Synapse Compression for Event-Based Convolutional-Neural-Network Accelerators

Lennart Bamberg, Arash Pourtaherian, Luc Waeijen, Anupam Chahar, and Orlando Moreira

Abstract—Manufacturing-viable neuromorphic chips require novel compute architectures to achieve the massively parallel and efficient information processing the brain supports so effortlessly. The most promising architectures for that are spiking/event-based, which enables massive parallelism at low complexity. However, the large memory requirements for synaptic connectivity are a showstopper for the execution of modern convolutional neural networks (CNNs) on massively parallel, event-based architectures. The present work overcomes this roadblock by contributing a lightweight hardware scheme to compress the synaptic memory requirements by several thousand times—enabling the execution of complex CNNs on a single chip of small form factor. A silicon implementation in a 12-nm technology shows that the technique achieves a total memory-footprint reduction of up to 374 \times compared to the best previously published technique at a negligible area overhead.

Index Terms—CNN, compression, dataflow, event-based, hardware accelerator, near-memory compute, neuromorphic, sparsity, spiking.

I. INTRODUCTION

Convolutional neural networks (CNNs) achieve state-of-the-art computer-vision performance. Neural networks are not only more accurate but also easier to develop than hand-crafted techniques. Thereby, they enable a plethora of applications, opening new markets for digital systems.

One of the challenges CNNs pose is their large computational complexity and the massive parameter count (e.g., ResNet50 requires over 23M parameters and 4 \text{GFLOPS/frame} [1]). Therefore, CNN inference is relatively slow and power-hungry on traditional general-purpose compute architectures, as it requires a large number of expensive dynamic random access memory (DRAM) accesses and arithmetic operations. Thus, high frame-rate and yet low-power CNN inference is challenging on constrained edge devices. Consequently, accelerators for efficient CNN inference at the edge are one of the largest research topics in industry and academia today.

Some of the most promising processor architectures for efficient CNN inference are dataflow/event-based with a self-contained memory organization [2], [3], [4]. The neuromorphic dataflow model of computation, where execution is triggered on the arrival of data exchanged through events rather than by a sequential program, enables scaling to massive core counts at low complexity. Moreover, dataflow cores are typically lighter in hardware implementation than more traditional von-Neumann processors.

Another advantage of event-based systems is that they can exploit the large degrees of activation/firing sparsity in modern deep neural networks. This is especially true for leak-integrate-and-fire (LIF) or sigma-delta neuron models, which exploit temporal correlation in the input and the feature maps to increase sparsity [3], [5]. Thus, event-based processing delivers massive compute performance at low power consumption and area.

A self-contained memory architecture overcomes DRAM access bottlenecks, as all parameters required during execution are stored in local on-chip memories, distributed over the cores. This avoids slow and power-hungry DRAM accesses entirely. The need for no external DRAM makes the system not only more efficient power and performance wise, but also cheaper and smaller. This further enhances the suitability of event-based systems for embedded applications.

Due to these vast promises, a range of event-based and self-contained architectures was designed by academia [6], [7], and global semiconductor companies like Intel and IBM [2], [3], [4], [8]. However, many of the existing chips only allow a limited neuron count and hence do not support the inference of modern sophisticated CNNs (e.g., ResNet11 [1] or MobileNet [9]). A fundamental problem is the memory requirement for synapses, i.e., the weighted connections between the neurons. Published event-based architectures, store the synapses in (hierarchical) look-up tables (LUTs), resulting in memory requirements that scale with the total neuron count. This leads to high memory requirements for neural networks with a large neuron and synapse count. Consequently, architectures like Intel’s Loihi and IBM’s TrueNorth are not capable to execute modern, sophisticated CNNs as shown in Table I.

Sophisticated CNN models are still out of reach for existing event-based architectures, limiting their usability today.

This work overcomes this severe limitation. It contributes a technique that systematically exploits the translation invariance of CNN operations to make the synaptic-memory requirements independent of the neuron count. This results in extreme compression. Our technique employs two lightweight...
TABLE I
Neuron and Synapse Count of CNNs and the Maximum Capabilities of Event-Based Architectures From Major Semiconductor Companies

| Neurons | Synapses | CNNs | Architecture |
|---------|----------|------|--------------|
| PilotNet [10] | 0.2 M | 27 M | | |
| MobileNet [9] | 4.4 M | 0.5 B | | |
| ResNet50 [11] | 9.4 M | 3.8 B | | |
| IBM [4] | 1.1 M | 0.3 B | | |
| Intel [2], [8] | 1.1 M | 0.1 B | | |

Fig. 1. Taped-out 144-neuron-core SoC with SIMD execution.

hardware blocks, a programmable event generator (PEG) and an event-to-synapse unit (ESU). The PEG computes the synaptic connectivity of thousands of neurons—grouped in a neuron population—using a single instruction word, the axon. The ESU fills the synaptic connections with weights. Weights can be shared among all neurons of a population, reducing the memory requirements for parameters.

Our proposed technique was taped-out through the 144-neuron-core GrAl-VIP shown in Fig. 1. In the 12-nm system-on-chip (SoC) of size 60mm², our technique exhibits negligible implementation cost, while providing overall memory compression rates of over 300× for modern CNNs compared to the best-published reference technique.

Intel recently revealed its new self-contained and event-based deep-neural-network (DNN) accelerator Loihi 2. It is capable to run Nvidia’s PilotNet CNN in 30mm² using its 7-nm technology [2]. The same network (i.e., without any extra optimizations) requires only 3 out of 144 cores on our chip using a larger technology node at 2× the die area. This demonstrates the practical value of the proposed technique, as it allows event-based cores to run much more sophisticated/complex CNNs in a temporal-sparse fashion on a single low-cost, self-contained, and event-based accelerator.

The rest of this article is structured as follows. Related work is discussed in Section II. Section III includes the preliminaries covering CNNs and their sparse execution on event-based accelerators. The proposed technique is derived in Section IV. Section V includes discussions and experimental results. Finally, a conclusion is drawn.

II. RELATED WORK

This section reviews related work on event-based architectures and memory-footprint compression. A wide range of massively parallel, self-contained, event-based architectures have been proposed (e.g., [2], [3], [4], [6], [7], [8], [11]). Most only support neuromorphic neuron models, while GrAI Matter Lab’s Neuronflow architecture [3] and the Loihi2 [2] support also traditional DNNs—executed in a time-sparse manner to reduce the compute requirements [3], [5]. Such DNNs are today more accurate and easier to train.

Reducing the memory footprint of neural networks is an actively researched topic. Compression is vital for self-contained architectures, as non-fitting CNNs cannot be supported. Traditional architectures have looser memory constraints due to the usage of external DRAM. However, even here compression has great advantages, as it reduces latency- and power-hungry DRAM accesses [12].

One approach to reducing memory requirements at the application level is to use a lighter-weight CNN (e.g., [9], [13]). Another possibility is to execute a given CNN at a lower precision. A common technique is to run the network in 8-bit integer arithmetic [14], but even binary network implementations exist [15]. Han et al. proposed to iteratively prune weights with a small magnitude (effectively setting them to 0), followed by incremental training to recover accuracy [12]. In combination with 8-bit integer arithmetic and weight entropy-encoding, weight pruning can achieve compression rates larger than 10× for modern CNNs [12].

The approaches mentioned above do not only reduce the memory footprint, but also the computational complexity of the CNN inference. Also, since these techniques transform the neural network itself, they are effective for most compute architectures. The drawback is that the techniques are lossy, i.e., they typically entail some accuracy loss that is partially recoverable by retraining [12], [14].

