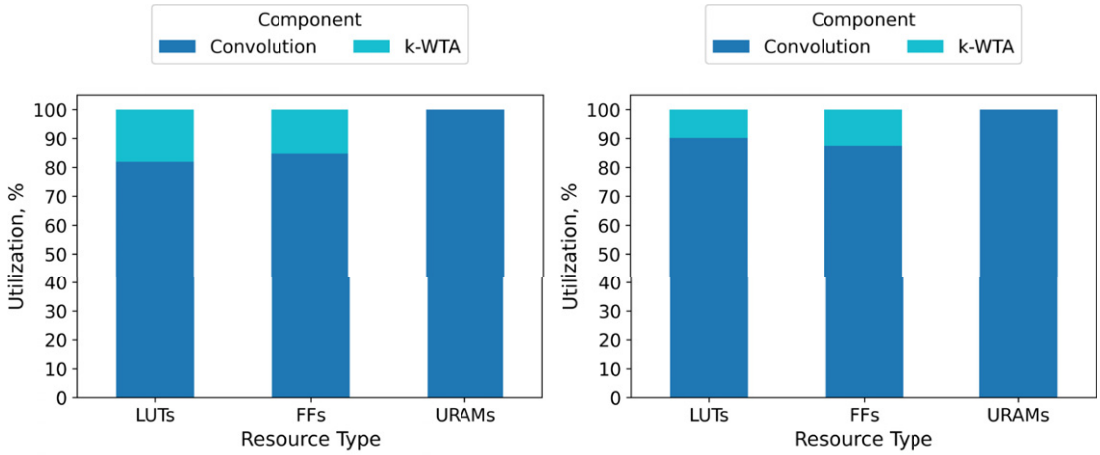


Figure 16. Impact of activation sparsity levels on k-WTA resource utilization (K indicates the number of non-zero elements, reduction in utilization relative to $K = 32$).

6. Discussion

Over the last decade there has been significant attention focused on accelerating DNNs using FPGAs and other architectures, including convolutional networks [18]. In this section we compare our approach to research that is closest to our work, discuss some of the issues that arise in deploying our solution to complex networks, and suggest some directions for future work.



(a) 1x1 Convolution + k-WTA (b) 3x3 Convolution + k-WTA

Figure 17. Total resource utilization of convolution operations and associated k-WTA components (for $N = 8$ and $k = 8$). k-WTA consumes a relatively small percentage of LUTs and FFs, and no URAMS.

6.1. Accelerating sparse DNNs on FPGAs

With Complementary Sparsity we demonstrated that sparse filter kernels can be interleaved such that multiple convolutional kernels are processed simultaneously. A related idea has appeared in [39] where they compact columns of a weight matrix used in a matrix multiply implementation of convolution, which is then processed through a bit-serial systolic array. As such they can reduce the number of MAC operations by a factor of 8 (see below for additional discussion on systolic arrays). They also discuss a process for creating an interleaved weight matrix by incrementally pruning and compacting during training. Although they did not explicitly discuss sparse–sparse optimizations, their compaction technique could potentially be adapted for creating complementary sparse kernels that are compatible with our implementation.

In [90] the authors implement both sparse training and inference on FPGAs. By implementing unstructured pruning and a fine-grained dataflow approach they demonstrate a 1.9× overall improvement in performance due to sparsity. Their unstructured pruning approach is not compatible with Complementary Sparsity, as our approach requires precisely non-overlapping weights. Although we have not demonstrated training speedups, our technique leads to significantly higher inference throughput than their technique.

There have been a number of papers investigating sparse–dense network implementations on FPGAs. Employing either weight [10, 16, 20, 31, 34, 39, 92] or activation sparsity [2], they show it is possible to reduce the number of MAC operations by routing a subset of the dense values to the sparse set of operands at the processing units. This can be done either via multi-ported memories [16] or multiplexor networks [20]. Although reducing the number of multiplies results in power savings, these techniques typically perform only one dot product at a time in each processing unit. Unlike these methods, Complementary Sparsity makes full use of dense activations and sparse weights. Each activation is paired with a corresponding weight value which allows multiple dot products to be performed every cycle and enables fully parallel operations. In addition, Complementary Sparsity provides a path to sparse–sparse implementations.

6.2. Accelerating sparse networks on other platforms

Recognizing that hardware limitations have held back the deployment of sparse networks [29], there has been increasing interest in accelerating sparsity on GPU platforms. It is possible to extract meaningful performance gains with block-sparse kernels by implementing large blocks, of size 32 × 32 or larger [24] with a potential negative impact on accuracy. In [21] CSR based techniques are used to accelerate common DNN networks such as MobileNet [66]. However, the end to end performance gains are limited and restricted to about 1.2× and 2× increase over the dense implementations, respectively. Recently NVIDIA has introduced native support for sparsity in their Ampere [52] architecture. In Ampere there is a limit of 50% sparsity and end-to-end gains are modest at about 1.3× faster than dense. To date, GPU based techniques are limited in their ability to achieve significant performance gains on full networks. In addition they do not provide a path to exploiting both sparse activations and sparse weights.

For sparse–sparse networks, when both weight and activation sparsity are employed [11, 22, 32, 49, 76, 77], it is difficult to efficiently pair the non-zero weight and activations. In [78] the authors discuss modifications that could be made to NVidia's Volta Tensor Core to support unstructured weight and activation sparsity. Many emerging solutions are based around 2D systolic arrays of processing units, on both FPGAs and custom ASIC designs. Here each processing unit performs a check for either matching indices [32, 49] or non-zeros [11] as the weight and activation values are streamed through the systolic array. One concern with this approach is overall performance. With Complementary Sparsity we are able to parallelize computation such that we can execute an entire 1×1 conv block, representing many sparse kernels, in one cycle. Systolic arrays fundamentally require several cycles to flow through the weights and activations. This process would then have to occur for each sparse kernel, thereby limiting their performance gains. Finally, implementing them efficiently often requires the costly development of specialized hardware. In contrast, Complementary Sparsity can deliver performance gains today on currently available hardware.

In our implementation we use k-WTA to achieve activation sparsity. Another approach is to remove entire channels in convolutional layers during training through a structured pruning process [19, 44, 73]. In [19] they notice that activations naturally become sparse during training and use a measure of sparsity to gradually prune channels. In [73] the method is extended to incorporate mixed precision quantization based on activation sparsity and then evaluated on a hardware simulation platform. At a high level our approach is complementary to theirs and can be combined to achieve even greater speedups. k-WTA can be applied to the channel-pruned models as can weight sparsity. However, since the pruned models presumably have less redundancy, larger values of K may be required. Finding the balance between these two complementary approaches is an interesting area of future research.

In this paper we have focused on standard DNNs. Spiking neural networks (SNNs) represent an alternate formalism that offers significant potential for performance improvements [23, 63]. SNNs model neurons using an analog, continuous time, framework. Neurons in SNNs have high temporal sparsity, i.e., they rarely become active. Hardware chips are emerging that exploit this characteristic to create event-based systems that achieve significant energy efficiencies [15, 63]. SNNs historically have been unable to match the accuracy of DNNs on complex tasks, an issue that has held back their wide-scale deployment. This problem is an active area of research, with promising recent results [36, 71], including approaches that attempt to model the temporal sparsity of SNNs in DNN systems [89].

There exist a number of emerging hardware architectures for exploiting sparsity. In [35] the authors review different factors for DNNs, including activation and weight sparsity, and compare a large set of architectures. They suggest that analog crossbar-based architectures represent the most promising direction. This is also investigated in [4] where they review memristors, memristive crossbars, FPGAs, and SNNs for embedded healthcare applications. Another approach is to implement a scatter-compute-gather module to aggregate operands based upon the indices of their non-zero values [77]. In [94] the authors implemented a completely custom memristor-based mixed signal architecture. They demonstrate large performance gains and energy efficiencies for embedded applications using a biologically inspired sparse–sparse learning algorithm.