Synapse-compression techniques for event-based systems, such as the one proposed in this work, are orthogonal to the lossy techniques discussed above. Hence, both compression types can be applied in conjunction and studied independently. Since synapse-compression techniques are lossless, they do not affect the network prediction/output. Hence, these techniques, tailored for event-based systems, entail no loss in network accuracy. Thus, they can be applied post-training.

Early event-based architectures used a single routing LUT in the source to look up all synaptic connections of a neuron on its firing [6]. This approach results in many events and large memory requirements for DNNs with many neurons, [N], as the number of required bits is of complexity O(√[N] log₂([N])). Moradi et al. and Davies et al. independently, found that the events can be reduced and the synaptic-memory compressed to O([N]√ log₂([N])) through a hierarchical, tag-based routing scheme [6], [8]. This hierarchical routing scheme found application in Intel’s original Loihi architecture and the DYNAPs architecture.

The technical note of the recently released Loihi 2 claims that the synaptic memory footprint for CNNs has been compressed again by up to 17× [2]. Unfortunately, details of the approach are (at the time of writing) not fully disclosed. However, at a reduction of up to 17×, the synaptic memory requirements must still be at least of complexity O([N]). Thus, larger CNNs would remain unfeasible for Loihi 2.

The idea of generating a synaptic memory structure for event-based CNN inference that scales independently of [N]
(required to run today’s complex CNNs), has been tried for the SpiNNaker chip [16]. However, the existing method is not generic enough. It only supports simple convolution layers. Other common CNN operations such as stride-2 convolutions, polling, and depth-wise convolutions are all not supported. Moreover, the technique does not allow the cutting of feature maps (FMs) in multiple fragments for compute or memory load-balancing. Hence, the technique was only demonstrated in Ref. [16] to work for a tiny CNN with five standard convolution layers and a maximum feature shape of 28 × 28. Thus, it cannot be implemented for modern deep CNNs which include many more layer types. Moreover, modern CNNs have large FM dimensions and strongly heterogeneous compute requirements per layer which make FM-cuts inevitable. Thus, the problem addressed by this work remained unsolved until this point.

III. PRELIMINARIES

A. Neural Networks

In neural networks for image processing, a neuron population is represented by a three-dimensional matrix, \( \mathbf{P} \in \mathbb{R}^{D \times W \times H} \). The last two dimensions determine the two-dimensional (2D) Cartesian-grid position to correlate a neuron with a location in space; the first coordinate indicates the feature (or channel) that is sampled. Thus, a single neuron population represents multiple features. Here, the number of channels of such a multi-channel FM is denoted as \( D \). The 2D submap of channel \( c \) (\( \mathbf{P}_{c,x,y} \)) has the dimensions of a single feature, defined by a tuple: width, \( W \), and height, \( H \).

The raw input image represents the input population of the network. For example, a 300 × 300 RGB image is represented by a 3 × 300 × 300 \( \mathbf{P} \) matrix, where \( \mathbf{P}_{c,x,y} \) is the pixel value for color channel \( c \) at location \((x, y)\). Starting with this input population, new FMs are extracted out of the existing FMs successively (feed-forward structure). One such extraction step is symbolized as a network layer in a dataflow graph structure. The most intuitive form is a fully connected layer, in which each neuron in the destination FM, \( \mathbf{P}^+ \), is a function of all neurons in a source FM, \( \mathbf{P} \)

\[
P^+_{c,x,y} = \sigma \left( \sum_{i=0}^{D-1} \sum_{j=0}^{W-1} \sum_{k=0}^{H-1} \mathbf{W}_{c,x,y,i,j,k} \mathbf{P}_{i,j,k} + b_c \right).
\]

Here, \( b_c \) is the bias of the neurons of feature \( c \), and \( \mathbf{W} \) is the weight matrix. A DNN can be implemented in its traditional form or as a spiking leak-integrate-fire neural network (LIF-NN). In the first variant, \( \sigma() \) is a non-linear activation function to enable non-linear feature extraction. Often, a rectifier/ReLU is used, clipping all negative values to zero. The ReLU function exhibits low hardware complexity. Neuromorphic LIF-NNs differ from traditional DNNs through a more complex, stateful activation, \( \sigma() \), with periodic state-leakage.

A major problem with fully connected layers is the large number of synaptic weights which determine the compute and memory requirements. For example, extracting ten 300 × 300 features from a 3 × 300 × 300 FM requires 243M weights and multiply-accumulate (MAC) operations.

This can be overcome by exploiting spatial locality [17]. For a meaningful feature extraction, it is not needed to look far in XY to glean relevant information for one XY coordinate of the extracted feature. Thus, outside a relative range of XY size (\( KW, KH \) —denoted as the receptive field—all synapses between neurons are removed. This drastically reduces the compute and memory requirements for such locally connected 2D layers.

Moreover, network filters should respond similarly to the same patch, irrespective of its XY location in the source FM. This principle is called translation invariance [17]. Hence, the relative weights can be independent of the XY location of the destination neuron; i.e., weights can be reused for all neurons of the same output channel. This further reduces the memory requirements.

The regular convolution operation in a CNN adds both, spatial locality and translation invariance compared to fully connected layers. This results in the following formula for a convolution layer with a kernel size of \( KW \times KH \)

\[
P^+_{c,x,y} = \sigma \left( \sum_{i=0}^{D-1} \sum_{j=0}^{W-1} \sum_{k=0}^{H-1} \mathbf{W}_{c,i,j,k} \mathbf{P}_{i+x,j+y+k} + b_c \right).
\]

The XY size of the extracted FM in a convolution operation is the XY size of the source FM minus \( KW - 1 \) in width and \( KH - 1 \) in height. This is the number of times the convolution kernel fits in the source FM, illustrated in Fig. 2.a.

Paddind rows and columns of zeros around the source is done to control the size of the output FM. To achieve an output of the same size as the input (i.e., to preserve the feature resolution), one must pad \((KW-1)/2\) rows of zeros at the top and bottom of the source FM, and \((KH-1)/2\) columns at the left and right. Fig. 2.b illustrates this zero-padding.
Instead of explicitly padding the source FM in a preprocessing step, the effect of zero padding can be expressed implicitly, by changing the source-FM indices in (2)

\[ P_{c,x,y}^+ = \sigma \left( \sum_{i=0}^{D-1} \sum_{j=0}^{K'-1} \sum_{k=0}^{H-1} W_{c,i,j,k} P_{i,x+j-XP,y+k-YP} \right) + b_c \]

with \( P_{i,x,y} = 0 \) for \( x < 0 \) or \( y < 0 \). (3)

Here, \( XP \) and \( YP \) are the rows/columns of zeros padded at the left and the top of the source FM, respectively.

Kernel striding and source upsampling are other ways of controlling the dimensions of the extracted features. With a stride of two, the kernel is not shifted by one step for every generated output point but by two, resulting in extracted features with a 50\% smaller resolution. When the source is upscaled to increase the resolution of the extracted features, zeros are filled between the upsampled source values followed by a trainable convolution on the upsampled map.

B. Event-Based Architectures

In this section, we outline the conceptual idea of event-based compute architectures for efficient CNN inference. These architectures exhibit massive parallelism with many identical cores connected through a network-on-chip (NoC) [2, 6].

To each of the \( C \) cores, one or more neuron populations, \( P_i \), are statically mapped. Near-memory computing is applied in the architectures. Thus, the entire memory is distributed equally over the cores, and the neural parameters (e.g., weighted synapses) are stored in the cores to which the respective neurons are mapped. The dynamic variables, computed by one core (i.e., the activated neuron states; see (3)), and needed as inputs by other cores for the calculation of neuron states, are exchanged through events over the NoC. Hence, there are no coherence problems. This allows the removal of higher/shared memory levels, improving latency and power consumption.