6.3. Deploying complex sparse–sparse systems

Our results indicate that it is possible to create convolutional networks that exploit both sparse activations and sparse weights. In this article we presented results for an end-to-end speech network as well as the core components used in most convolutional networks. Although these components can form the foundation for building many networks, modern convolutional networks often contain a large number of layers and a variety of structures. In these networks a number of other issues come into play when designing end-to-end systems. These issues, outlined below, are important design considerations in implementing efficient commercial systems based on Complementary Sparsity.

Channel partitioning. The number of channels associated with the convolutional kernels is not constant and often increases for the deeper layers. For example, in a Resnet-50, layers start with 64 channels, but this increases to 2048, as illustrated in figure 11. However, as explicitly noted in [26], the feature map size is reduced correspondingly, keeping the computational requirements roughly constant. In ResNet-50, all convolution operations can be decomposed into groups of 64 dot-products between 64 element vectors, enabling the increasing channel dimension to be handled by the repeated use of our modular [64:64] channel blocks. Our implementation of the k-WTA operator also processes the output of the convolutions in units of 64 elements, enabling the modular construction of the ResNet-50 layers.

Pipeline latency balancing. When balancing the pipeline of an implementation with multiple layers, carefully 'right-sizing' the layers is important to maximize efficiency and minimize resource utilization. This in particularly important in sparse–sparse networks. As discussed in section 5.4 we find that the large gains achieved

by Complementary Sparsity can lead to unexpected bottlenecks in other areas, such as the initial stem layer. For dense implementations, the main option is a choice between serial or parallel implementations. However, for sparse networks, we also have an additional option. Increasing weight and/or activation sparsity for a given layer translates into reductions in compute operations per layer, reducing (serial) latency, and reducing the memory bandwidth required to supply the operands to the computation.

Training accuracy. An important issue, outside the focus of this article, is the ability to train sparse–sparse networks that have sufficient accuracy while retaining high sparsity. As discussed in section 2.2 research in training sparse networks has increased significantly. Of particular interest is the ability to learn the mask itself. Using such adaptive techniques it is now possible to create accurate networks with 90% sparsity on ImageNet [17] and Transformers [13]. Most of the training work has focused on weight sparsity with a few papers focused on activation sparsity. There is relative lack of research on networks that have both forms of sparsity (exceptions are [1, 94]). In some scenarios networks trained without explicit activation sparsity end up with highly sparse activations anyway [8, 19, 20, 33]. This is encouraging because it suggests that sparse activations may naturally be an optimal outcome. We hope the performance results shown in this article will help lead to additional research on sparse–sparse networks.

6.4. Future directions

We have presented an initial set of results on Complementary Sparsity, and there are a number of areas for future research. Our technique currently imposes restrictions on the weights that do not clearly map to neuroscience. Connectivity in the brain is thought to have properties such as a small-world structure and locality [7, 80] that could be very beneficial in hardware implementations. It would be interesting to see if there exist Complementary Sparsity patterns which more closely mimic biological wiring patterns. Our activation sparsity mechanism, *k*-WTA, is directly inspired by neuroscience but is sub-optimal from a coding standpoint. A more optimal code, such as those described in [64, 67] would increase the overall sparsity in activations and increase the impact of sparse–sparse. It would be interesting to see if these more optimal sparse coding methods can be implemented efficiently in hardware.

Another direction is to look beyond convolutional networks and apply Complementary Sparsity to other important architectures, such as Transformers [74], and Deep Recommender systems [91]. This will require a greater focus on linear layers, where it is possible to overlay multiple rows or columns from a layer's sparse weight matrix. A second promising direction is to leverage our FPGA designs to create hardened IP blocks for a variety of ASICs. A third area is to consider the application of Complementary Sparsity to existing hardware platforms beyond FPGAs [68].