A drawback is that local on-chip memories have limited capacity. Thus, the static/compile-time mapping of neuron populations to cores must satisfy hard memory constraints; i.e., the number of neurons and synapses that can fit in a core is limited by the core’s memory capacity. If a neuron population \( P_i \) of the CNN has too many parameters or neurons, it needs to be split/cut into multiple smaller populations before core assignment.

The arrival of data in form of an event triggers the compute (dataflow execution). A buffer stores the received events for first-in-first-out (FIFO) processing. Event processing multiplies the event value with the weights of the synapses between the source neuron and the destination neurons mapped to the core. The results are accumulated in the destination neuron states. After accumulation, activated neuron states are sent through events to all synaptically connected neurons in the following layers. Since these neurons can be mapped to any core in the cluster, events can also be injected into the local event queue instead of going out on the NoC.

Note that above we describe how more recent architectures (considered in this article) work. In some earlier architectures, the multiplications of activations by synaptic weights are executed at the source (i.e., before event transmission). Here, event processing at the destination only requires the addition of the event value to the target neuron state [6]. This scheme results in more events for neurons with multiple synapses. Thus, it is no longer applied in modern architectures.

1) Sparsity Exploitation in Event-Based Architectures: During DNN execution, no events are generated for zero-valued activations. Skipping zero-valued events induces no neuron-state accumulation error. Thus, each zero in an FM reduces the compute and communication requirements—saving energy and compute resources [18]. This is an advantage of event-based architectures compared to more traditional architectures. Particularly for modern CNNs with large degrees of activation sparsity (i.e., zero-valued activations), it gives event-based architectures a power-performance advantage. Standard CNNs using ReLU activations exhibit an activation sparsity of over 50\%, as a ReLU clips all negative values to zero [19]. Thus, event-based processing improves the power-delay product by about 4\% at no accuracy loss.

Event-based architectures exhibit even larger gains by implementing CNNs as LIF-NNs or sigma-delta neural networks (SD-NNs) to increase sparsity. A LIF-based CNN has the same synaptic connectivity as a standard CNN implementation [20]. However, LIF-NNs exhibit a more complicated firing behavior—expressed by \( \sigma(\cdot) \) in (3). This increases sparsity but also makes back-propagation training hard. Consequently, LIF-NNs cannot yet compete accuracy-wise with standard CNNs for complex applications.

To address this, SD-NN implementations increase the neural sparsity by transforming temporal correlation of neuron potentials between subsequent inferences into event sparsity [5]. This is particularly efficient for time-continuous input data such as video streams. By keeping memory for a persistent accumulator state for each neuron, the temporal deltas in the activations can be exchanged/processed instead of absolute values. Still, the same inference results as a standard CNN implementation are obtained as state accumulation is a linear operation. Hence, SD-NNs exhibit no training or accuracy issues. Any standard CNN can be executed as an SD-NN at no accuracy loss and without retraining. Still, SD-NNs improve sparsity noticeably. The temporal deltas of the activations exhibit more zeros than the absolute activation maps, due to the present temporal correlation—especially if activations are quantized [5]. Thereby, SD-NNs increase the gains of event-based processing without sacrificing any network accuracy.

The proposed, as well as the reference techniques, work for all three described CNN variants (standard, LIF, and SD). Thus, in the remainder of this article, we use the term firing neuron to describe a neuron of any type for which the synaptic connectivity has to be triggered due to a non-zero activation, activation-delta, or spike.

2) Synaptic-Memory in Existing Architectures: Event-based accelerators need information on the synaptic connectivity of a firing neuron to identify which neurons to update with which
weights. Mathematically, the connectivity of neurons can be described as a directed graph \( G = (\mathcal{N}, \mathcal{S}) \), where \( \mathcal{N} \) and \( \mathcal{S} \) are the sets of neurons and synapses, respectively. Each synapse, \( s_i \in \mathcal{S} \), can be described by a tuple \((n_{\text{src}}, n_{\text{dst}}, w)\), where \((n_{\text{src}}, n_{\text{dst}}) \in \mathcal{N}^2\) is the synapse’s source and destination neuron, respectively, and \(w\) is the synaptic weight.

In early event-based accelerators, LUTs store the synapse information, mapped to the cores together with the neurons. On the firing of a neuron, the addresses of the destination neurons together with the weights/parameters are looked-up, and for each pair an event carrying the firing value multiplied by the weight is emitted towards the destination neuron. If \( F \) is the average number of outgoing synapses per neuron, the required memory bits to store the connectivity (i.e., synapses without the weights) for \(|\mathcal{N}|\) neurons in this scheme is

\[
Mem_c = |\mathcal{N}| F \log_2 (|\mathcal{N}|).
\]  

(4)

With \( B \)-bit weights, the parameter requirements become

\[
Mem_p = |\mathcal{N}| FB.
\]  

(5)

For modern neural networks with many neurons, a LUT results in high memory requirements. As an illustrative example, let us investigate the last 3 \( \times \) 3 convolution of the ResNet50 CNN [1], with \( 512 \times 7 \times 7 \) neurons in the source and destination. Storing the connectivity of all neurons in this layer in a LUT requires at least 110MB. The weights would need an additional 201MB at an 8-bit precision.

Moreover, the standard LUT-based approach has the disadvantage that it results in one event per synapse—so \( F \) events per firing neuron. A hierarchical/two-stage LUT layout reduces the number of events and the memory requirements [6], [8]. For each spiking neuron, only the addresses of the destination cores are looked up in the first routing table at the source. Thus, only one event is generated per destination core instead of one per destination neuron (while hundreds of neurons are mapped to a single core). An event contains a \( K \)-bit tag, stored also in the LUT at the source, and the firing value. In the destination core, the weights and the target neurons are selected based on the tag through a second LUT.

For a maximum of \( F_{\text{max}} \) in-going synapses per neuron, the tag-width must be at least \( \log_2 (F_{\text{max}}) \). With \( C \) cores, \( M \) neurons per core, and an average of \( F \) in-going synapses per neuron, the number of bits to store the connectivity of \(|\mathcal{N}|\) neurons reduces to

\[
Mem_c = Mem_{\text{core}} + Mem_{\text{dist}}
= |\mathcal{N}| \left( \frac{F}{M} \log_2 (F_{\text{max}} C) + \log_2 (F_{\text{max}} F) \right),
\]  

(6)

in the best case. However, the memory requirement still scales with \(|\mathcal{N}|\). Thus, the scheme is still impractical for modern CNNs. Analyzing again the last 3 \( \times \) 3 convolution layer of ResNet50, the synaptic memory requirement is still at least 167MB.

IV. TECHNIQUE

In the following, we propose a technique to compress the synapses for CNNs such that the memory requirements no longer scale with the neuron count but just with the population count.
multiplied by the respective weight to any neuron state in the range of the transposed kernel. The final neuron states for the inference are obtained after all events have been processed.

To obtain the same final neuron states in the event-based implementation as in the standard implementation, the weight arrangement for the broadcasted kernel must be transposed in XY (e.g., top-left weight becomes bottom-right) compared to the regular view. In the regular view, the destination neuron’s XY coordinate is the upper-left anchor point of the kernel in the source-FM. Thus, in the event-based variant, the XY location of the broadcasted source neuron is the bottom-right anchor point of the transposed kernel in the destination FM. Equation (3) on page 8 shows that padding shifts the XY coordinate of the bottom-right anchor-point of the transposed kernel by the padding amount, \((X P, Y P)\).

Thus, on the firing of a neuron on channel \(c_{src}\) with XY coordinate \((x_{src}, y_{src})\), all neurons in the destination Three-dimensional (3D) FM must be updated whose X and Y coordinate is in \([x_{src} - KW + XP, x_{src} + XP]\) and \([y_{src} - KH + YP, y_{src} + YP]\), respectively. In that range, the XY-transposed kernel weights \(W_{i,j,k,l}|i=c_{src}\) are scaled (i.e., multiplied) by the firing value and added to the respective neuron states. For some firing neurons nearby the edges of an FM, part of the broadcasted transposed kernel does not overlap with the destination FM, as illustrated in Fig. 4.b. Such kernel parts located outside the FM boundaries do not result in neuron updates. Thus, they are skipped in the ESU.

The pseudo codes in Algorithms 1 and 2 describe the required functionality of the PEG and the ESU, respectively. Hardware implementation is done most efficiently through simple state machines. The PEG of the source core (i.e., the core with the firing neuron) calculates the top-left anchor point for the transposed convolution kernel in the destination FM, \((x_{min}, y_{min})\). As outlined above, this requires a subtraction of \((KW - XP - 1, KH - YP - 1)\) from the XY coordinates of the firing neuron \((x_{src}, y_{src})\). Since the padding and kernel shape are constant at CNN-inference-time, this is done in the PEG at once by adding a signed offset pair to the neuron’s coordinates

\[
X_{off}, Y_{off} = \left(-KW + XP + 1, -KH + YP + 1\right). \tag{7}
\]

Storing compile-time calculated offset values rather than the kernel shape and the padding in an axon, saves additions/subtractions in the PEG for each firing neuron as well as bits in the axon.

The outgoing event generated by the PEG contains the computed \((x_{min}, y_{min})\) coordinates, which can contain negative values if the top-left anchor-point is located outside the FM range. Moreover, the firing value, \(v\), and the channel of the firing neuron, \(c_{src}\), are added to the event body. Finally, events need the NoC address of the core to which the target population is mapped, \(AD_{c}\), as well as a destination-population identifier, \(ID_{p}\). The later enables that multiple populations can be mapped onto the same core. These last two parameters are static and thus part of the axons.

Each source population has an independent axon for each directly connected destination population. On each firing, the PEG must process all these axons to generate events towards all populations.

At the far end, the ESU takes the event and selects the required compile-time-transposed 3D kernel, \(W_{i,j,k,l}|i=c_{src}\) based on \(c_{src}\) and \(ID_{p}\) from the event. It applies the weights scaled by the event value on the neurons with the top-left anchor point taken from the event \((x_{min}, y_{min})\). It skips kernel parts that are outside the population boundaries. The iteration order is channel-first (HWC). This minimizes the amount of required continue statements to at most \(KH \cdot KW\), reducing the complexity of a hardware implementation.

The proposed technique shares kernels and axons among all neurons of a population. Hence, besides axons, neurons, and weights, we need two more types of memory words: population descriptors and kernel descriptors. A population descriptor contains the 3D neuron population shape, the neuron type incl. the activation function, the start address of the state values, and the number of axons linked to the population. A kernel...
The descriptor contains the kernel shape, its weight width, and a memory pointer for the weights.

B. Support for FM Cutting

A multi-channel FM in the original/pre-mapped CNN graph can have so many parameters or neurons that it cannot be mapped onto a single core due to memory or compute constraints. Thus, the proposed technique supports the cutting of FMs into several smaller neuron populations, mapped onto different cores.

An FM cut that divides the channels results in a reduction in the number of neurons as well as weights mapped to each core. On the other hand, a cut in the XY space of the features requires all populations to contain all weights. This is due to the translation invariance of the convolution operation. Hence, a cut reducing only the X or Y size of the individual populations increases the overall memory requirements, as shared weights are duplicated (i.e., mapped onto multiple cores). Nevertheless, our approach supports both: cuts in channel coordinate C and XY.

The rationale is that, for high-resolution features, it may happen that even a single channel does not fit into a core, making XY cuts inevitable. Also, addressing limitations can result in inevitable XY cuts. For example, for 8-bit population-width/height fields, a high-resolution FM has to be split into populations with a width and height of at most 255.¹

Fig. 5 shows an example where the source and destination FM are cut into four and three pieces, respectively.

The example helps to understand the following mathematical derivation of the FM-fragmentation technique.

Let the shape of the original, unfragmented \( i \)th FM be \( (D_i, W_i, H_i) \). The FM is cut into \( M \) disjoint populations \( P_{i1}^{(i)}, P_{i2}^{(i)}, \ldots, P_{iM-1}^{(i)} \). The shape of the \( j \)th fragment is \( (D_j, W_j, H_j) \) and it contains all neurons from the original FM with:

\[
C_{0,i}^{(j)}, C_{0,i}^{(j + 1)}, \ldots, C_{0,i}^{(j + D_j - 1)} \in C_{0,i}^{(j)} + D_j, X_{0,i}^{(j)}, X_{0,i}^{(j + 1)}, \ldots, X_{0,i}^{(j + D_j - 1)} \in X_{0,i}^{(j)} + W_j, Y_{0,i}^{(j)}, Y_{0,i}^{(j + 1)}, \ldots, Y_{0,i}^{(j + D_j - 1)} \in Y_{0,i}^{(j)} + W_j.
\]

¹Such fragments can be mapped to the same core to share weights.

For each fragment of the destination FM, we then calculate the location of this upper-left anchor point of the transposed kernel relative to their XY start-point in the original FM

\[
x_{\text{min}} = x_{\text{min,orig}} - X_{0,\text{dist}} = x_{\text{src}} + X_{0,\text{src}} - K W + X P + 1
\]

\[
y_{\text{min}} = y_{\text{min,orig}} - Y_{0,\text{dist}} = y_{\text{src}} + Y_{0,\text{src}} - K H + Y P + 1 - Y_{0,\text{dist}}.
\]

These \((x_{\text{min}}, y_{\text{min}})\) coordinates, together with \(c_{\text{src,orig}}\), are sent via an event to the core of the destination-FM fragment. All parameters but the coordinates of the firing neuron in (9) are constants at run-time. Thus, supporting FM cuts just needs an adjustment of the \( X \) and \( Y \) offset values calculated at compile time, plus an additional offset value for the channel coordinate

\[
C_{\text{off}} = C_{0,\text{src}}
\]

\[
X_{\text{off}} = X_{0,\text{src}} - K W + X P + 1 - X_{0,\text{dist}}
\]

\[
Y_{\text{off}} = Y_{0,\text{src}} - K H + Y P + 1 - Y_{0,\text{dist}}.
\]

Note that the offset values for \( X \) and \( Y \) can be negative here. Thus, they are signed values. The channel offset is always positive. Hence, it can be an unsigned value.

For each population/FM-fragment pair that is connected by at least one synapse, an axon is needed. However, the number of axons remains minimal, as different populations are mapped onto different cores. Thus, different axons are required anyway due to the different core addresses.

Each core only stores the weights for the channels that are mapped to it. Thus, the indices of the weight matrix are relative to the first channel of the respective fragment; i.e., when neurons of the channels 4 to 7 are mapped onto a core, \(W_{0,0,0,0}\) stored in the local core memory is the upper-left kernel weight for source channel 0 and destination channel 4 of the original transposed.

---

**Fig. 5.** Cutting of the source and the destination FM.
weight matrix. This fragmentation of the weight matrix is done entirely at compile time.2

In summary, besides an additional axon parameter $C_{off}$ added to the channel coordinate, nothing is added to the PEG hardware. The ESU must not be modified at all.