Finally, it would be interesting to see if Complementary Sparsity can be used to accelerate the training of sparse–sparse networks. In this paper we showed that the feedforward pass, one of the main steps in training, can be accelerated. Two additional steps are the accumulation of forward gradients and the backward pass itself. The number of non-zero gradients reduces in multiplicative fashion with the weight and activation sparsities, and the loop structure of the backward pass is the transpose of the forward pass. As a result in principle all phases of training should see significant speedup through Complementary Sparsity. Structured pruning steps [19, 44, 73], as discussed earlier, can also be folded in to further accelerate training. These techniques may require highly flexible circuitry, and as such, FPGAs may serve as a better platform than GPUs for developing these ideas. Overall we believe there is significant potential to increase training speed using sparse–sparse approaches.

7. Conclusions

In this article, inspired by the high levels of sparsity in the brain, we investigate the performance benefits of DNNs that exploit both weight and activation sparsity. Using a novel technique that we term *Complementary Sparsity* we show that it enables highly efficient sparse–dense and sparse–sparse networks. Using FPGAs we demonstrate that individual sparse–sparse networks can outperform standard dense DNN networks by over 30×. We further illustrate that sparse–sparse networks can be implemented using far fewer hardware resources than their dense counterparts, and that the resource requirements are inversely proportional to the degree of sparsity. This frugal use of resources allows 5× more networks to be accommodated on an FPGA, delivering a full-chip throughput over 110× higher than the corresponding dense networks. Complementary Sparsity also enables the deployment of DNNs on smaller embedded platforms than previously possible. To our knowledge, we are the first to report such dramatic benefits for both sparse–dense and sparse–sparse networks on FPGAs.

Acknowledgments

We would like to acknowledge the guidance and feedback from Xilinx, especially Prasun Raha, Vamsi Nalluri, Mrinal Sarmah, and Mary Low. We thank Greg Maltz for his critical feedback and help reviewing and editing the article. Additionally, we would like to acknowledge the help and feedback from Jeff Hawkins, Luiz Scheinkman, Celeste Baranski, Enno Wein and the Instigate team (Naira Khurshudyan, Marine Tumasyan, Hmayak Arzumanyan, Hasmik Mantshyan, Armenuhi Petrosyan, Armen Ohanyan, Davit Petrosyan, Anna Sargsyan, Armen Hovhannisyan, and Tatul Yeghiazaryan) over the course of this research.

Data availability statement

The data generated and/or analysed during the current study are not publicly available for legal/ethical reasons but are available from the corresponding author on reasonable request.