There is an optional extension to the PEG. A hit detection to filter out events that an ESU decodes in zero synapses as the kernel does not overlap at all with the destination population. Consider in Fig. 5, that the neuron at position $(0, 1, 1)$ instead of $(0, 0, 1)$ fires. The PEG depicted by Algorithm 1 sends an event to the core, destination-FM fragment 0 is mapped onto. This event has an $x_{min}$ value of 4. However, the population only contains neurons with $x \in [0, 3]$. Thus, event decoding by the ESU at the receiving side results in no activated synapse (i.e., continue in line 3 of Algorithm 2 always triggered). The system functions correctly without an extension. However, transmitted events yielding no activated synapses after decoding increase the energy and processing requirements unnecessarily. Such events are best filtered out by the PEG in the source already.

This PEG power-performance optimization is achieved by calculating not only the XY start-coordinate of the projected kernel (i.e., the top-left anchor point) but also the first X and Y coordinates outside the range. These coordinates are $(x_{min}, y_{min}) + (KW, KH)$. One can then check if the kernel overlaps with at least one neuron of the population through simple Boolean operations. If it does not, no event is generated. To enable this optimization, axons must contain the kernel shape as well as the width and height of the destination population.

Hit detection in the channel coordinate is not required, as all channels in the XY kernel range are updated at the destination, irrespective of the channel of the firing neuron. Thus, if cutting in XY is the rare exception, hit-detection support might be skipped to reduce cost. However, hit detection has a reasonable hardware cost. Hence, hit-detection support is beneficial in most cases. Algorithm 3 shows the pseudo code for the PEG fragmentation and hit detection.

Algorithm 3: PEG for a Regular Convolution With Support for FM Fragmentation and Empty-Event Skipping.

Require:Firing neuron & value: $x_{src}, y_{src}, c_{src}, v$
Require:Axon set of population: $A$
1: // grayed hit-detection only needed with many XY cuts
2: for $(X_{off}, Y_{off}, C_{off}, W, H, KW, KH, AD_c, ID_p) \in A$
3: \hspace{1em} $x_{min}, y_{min}, c_{src} = (x_{src}, y_{src}, c_{src}) + (X_{off}, Y_{off}, C_{off})$
4: \hspace{1em} $x_{max}, y_{max} = (x_{min}, y_{min}) + (KW, KH)$
5: \hspace{1em} if $(x_{min} < W)$ & $(x_{max} > 0)$ & $(y_{min} < H)$ & $(y_{max} > 0)$ then
6: \hspace{2em} gen_event($AD_c, ID_p, x_{min}, y_{min}, c_{src}$)
7: \hspace{1em} end if
8: end for

C. Down- and Upsampling Support

In the following, we extend the PEG-ESU pair to support upsampling and strided convolutions. A kernel stride of $N$ in a convolution implies that the anchor point of the kernel in the regular/output-centric view (cf. Fig. 4.a) is advanced by $N$ in X between two generated output points. Once the end of a line is reached, the anchor point is advanced by $N$ in Y while $X$ is reset to 0. Consequently, a kernel stride of $N$ in a convolution results in the same extracted FM as applying the convolution kernel regularly (i.e., with a stride of 1), followed by downsampling the resulting FM by a factor $N$ in X and Y.3 This "destination-downsampling view" of a convolution stride is used to derive our technique.

The pseudo-code of the ESU with support for programmable kernel striding is shown in Algorithm 4.

We add a field $SL$, containing the $\log_2$ of the applied kernel stride, to the kernel descriptor. Through $SL$, the ESU knows which rows and columns to skip as they are gone after downsampling such that there is no need to update a neuron state. Thus, our stride implementation based on destination downsampling results in no compute overhead as accumulation for neurons removed by downsampling is skipped.

The PEG still behaves as it would for stride-1 convolutions. Hence, the widths $W$ and heights $H$ of the destination fragments in the axons are the ones that would be obtained for a stride of 1. Thus, the physical width and height of the true/final downsamped FM shifted left by $SL$ ($W_i \ll SL$ and $H_i \ll SL$) are stored in the axon as $W$ and $H$. This inverts the downsampling effects. Also, the offset values are adjusted to account for the destination-FM downsampling

$$X_{off} = X_{0,src} - KW + XP + 1 - (X_{0,dst} \ll SL)$$

$$Y_{off} = Y_{0,src} - KH +YP + 1 - (Y_{0,dst} \ll SL).$$  (11)

All this only affects constant parameters stored in the axons. Thus, the PEG hardware remains unchanged.

3Downsampling by a factor $N$ means here simply keeping only every $N^{th}$ row and column.
Also, the ESU initially behaves as for a regular convolution. Thus, also here the true population width and height shifted left by \(SL\) are stored in the descriptor as \(W\) and \(H\), respectively. The effective downsampling after convolution due to the kernel stride is considered by extending the continue conditions. In detail, if the for-loop iteration over the transposed kernel is at a row or column removed by the downsampling, synapse generation is skipped. Rows and columns that are not skipped are only the ones with \(X\) or \(Y\) coordinates which are 0 modulo \(2^SL\). For non-skipped kernel parts, the downsampling of the \(XY\) coordinate is applied for the selection of the neuron to be updated. This down-sampling is realized in hardware at a low cost through a right shift by \(SL\).

Adding stride support does not add noticeably to the complexity of the ESU. First, checking that a binary signal is not 0 in modulo \(2^SL\) just requires a logical OR operation of the last \(SL\) bits. Moreover, all common CNNs only use strides of 1 or 2. Thus, we propose to keep \(SL\) a 1-bit value such that a stride of 1 (regular convolution) or 2 can be applied in a single step. Larger strides can still be implemented by inserting dummy/identity layers with another stride of two until the needed downsampling rate is reached. This follows the architectural principle of optimizing for the common case, with non-optimized support for the uncommon case. With a 1-bit \(SL\) field, checking that a signal is not 0 modulo \(2^SL\), requires only a single AND-gate with the \(SL\) bit and the signal’s least-significant bit (LSB) as the two binary inputs. Moreover, with a 1-bit \(SL\) value, a shift left by \(SL\) has a negligible implementation cost. It can be synthesized in a single 2-to-1 multiplexer with \(SL\) as the select signal. Thus, the hardware cost for adding stride support is low.

Next, we discuss the PEG changes to support upsampling the source before a convolution operation is applied. This is required for example for fractionally-strided convolutions [21]. Upsampling requires filling zeros between the upsampled source values. Source up sampling is realized in the PEG by shifting the \(XY\) coordinates of the firing neuron left by the \(\log_2\) of the upsampling factor \(US\) before adding the offset values. Also, the \(X\) and \(Y\) offsets calculated at compile-time must consider the upsampling of the source

\[
X_{\text{off}} = (X_{\text{off}} < US) - KW + XP + 1 - (X_{\text{off}} < SL)
\]

\[
Y_{\text{off}} = (Y_{\text{off}} < US) - KH + YP + 1 - (Y_{\text{off}} < SL).
\]

The resulting final pseudo code of the PEG with upsampling support is shown in Algorithm 5.