ORCID iDs

Subutai Ahmad https://orcid.org/0000-0002-2141-0629

References

[1] Ahmad S and Scheinkman L 2019 How can we be so dense? The benefits of using highly sparse representations (arXiv:1903.11257 [cs.LG])
[2] Aimar A *et al* 2019 NullHop: a flexible convolutional neural network accelerator based on sparse representations of feature maps *IEEE Trans. Neural Netw. Learn. Syst.* **30** 644–56
[3] Attwell D and Laughlin S B 2001 An energy budget for signaling in the grey matter of the brain *J. Cereb. Blood Flow Metab.* **21** 1133–45
[4] Azghadi M R, Lammie C, Eshraghian J K, Payvand M, Donati E, Linares-Barranco B and Indiveri G 2020 Hardware implementation of deep network accelerators towards healthcare and biomedical applications *IEEE Trans. Biomed. Circuits Syst.* **14** 1138–59
[5] Bank R E and Douglas C C 1992 *Sparse Matrix Multiplication Package (SMMP)* (IBM Thomas J Watson Research Division) https://searchworks.stanford.edu/view/6958683
[6] Barth A L and Poulet J F A 2012 Experimental evidence for sparse firing in the neocortex *Trends Neurosci.* **35** 345–55
[7] Bassett D S and Bullmore E 2006 Small-world brain networks *Neuroscientist* **12** 512–23
[8] Beaulieu S, Frati L, Miconi T, Lehman J, Stanley K O, Clune J and Cheney N 2020 Learning to continually learn (arXiv:2002.09571)
[9] Changpinyo S, Sandler M and Zhmoginov A 2017 The power of sparsity in convolutional neural networks (arXiv:1702.06257)
[10] Chen C-F, Oh J, Fan Q and Pistoia M 2018 SC-Conv: sparse-complementary convolution for efficient model utilization on CNNs *2018 IEEE Int. Symp. Multimedia (ISM)* pp 97–100
[11] Chen Q, Huang Y, Sun R, Song W, Lu Z, Fu Y and Li L 2020 An efficient accelerator for multiple convolutions from the sparsity perspective *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **28** 1540–4
[12] Chen Y, Paiton D and Olshausen B 2018 The sparse manifold transform *Advances in Neural Information Processing Systems* vol 31 ed S Bengio, H Wallach, H Larochelle, K Grauman, N Cesa-Bianchi and R Garnett (Curran Associates, Inc.) pp 10533–44
[13] Cohen B 2021 Sparsity without sacrifice: accurate BERT with 10× fewer parameters https://numenta.com/blog/2021/12/13/sparsity-without-sacrifice-accurate-bert-with-10x-fewer-parameters
[14] Cui Y, Ahmad S and Hawkins J 2017 The HTM spatial pooler—a neocortical algorithm for online sparse distributed coding *Front. Comput. Neurosci.* **11** 111
[15] Davies M *et al* 2018 Loihi: a neuromorphic manycore processor with on-chip learning *IEEE Micro* **38** 82–99
[16] Dey S, Chen D, Li Z, Kundu S, Huang K-W, Chugg K M and Beerel P A 2018 A highly parallel FPGA implementation of sparse neural network training *2018 Int. Conf. ReConFigurable Computing and FPGAs (ReConFig)* pp 1–4
[17] Evci U, Gale T, Menick J, Castro P S and Elsen E 2020 Rigging the lottery: making all tickets winners *Proc. 37th Int. Conf. Machine Learning* (Proceedings of Machine Learning Research (PMLR))vol 119) (13–18 July 2020) pp 2943–52 https://proceedings.mlr.press/v119/evci20a.html
[18] Farabet C, Poulet C, Han J and LeCun Y 2009 CNP: an FPGA-based processor for convolutional networks *FPL 09: 19th Int. Conf. Field Programmable Logic and Applications*
[19] Foldy-Porto T, Venkatesha Y and Panda P 2020 Activation density driven energy-efficient pruning in training (arXiv:2002.02949)
[20] Fowers J, Ovtcharov K, Strauss K, Chung E S and Stitt G 2014 A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication *2014 IEEE 22nd Annual Int. Symp. Field-Programmable Custom Computing Machines* pp 36–43
[21] Gale T, Zaharia M, Young C and Elsen E 2020 Sparse GPU kernels for deep learning (arXiv:2006.10901)
[22] Gao C and Delbrück T 2021 Spartus: a 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity (arXiv:2108.02297)
[23] Ghosh-Dastidar S and Adeli H 2009 Spiking neural networks *Int. J. Neural Syst.* **19** 295–308
[24] Gray S, Radford A and Kingma D P 2017 GPU kernels for block-sparse weights *OpenAI* http://openai-assets.s3.amazonaws.com/blocksparse/blocksparsepaper.pdf
[25] Hawkins J and Ahmad S 2016 Why neurons have thousands of synapses, a theory of sequence memory in neocortex *Front. Neural Circuits* **10** 1–13
[26] He K, Zhang X, Ren S and Sun J 2015 Deep residual learning for image recognition (arXiv:1512.03385)
[27] Hoefler T, Alistarh D, Ben-Nun T, Dryden N and Peste A 2021 Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks (arXiv:2102.00554)

[28] Holmgren C, Harkany T, Svennenfors B and Zilberter Y 2003 Pyramidal cell communication within local networks in layer 2/3 of rat neocortex *J. Physiol.* **551** 139–53
[29] Hooker S 2020 The hardware lottery (arXiv:2009.06489)
[30] Intel 2021 Intel® distribution of OpenVINO™ toolkit https://intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html
[31] Jain A K, Omidian H, Fraisse H, Benipal M, Liu L and Gaitonde D 2020 A domain-specific architecture for accelerating sparse matrix vector multiplication on FPGAs *2020 30th Int. Conf. Field-Programmable Logic and Applications (FPL)* pp 127–32
[32] Jamro E, Pabis T, Russek P and Wiatr K 2014 The algorithms for FPGA implementation of sparse matrices multiplication *Comput. Inf.* **33** 667–84
[33] Javed K and White M 2019 Meta-learning representations for continual learning (arXiv:1905.12588)
[34] Jiang C, Ojika D, Patel B and Lam H 2021 Optimized FPGA-based deep learning accelerator for sparse CNN using high bandwidth memory *2021 IEEE 29th Annual Int. Symp. Field-Programmable Custom Computing Machines (FCCM)* pp 157–64
[35] Kendall J D and Kumar S 2020 The building blocks of a brain-inspired computer *Appl. Phys. Rev.* **7** 011305
[36] Kim Y and Panda P 2021 Optimizing deeper spiking neural networks for dynamic vision sensing *Neural Netw.* **144** 686–98
[37] King P D, Zylberberg J and DeWeese M R 2013 Inhibitory interneurons decorrelate excitatory cells to drive sparse code formation in a spiking model of V1 *J. Neurosci.* **33** 5475–85
[38] Knuth D E 1998 *The Art of Computer Programming. Sorting and Searching* 2nd edn vol 3 (Reading, MA: Addison-Wesley)
[39] Kung H, McDanel B and Zhang S Q 2019 Packing sparse convolutional neural networks for efficient systolic array implementations: column combining under joint optimization *Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems, ASPLOS '19* (New York: Association for Computing Machinery) pp 821–34
[40] Kurtz M *et al* 2020 Inducing and exploiting activation sparsity for fast inference on deep neural networks *Proc. 37th Int. Conf. Machine Learning* (13–18 July 2020) https://proceedings.mlr.press/v119/kurtz20a.html
[41] Lagunas F, Charlaix E, Sanh V and Rush A M 2021 Block pruning for faster transformers (arXiv:2109.04838)
[42] Lee H, Ekanadham C and Ng A Y 2008 Sparse deep belief net model for visual area V2 *Advances in Neural Information Processing Systems*
[43] Lennie P 2003 The cost of cortical computation *Curr. Biol.* **13** 493–7
[44] Liu Z, Sun M, Zhou T, Huang G and Darrell T 2019 Rethinking the value of network pruning *Int. Conf. Learning Representations* https://openreview.net/forum?id=rJlnB3C5Ym
[45] Maass W 2000 On the computational power of winner-take-all *Neural Comput.* **12** 2519–35
[46] Majani E, Erlanson R and Abu-Mostafa Y S 1989 On the *k*-winners-take-all network *Advances in Neural Information Processing Systems* pp 634–42
[47] Makhzani A and Frey B 2013 *k*-sparse autoencoders (arXiv:1312.5663)
[48] Makhzani A and Frey B 2015 Winner-take-all autoencoders *Advances in Neural Information Processing* http://papers.nips.cc/paper/5783-winner-take-all-autoencoders
[49] Malik S and Golnari P A 2019 Sparse matrix to matrix multiplication: a representation and architecture for acceleration *2019 IEEE 30th Int. Conf. Application-specific Systems, Architectures and Processors (ASAP)* vol 2160-052X pp 67–70