### Algorithm 5: PEG With Upsampling Support.

**Require:** Firing neuron & value: \(x_{\text{src}}, y_{\text{src}}, c_{\text{src}}, v\)

**Require:** Axon set of population: \(A\)

1: // grayed hit-detection only needed with many XY cuts
2: for \((X_{\text{off}}, Y_{\text{off}}, C_{\text{off}}, W, H, KW, KH, US, AD, IC, ID_p) \in A\) do
3:   \(x_{\text{src}}, y_{\text{src}} = (x_{\text{src}}, y_{\text{src}}) < US // XY up-sampling\)
4:   \(x_{\text{min}}, y_{\text{min}}, c_{\text{src}} = (x_{\text{src}}, y_{\text{src}}, c_{\text{src}}) + (X_{\text{off}}, Y_{\text{off}}, C_{\text{off}})\)
5:   \(x_{\text{max}}, y_{\text{max}} = (x_{\text{min}} < \text{min} + (KW + KH)\)
6:   if \((x_{\text{min}} < W) \& (x_{\text{max}} > 0) \& (y_{\text{min}} < H) \& (y_{\text{max}} > 0)\) then
7:     gen_event(IC, ID_p, x_{\text{min}}, y_{\text{min}}, c_{\text{src}})
8: end if
9: end for

The derivation of our proposed technique only considered some of the layer types/operations found in modern CNNs. Thus, this section first discusses how the technique is used to implement the various additional layer types. Afterwards, we discuss silicon-implementation results. Finally, we determine the compression gains for state-of-the-art CNNs through experimental results.

### V. DISCUSSIONS & EXPERIMENTAL RESULTS

In this section, the proposed technique is evaluated through experimental results. Furthermore, we discuss the universal usability of the technique. As stated before, the proposed synapse compression technique is lossless. Hence, it has no impact on the accuracy or prediction quality of the CNNs to be executed. Thus, this section does not discuss network accuracies as the accuracies reported in the original papers of the investigated CNNs can be matched.

![Example Image](image-url)
the intermediate positions between the learned weights. Ideally, this is paired with an efficient zero-weight skipping technique as it is done in our silicon implementation presented in the next subsection. Note that zero skipping furthermore enables an additional drastic weight compression through pruning. Thus, zero-skipping support is typically anyway desired [22].

**Depthwise & Grouped Convolutions:** A depthwise convolution is a lightweight filtering, applying a single convolution per input channel (i.e., no channel mixing; 
jth output channel only depends on 
jth input channel instead of all). In a grouped convolution, each output channel depends on a small set of 
Dgroup input channels. Depthwise and grouped convolutions can be implemented with the PEG-ESU pair by splitting the original source and destination FM into many FMs of depth 1 (depthwise) or 
Dgroup (grouped), with regular convolution operations between the source-destination pairs covering the same channel(s).

**Average & Max. Pooling:** Pooling operations are implemented as strided depthwise convolutions. For example, an average pooling over non-overlapping 2 × 2 windows is simply a stride-2, depthwise, 2 × 2 convolution. Thereby, all four weights are 
1/4. Max. pooling has the same connectivity. The only difference is that the weights are 1 rather than 
1/PoolingWindowSize. Thus, only the neuron-update routine is changed for max. pooling, not the proposed synapse compression technique.

**Dense Layers:** A fully-connected layer from 
N to 
N+ neurons is the same as a 1 × 1 convolution between two FMs of shape 
N × 1 × 1 and 
N+ × 1 × 1. This is naturally supported by the proposed techniques. In this case, each neuron forms an individual feature. Thus, all neurons in the destination population depend on all neurons in the source population.

**Flattening & Global Pooling:** Flattening an FM of shape 
D × W × H followed by a dense layer with 
N neurons (i.e., destination FM shape is 
N × 1 × 1) is implemented as a single regular convolution with a kernel XY shape equal to 
(W, H) with 
N output channels. Thereby, the two layers are merged into a single operation which saves compute and events. Global pooling is realized in the same fashion, only that a depthwise connectivity must be implemented.

**Nearest-Neighbor & Bilinear Interpolation:** An upsampling together with an interpolation requires combining the upsampling feature with an untrainable depthwise convolution representing the interpolation between the upsampling points.

**Add and Multiply Layers:** Add layers and pointwise multiply layers require two individual source FMs of the same shape pointing to the same destination FM. Between the two individual source FMs and the destination FM the same pointwise synaptic connectivity is required. This is implemented as the connectivity of a depthwise 1 × 1 convolution with a weight of 1 shared by both source FMs.

### B. Silicon Implementation

A variant of the PEG-ESU pair supporting all layer types is implemented in our next-generation GrAI architecture, taped-out in the Taiwan Semiconductor Manufacturing Company (TSMC)12 fin field effect transistor (FinFET) technology. The heart of the 60mm² SoC (shown in Fig. 1 on page 3) is a self-contained, event-based accelerator, supporting standard DNNs, LIF-NNs, and SD-NNs with over 18 million neurons.

The individual cores are arranged as an XY mesh connected through an NoC. By using an 8-bit relative core address (4-bit X, 4-bit Y), a neuron can have a synaptic connection with any neuron in the same core or any of the nearest 255 cores, while enabling the architecture to theoretically scale to an arbitrary number of cores. Our pre-product chip for embedded applications has 144 cores per die, but chip tiling—enabled by the relative addressing scheme—still allows for meshes with larger core counts.

Due to the self-contained nature, all neurons, weights, and other parameters mapped to a core must fit into the on-chip static random access memories (SRAMs). Each core contains 256kB of unified local memory (with 64-bit words and 15-bit addresses) that can be allocated freely to weights, neuron states, and connectivity.

To utilize the memory optimally, the architecture uses 8 bits to describe the width and height of a neuron population and 10 bits to describe the depth. This is enough for modern CNNs. Larger populations can still be realized through FM cutting. The addressing allows a single population to fit over 1 million neurons. Still a neuron-population descriptor containing width, height, depth, neuron type, activation function, axon count, and start address of the population in the memory fits in one 64-bit word. Note that in our silicon implementation axons and states are stored in a continuous block. Thus, we can store only the axon count instead of a 15-bit pointer in the population descriptor, which saves bits.

The kernel width, 
KW, and height, 
KH, in the axons and kernel descriptors are 4-bit numbers. The selection of the bit widths used for kernel width and height are important design choices. Together with the width of an XY coordinate (here 8 bit), they determine the width of most data-path components in ESU and PEG. Moreover, they determine the maximum range of the for loops implemented through the control path.

We found that rarely a CNN has a kernel size larger than 16. Hence, the used 4 bits do not only yield a low hardware complexity but also an efficient support of most CNNs. Note that, due to the flexibility provided by the offset fields 
Xoff and 
Yoff, any larger kernel size can still be implemented through multiple axons and weight matrices. For example, a 32 × 16 convolution is realized as a 16 × 16 convolution paired with another 16 × 16 convolution between the same FMs for which the 
Xoffset is increased by 16. The upsampling and stride fields are 3 bits and 1 bit wide, respectively—enough for all common CNNs. Nevertheless, larger up- and downsampling factors can be realized by adding dummy layers with an identity weight matrix.
fragmentation by channels is not limited, which is strongly preferred anyway.

Given all that, also an axon and a kernel descriptor fit comfortably into a single 64-bit word. We made the design choice to have a kernel descriptor per channel of the source FM instead of only one per FM. Thus, $c_{src}$ of the event is used besides $D_{src}$ to select the kernel descriptor. A kernel descriptor contains the 3D kernel shape ($KD$, $KW$, $KH$), and a pointer to the start address of the resulting 3D sub-weight-matrix, $W_{sub,c_{src}} = W_{i,j,k,l|l=c_{src}}$, in the memory. Moreover, the kernel descriptor contains information required for zero-weight skipping and weight quantization, which is not relevant to the synapse-compression technique.

Multiple kernel descriptors per layer enable weight reuse among different source channels as well as different quantization and pruning schemes for each sub-weight matrix. Also, it overcomes the need to calculate the start point in the weight matrix the descriptor points at. All weights in the sub-matrix are traversed on an event processing. The drawback is a slightly reduced compression as we need at least $C_{src}$ rather than one kernel descriptor per layer.