[50] Markram H *et al* 2015 Reconstruction and simulation of neocortical microcircuitry *Cell* **163** 456–92
[51] Miller J K, Ayzenshtat I, Carrillo-Reid L and Yuste R 2014 Visual stimuli recruit intrinsically generated cortical ensembles *Proc. Natl Acad. Sci. USA* **111** 4053–61
[52] Mishra A K, Latorre J A, Pool J, Stosic D, Stosic D, Venkatesh G, Yu C and Micikevicius P 2021 Accelerating sparse deep neural networks (arXiv:2104.08378)
[53] Mocanu D C, Mocanu E, Stone P, Nguyen P H, Gibescu M and Liotta A 2018 Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science *Nat. Commun.* **9** 2383
[54] Neural Magic 2021 Neural magic deepsparse https://github.com/neuralmagic/deepsparse
[55] Neural Magic 2021 YOLOv3: sparsifying to improve object detection performance https://docs.neuralmagic.com/source/model-pages/cv-detection-yolov3.html
[56] Olshausen B A and Field D J 1996 Emergence of simple-cell receptive field properties by learning a sparse code for natural images *Nature* **381** 607–9
[57] Olshausen B A and Field D J 2004 Sparse coding of sensory inputs *Curr. Opin. Neurobiol.* **14** 481–7
[58] ONNX Runtime developers 2021 ONNX runtime https://onnxruntime.ai/
[59] Paiton D M, Frye C G, Lundquist S Y, Bowen J D, Zarcone R and Olshausen B A 2020 Selectivity and robustness of sparse coding networks *J. Vis.* **20** 10
[60] Pulido C and Ryan T A 2021 Synaptic vesicle pools are a major hidden resting metabolic burden of nerve terminals *Sci. Adv.* **7** eabi9027
[61] Rawat W and Wang Z 2017 Deep convolutional neural networks for image classification: a comprehensive review *Neural Comput.* **29** 2352–449
[62] Reuther A, Michaleas P, Jones M, Gadepally V, Samsi S and Kepner J 2020 Survey of machine learning accelerators *2020 IEEE High Performance Extreme Computing Conf. (HPEC)*
[63] Roy K, Jaiswal A and Panda P 2019 Towards spike-based machine intelligence with neuromorphic computing *Nature* **575** 607–17
[64] Rozell C J, Johnson D H, Baraniuk R G and Olshausen B A 2008 Sparse coding via thresholding and local competition in neural circuits *Neural Comput.* **20** 2526–63
[65] Sainath T N and Parada C 2015 Convolutional neural networks for small-footprint keyword spotting *16th Annual Conf. Int. Speech Communication Association*
[66] Sandler M, Howard A, Zhu M, Zhmoginov A and Chen L-C 2018 MobileNetV2: inverted residuals and linear bottlenecks (arXiv:1801.04381)
[67] Smith E C and Lewicki M S 2006 Efficient auditory coding *Nature* **439** 978–82
[68] Spracklen L, Hunter K and Ahmad S 2021 Poster: 'how can we be so slow?' realizing the performance benefits of sparse networks *Sparsity in Neural Networks: Advancing Understanding and Practice* https://tinyurl.com/so-slow
[69] Strubell E, Ganesh A and McCallum A 2019 Energy and policy considerations for deep learning in NLP (arXiv:1906.02243)
[70] Tang R and Lin J 2017 Deep residual learning for small-footprint keyword spotting (arXiv:1710.10361)
[71] Tavanaei A, Ghodrati M, Kheradpisheh S R, Masquelier T and Maida A 2019 Deep learning in spiking neural networks *Neural Netw.* **111** 47–63
[72] Thompson N C, Greenewald K H, Lee K and Manso G F 2020 The computational limits of deep learning (arXiv:2007.05558)