Each neuron can have a persistent state for SD-NN or LIF-NN inference, or can dynamically allocate a temporary accumulator state for regular DNN implementations. Dynamic state allocation allows mapping CNNs with larger neuron counts, as state entries can be shared among neurons whose accumulation phases are mutually exclusive. A state is a 16-bit float number for maximum precision, while a weight can be quantized to 8 bits or even less with no noticeable performance loss using adaptive-float quantization [23].

Other than that, the architecture supports multi-threading, single-instruction-multiple-data (SIMD) execution, as well as a wide range of neuron types and activation functions. Also, ESU and PEG still support absolute/non-compressed synapses for smaller network layers with irregular connectivity.

1) Impact on Power, Performance, and Area: In the following, we summarize the results of the physical implementation in the 12-nm technology. The ESU and PEG take in total less than 2% of the area of a processor node. Only a marginal fraction ($\ll 1\%$) of the power consumption is due to the ESU or the PEG, which are also not found to be part of the critical timing path. Note that the implemented ESU and PEG include many additional features beyond the scope of this article. Examples are support for multi-threading, weight quantization plus zero skipping, and non-compressed synaptic connectivity. Thus, the hardware costs of the proposed technique are negligible.

In line with previous work [12]—and despite applying the synapse compression—area, power, and timing of the chip are dominated by the on-chip memories used to store the parameters, neuron states, and connectivity. The memories take over 70% of the core area as shown in Fig. 1 on page 3. Thus, the power and area savings of the proposed technique are determined by the memory savings, investigated in the next subsection.

C. Compression Results

Through a Python tool, the memory savings of the proposed synapse-compression technique are quantified. The tool calculates the synaptic memory requirements of a given CNN for the proposed method, a simple LUT approach, and the hierarchical-LUT approach [6], [8]. For the proposed technique—other than for the reference techniques—the memory requirements of a layer increase with the number of fragments the layer is split in. Thus, the tool calculates the memory requirements for the proposed scheme considering FM cuts that ensure that the total memory footprint of each resulting fragment is below 256kB, the single core limit of our chip. By considering the real (non-beneficial) required FM cuts for the proposed technique, we report realistic rather than optimistic gains for the proposed technique.

To enable a detailed analysis, the memory requirements are categorized into neurons, connectivity, and parameters. The reported values for connectivity are the bits required to store the connectivity between the neurons, but they do not include the weights of the synaptic connections. Thus, for the proposed technique, it includes axons, kernel descriptors, and population descriptors. The category parameters includes the weights. The last section, neurons, reports the memory allocated for neuron states. Throughout this section, we assume that a persistent 16-bit state is allocated for each neuron in a population. Reporting the memory usage in these categories allows to access the gains of pattern/weight sharing and axon-based synapse computation in isolation. The former technique reduces the parameter requirements, while the latter reduces the connectivity requirements.

In the experiments, the tag width in the reference hierarchical-LUT technique is 15 bits. This is the minimum tag width required by the scheme to be able to map all CNNs analyzed in this section, as the analyzed networks have neurons with a fan-in (i.e., number of incoming synapses) of up to $7^2 \cdot 512$. Thereby, we present best-case values for the hierarchical-LUT reference technique, which in this form is only able to support CNNs with synaptic fan-ins of at most 32k.

The unified memory can theoretically fit $2^{17}$ 16-bit neuron states in 256kB of unified memory. Still, for the naive LUT technique, we consider only 15-bit neuron addresses, as none of our experiments resulted in a core memory that is occupied more than 25% by neurons for this technique. Hereby, we ensure a fair comparison also for the second reference technique. Table II summarises again the bit widths of all data types for the three techniques.

By considering persistent 16-bit states for each neuron but 8-bit weights/parameters, we present pessimistic compression
gains for the proposed technique. Our technique heavily compresses the parameters and the connectivity compared to the reference techniques but not the neurons. Generally, both, the effectively required bits per neuron and per parameter can be reduced further by compression techniques orthogonal to the one proposed in this work (e.g., non-persistent states, weight pruning, entropy coding, channel pruning) [12], [24]. However, reducing the memory footprints of neurons and parameters would automatically increase the compression gains of our proposed technique. The heavily compressed connectivity in our scheme would have a higher impact in this case.

Through this pessimistic experimental setup, we ensure that the reported gains of our technique are realistically achieved in practically all applications of event-based architectures. On the downside, the setup results in pessimistic total memory requirements, why the reported values allow no final conclusion on the mappability of the analyzed CNNs on our chip. Thus, the reported memory requirements are above what is realistically achieved in our chip, which also supports many parameter and neuron compression schemes orthogonal to the proposed technique.

1) Small CNNs: We start by analyzing a relatively small CNN for today’s standards, PilotNet [10]. It is a 9-layer CNN used for autonomous steering. This example is discussed in-depth first, as it allows us to understand the advantages as well as challenges revealed by the proposed technique. Also, PilotNet is the network that was demonstrated to fit in Intel’s newest event-based architecture, Loihi 2, making it a good reference benchmark. In the following subsection, we will validate the findings for more modern (and much larger) networks.

We calculate the memory requirements of each layer for the proposed and the two reference techniques. The results are plotted in Fig. 6.

They show that the memory requirements of the two other published techniques are dominated by connectivity. For the two reference techniques LUT and hierarchical LUT 74.1% and 65.2% of the memory requirements are due to connectivity, respectively. The parameters are the second dominant factor for the memory requirements of the reference techniques, contributing about 25% to 35% to the total memory usage. For non-quantized 16-bit weights, the parameter requirements would compete with the connectivity requirements.

Despite allocating a 16-bit state per neuron, the neurons only account for 0.2% to 0.3% of the memory requirements for the reference techniques. This proves that synapse compression—as done by the proposed technique—is the most effective way to reduce the memory requirements for event-based architectures at no accuracy loss.

The proposed synapse-computation methodology drastically reduces the connectivity and parameter requirements by a factor of 15.6 k × and 107 ×, compared to the best previously published method. Intel reports for the upcoming Loihi 2 architecture a CNN synapse compression of up to 17 ×. The synapses, i.e., the combination of parameters and connectivity, are compressed by the proposed technique by a factor of 305 ×. This means an additional compression by a factor of 18 × on top of the 17 × published by Intel [2].

Connectivity, from being the biggest contributor to the memory requirements, effectively becomes negligible with the proposed technique, contributing only 0.7% to the memory usage. Parameters, on the other hand, are still dominant (53.7%), despite the weight-sharing scheme. Neurons start to become an important factor as well (45.6%). This is only due to the persistent neuron states. For stateless neurons, the memory requirements are weight dominated, demonstrating a high memory efficiency of our architecture also for stateless neurons and low-bit-width weight quantization.

Fig. 6 shows that the memory requirements of the first layers are dominated by neurons with the proposed technique (for the considered stateful neurons). Later layers are parameter dominated. This strongly justifies the need for a unified memory where the section sizes for parameters and neurons can be freely assigned, as done in our taped-out chip. The growing ratio of neuron over parameter requirements is explained with the structure of a typical CNN. Early layers have a larger XY resolution than later ones. Stride-2 convolutions, non-padded convolutions, and pooling layers successively reduce the width, W, and height, H, of the FMs. On the other hand, the channel count, D increases with the depth of the layer in most CNNs.