[73] Vasquez K, Venkatesha Y, Bhattacharjee A, Moitra A and Panda P 2021 Activation density based mixed-precision quantization for energy efficient neural networks *2021 Design, Automation Test in Europe Conf. Exhibition (DATE)* pp 1360–5
[74] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser L and Polosukhin I 2017 Attention is all you need *31st Conf. Neural Information Processing Systems (NIPS 2017)* (Long Beach)
[75] Vinje W E and Gallant J L 2000 Sparse coding and decorrelation in primary visual cortex during natural vision *Science* **287** 1273–6
[76] Wang D, Shen J, Wen M and Zhang C 2019 Efficient implementation of 2D and 3D sparse deconvolutional neural networks with a uniform architecture on FPGAs *Electronics* **8** 803
[77] Wang X, Wang C, Cao J, Gong L and Zhou X 2020 WinoNN: optimizing FPGA-based convolutional neural network accelerators using sparse winograd algorithm *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **39** 4290–302
[78] Wang Y, Zhang C, Xie Z, Guo C, Liu Y and Leng J 2021 Dual-side sparse tensor core *Proc. 48th Annual Int. Symp. Computer Architecture, ISCA '21* (IEEE Press) pp 1083–95
[79] Warden P 2018 Speech commands: a dataset for limited-vocabulary speech recognition (arXiv:1804.03209)
[80] Watts D J and Strogatz S H 1998 Collective dynamics of 'small-world' networks *Nature* **393** 440–2
[81] Weliky M, Fiser J, Hunt R H and Wagner D N 2003 Coding of natural scenes in primary visual cortex *Neuron* **37** 703–18
[82] Xie S, Girshick R, Dollár P, Tu Z and He K 2017 Aggregated residual transformations for deep neural networks *Proc. 30th IEEE Conf. Computer Vision and Pattern Recognition, CVPR 2017* vol 2017 (Institute of Electrical and Electronics Engineers Inc.) pp 5987–95
[83] Xilinx 2020 Vivado design suite user guide, high-level synthesis https://xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf
[84] Xilinx 2021 Vitis high-level synthesis user guide: HLS pragmas https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas
[85] Xilinx 2021 Alveo U200 and U250 data center accelerator cards data sheet https://xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf
[86] Xilinx 2021 Xilinx Vitis AI https://xilinx.com/products/design-tools/vitis/vitis-ai.html
[87] Xilinx 2021 Zynq UltraScale+ MPSoC data sheet https://xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf
[88] Yoshimura Y, Dantzker J L M and Callaway E M 2005 Excitatory cortical neurons form fine-scale functional networks *Nature* **433** 868–73
[89] Yousefzadeh A and Sifalakis M 2021 Training for temporal sparsity in deep neural networks, application in video processing (arXiv:2107.07305)
[90] Zhang J, Chen X, Song M and Li T 2019 Eager pruning: algorithm and architecture support for fast training of deep neural networks *Proc. 46th Int. Symp. Computer Architecture, ISCA '19* (New York: Association for Computing Machinery) pp 292–303
[91] Zhang S, Yao L, Sun A and Tay Y 2019 Deep learning based recommender system: a survey and new perspectives *ACM Comput. Surv.* **52** 1–38
[92] Zhu C, Huang K, Yang S, Zhu Z, Zhang H and Shen H 2020 An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **28** 1953–65
[93] Znamenskiy P, Kim M-H, Muir D R, Iacaruso M F, Hofer S B and Mrsic-Flogel T D 2018 Functional selectivity and specific connectivity of inhibitory neurons in primary visual cortex 294835 *bioRxiv Preprint* https://biorxiv.org/content/early/2018/04/04/294835.1
[94] Zyarah A M, Gomez K and Kudithipudi D 2020 Neuromorphic system for spatial and temporal information processing *IEEE Trans. Comput.* **1** 1099