Since the neuron and weight counts of a regular convolution layer are $D \cdot W \cdot H$ and $D_{\text{sec}} \cdot D \cdot KW \cdot KH$, respectively, weight requirements increase and neuron requirements decrease with the layer depth.

\[ \text{As of writing this article, only maximum compression results are available for Loihi 2 [2]. The details of the used technique are not fully disclosed. Thus, more comparisons with Loihi 2 cannot be drawn.} \]
Most layers of the PilotNet CNN only take a small portion of the memory of a core with our technique. Consequently, a mapping aiming at a minimized core count, showed that the proposed synapse-compression technique allows fitting the full CNN without any further optimization in as few as 3 out of the 144 cores. With the two reference techniques, the same mapping experiment shows a core-count requirement that is at least 101× higher.

D. State-of-the-Art CNNs

The reference neuromorphic architectures do not support sophisticated CNNs. The reason is that such complex networks do not fit in the local memories due to the large synaptic memory requirements. The proposed technique overcomes this limitation through its large compression rates, as shown in the following.

We analyze the core and memory requirements for four of today’s most popular CNNs: Google’s MobileNet [9], ResNet50 [1], Darknet53 [25], and ResNet101 [1]. The first network is designed to run on mobile devices and is thus more lightweight than the other three designed for less constrained devices such as graphical processing units (GPUs). Hence, MobileNet has far fewer neurons and synapses than the other three analyzed CNNs (but still many more than PilotNet). In particular, ResNet101 is an extremely complex network, while ResNet50 and Darknet53 lay somewhere in the middle. We investigate these CNNs to quantify the gains of the proposed technique for a broad range of CNNs. Also, we show that the proposed compression technique enables the execution of even complex CNNs in a single self-contained chip.

The experimental setup here is the same as for the PilotNet analysis in the previous subsection.

Table III includes the results for all analyzed networks. The results show that, with the previously published techniques, mapping CNNs more complex than PilotNet is out of reach. For each network, the memory requirements for connectivity is already far above the reasonable on-chip memory limit. This change with the proposed technique, which compresses the memory requirements for connectivity compared to the best reference technique by up to 15k×. In fact, for all networks, the connectivity requirements become negligibly small. Even for the complex ResNet101 CNN, only 2.18MB are required for connectivity. This is far below the on-chip memory limit of modern embedded systems. Thus, in combination with other supported compression techniques such as weight pruning, low bit-width quantization, and stateless execution, even ResNet101 is mappable for our architecture.

Due to the nature of the experimental setup, parameters and neurons dominate after applying the proposed synapse compression technique. Still, our weight-sharing approach enables us to bring the parameter requirements down by up to 231×.

Despite the pessimistic experimental setup, our technique achieves overall memory compression rates in the range from 123× to 374× compared to the published state-of-the-art. Another advantage of the proposed technique is that it has larger gains for complex networks. The memory compression rate for the lightweight CNNs MobileNet and Pilotnet are the lowest.

For the complex CNNs Darknet53 and ResNet101—for which a high compression rate is crucial—the overall compression is about 300×.

The reason is that complex CNN layers have much more channels, resulting in many more synapses per neuron. With an increasing synapse count per neuron, the compression rate of the proposed technique grows, as our technique only requires one axon to describe the connectivity between two neuron populations, irrespective of the synapse count in between. This increases the mappability of complex CNNs on our event-based architecture.

Finally, we want to draw again an abstract comparison with Loihi 2, for which the technology brief reports a synapse compression of up to 17× for CNNs without disclosing the technique fully [2]. Compared to the stated 17× best-case reduction by Loihi2, we achieve another factor of at least 18× (PilotNet) and up to 37× (DarkNet53) on top. This demonstrates the clear advantage of the proposed technique over not only the published state-of-the-art but also over upcoming commercial architectures.

VI. CONCLUSION

This work presents a technique to compress the synaptic memory in neuromorphic event-based massive-multicore architectures for efficient CNN inference. The technique is based on two lightweight hardware blocks substituting memory costly look-up tables. Our proposed technique has been tape-out in a 144-core event-based CNN accelerator using a 12-nm technology. The hardware overhead of the proposed technique was found to be negligible. Nevertheless, it demonstrates compression rates for various modern CNNs ranging from 123× to 374×.
A systematic combination of the proposed technique with orthogonal weight and neuron compression schemes is left for future work. With such techniques, we will be even able to run extremely complex CNNs on a small form-factor chip in a temporal-sparse event-based fashion at low power consumption, latency, and cost.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770–778.

[2] “Taking neuromorphic computing with Loihi 2 to the next level – Technology brief,” 2021. Accessed: Oct. 21, 2021. [Online]. Available: https://download.intel.com/newsroom/2021/new-technologies/neuromorphic-computing-loihi-2-brief.pdf

[3] O. Moreira et al., “Neuronflow: A hybrid neuromorphic-dataflow processor architecture for AI workloads,” in Proc. IEEE 2nd Int. Conf. Artif. Intell. Circuits Syst., 2020, pp. 01–05.

[4] F. Akopyan et al., “TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 34, no. 10, pp. 1537–1557, Oct. 2015.

[5] P. O’Connor and M. Welling, “Sigma delta quantized networks,” 2016, arXiv:1611.02024.

[6] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, “A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs),” IEEE Trans. Biomed. Circuits Syst., vol. 12, no. 1, pp. 106–122, Feb. 2017.

[7] S. B. Furer, F. Galluppi, S. Temple, and L. A. Plana, “The SpiNNaker project,” Proc. IEEE, vol. 102, no. 5, pp. 652–665, May 2014.

[8] M. Davies et al., “Loihi: A neuromorphic manycore processor with on-chip learning,” IEEE Micro, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018.

[9] A. G. Howard et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” 2017, arXiv:1704.04861.

[10] M. Bojarski et al., “End to end learning for self-driving cars,” 2016, arXiv:1604.07316.

[11] “BrainChip,” Accessed: Nov. 03, 2022. [Online]. Available: https://brainchip.com/

[12] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” 2015, arXiv:1510.00149.

[13] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2017, pp. 1492–1500.

[14] B. Jacob et al., “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2018, pp. 2704–2713.

[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: Imagenet classification using binary convolutional neural networks,” in Proc. Eur. Conf. Comput. Vis., Springer, 2016, pp. 525–542.

[16] T. Serrano-Gotarredona, B. Linares-Barranco, F. Galluppi, L. Plana, and S. Furer, “ConvNets experiments on SpiNNaker,” in Proc. IEEE Int. Symp. Circuits Syst., 2015, pp. 2405–2408.

[17] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Dive into deep learning,” 2021, arXiv:2106.1342.

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

[19] Z. Zhu, A. Pourtaherian, L. Waeijen, L. Bamberg, E. Bondarev, and O. Moreira, “ARTS: An adaptive regularization training schedule for activation sparsity exploration,” in Proc. IEEE 25th Euromicro Conf. Digit. Syst. Des., 2022, pp. 415–422.

[20] Y. Cao, Y. Chen, and D. Khosla, “Spiking deep convolutional neural networks for energy-efficient object recognition,” Int. J. Comput. Vis., vol. 113, no. 1, pp. 54–66, 2015.

[21] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional networks for biomedical image segmentation,” in Int. Conf. Med. Image Comput. Comput. Assist. Interv., Springer, 2015, pp. 234–241.

[22] S. Han et al., “EIE: Efficient inference engine on compressed deep neural network,” ACM SIGARCH Comput. Architect. News, vol. 44, no. 3, pp. 243–254, 2016.

[23] T. Tambe et al., “AdaptivFloat: A floating-point based data type for resilient deep learning inference,” 2019, arXiv:1909.13271.

[24] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in Proc. IEEE Int. Conf. Comput. Vis., 2017, pp. 1389–1397.

[25] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” 2018, arXiv:1804.02767.