




# Smaller together: Groupwise Encoding of Sparse Neural Networks

Elias Trommer , Bernd Waschneck , Akash Kumar 

**Abstract**—With the drive towards ever more intelligent devices, neural networks are deployed on smaller and smaller systems. For these embedded microcontrollers, memory consumption becomes a significant challenge. We propose multiple encoding schemes that convert the decrease in parameter counts, achieved through unstructured pruning, into tangible memory savings. We first discuss a sparse encoding scheme for arbitrary sparse matrices that is based on encoding offsets from a predicted even spacing of elements in a row. The compression rate of this scheme is improved further by identifying groups of elements which can be encoded with even lower overhead. Both methods are combined into a hybrid scheme which encodes arbitrary sparse matrices with low overhead, while allowing for parallel access to multiple elements in a row at once—an important feature for using the scheme on the latest generation of microcontrollers with parallel SIMD capabilities. Our scheme compresses sparse models to below the size of their dense counterparts for sparsities as low as 30% and reduces model size by 32.4% and 26.4% at less than one percentage point of accuracy loss for two convolutional neural network tasks in our evaluation.

**Index Terms**—neural networks, pruning, microcontrollers, embedded systems, SIMD

## I. INTRODUCTION

**B**OTH programmers and users have come to rely on neural networks (NNs) for solving problems where handwritten algorithms fail. Across an enormous number of fields, NNs are used for their capacity to learn functions from data [1]. Due to their large consumption of resources, NNs often remain confined to large and centralized high performance computing (HPC) clusters. This approach introduces issues through a lack of privacy, the need for device connectivity and increased latency. An alternative approach is tinyML, which brings models directly to a user’s device and uses local NNs to process the data where it originates [2]. Even these small models present a significant challenge for microcontroller units (MCUs), the smallest of compute systems. While desktop and HPC systems offer an abundance of memory at minimal latency through hierarchical layers of caches, MCUs are only equipped with small amounts of Flash and Static random-access memory (SRAM) memory, usually in the range of KiB to a few MiB [3].

When memory is precious, increasing the performance per parameter becomes vital. Unstructured Pruning leverages the redundancy in NN weights by identifying those weights that do not contribute much to overall performance and replacing them with zeros. Previous research shows that a significant number of parameters can be pruned without a noticeable impact on the NN’s performance [4]. The remaining weights, however, pose a new challenge: the previously regular structure becomes

highly irregular after pruning. Encoding the positions of non-zero elements in a way that translates the reduction of effective parameters into real-world savings requires a data structure that can quickly reconstruct the position of non-zero elements at runtime, while keeping its own memory footprint as low as possible.

Scientific sparse matrix encoding schemes optimize for extremely high sparsities, but create significant overhead when they are deployed with the much lower sparsity seen from unstructured pruning [5]. Scientific applications also put throughput over memory savings. Schemes for NN accelerators, in contrast, parallelize the extraction of rows but lack the capability to employ intra-row parallel single instruction, multiple data (SIMD) operations present in the latest generation of commercial off-the-shelf (COTS) embedded systems [6].

We aim to fill this gap with a combination of different approaches. We present a parallel encoding scheme for sparse NNs and embedded SIMD instructions that was first introduced in [7]. We provide an extension to this work by addressing a major shortcoming: a low compression rate, particularly in the case of low to medium levels of sparsity. For many NN architectures, these lower levels of sparsity maintain most of the original network’s performance. This makes them most relevant for decreasing the size of NNs while providing performance similar to that of baseline models. In practice, these sparsities are rarely used since the overhead of sparse formats negates most or all of the potential gains from omitting zero values. As stated above, an important shortcoming of many encoding schemes is the lack of parallelism from making the recovery of non-zero element indices a recursive operation. Our proposed scheme, in contrast, increases the compression rate, but also maintains parallelism in retrieving element indices.

The novel contributions of this work are:

- A groupwise encoding scheme that encodes groups of equidistant elements. By grouping non-sequential elements based on a shared property, encoding overhead is not incurred for each element, but rather for each group of elements. This spreads the encoding overhead out across a larger element pool, which reduces the overhead per active element. We provide a greedy search algorithm that can identify these groups of elements from a given sparse matrix.
- A hybrid encoding scheme that combines the groupwise encoding with delta-Compressed Storage Row (dCSR) to leverage the strengths of both formats: while groupwise encoding is used to encode the majority of elements in a memory-efficient way, dCSR processes the arbitrarily

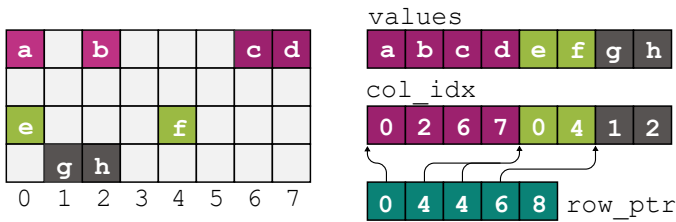


Figure 1. CSR representation of a sparse matrix

positioned elements that can not be assigned to a group. These encoding formats for sparse matrices add important tools for deploying NNs with a low memory footprint on compact devices.

## II. BACKGROUND AND RELATED WORK

The landscape of dedicated encoding formats for sparse matrices can be divided into two major categories. The first category are schemes that target scientific applications with extremely high sparsity in HPC environments. Secondly, there are schemes aimed towards neural network applications on embedded platforms and accelerators. We lay out where both categories are similar, and where they differ. We also provide background on embedded SIMD instructions, a method that parallelizes NNs execution at the intersection of regular MCUs and custom NN accelerators.

### A. Sparse Matrices in HPC

Large numbers of scientific problems involve computation on sparse matrices. Examples range from network graphs, fluid dynamics, computational biology and electrical circuits to optimization problems and computer vision [8]. Across such a diverse range of applications, sparse matrix properties like the degree of sparsity, inherent structure, matrix dimensions and others vary widely. Because scientific applications are run on desktop or HPC systems, they are rarely constrained by memory or power consumption. Sparse matrix encoding schemes in the scientific domain are instead judged mainly by the computational throughput they achieve [9]. The most commonly used representation of a sparse matrix for scientific applications is the CSR format [10]. CSR decomposes a sparse matrix into three contiguous, one-dimensional arrays as shown in Figure 1. The `values` array holds the non-zero values of the matrix. The column index of each element is contained in the `col_idx` array. Lastly, the `row_ptr` array holds a pointer to the first element of each row in the `values` and `col_idx` arrays. Numerous CSR derivatives have been proposed in the literature to increase throughput, both on SIMD [11] and graphics processing unit (GPU) [12] systems.

### B. Sparsity for Neural Networks

The process of ‘Pruning’, or the removal of unnecessary weights from the dense, regular computation of an NN, was proposed early on in the research of NNs [13]. With the success of deep learning, the method has gained renewed interest as an effective means to deal with the ever-growing

parameter counts associated with deep NNs [4]. One approach is the removal of entire filters, a method known as ‘structured pruning’ [14]. An important benefit of structured pruning is that the computation and memory representation of the pruned network remains very similar to the unpruned network. A different approach—and the focus of this work—is unstructured pruning; here, each weight parameter can be an independent subject to pruning. Due to the much larger decision space on what to prune, unstructured pruning yields models that achieve a higher performance per parameter, albeit at the cost of a highly irregular structure [15] of the pruned network. Much of the existing research on pruning discusses how to best identify parameters that should be pruned and carry out retraining of the network to maximize performance per non-zero parameter. Improvements are typically quantified using the active parameter count, which leaves the practical aspects of how to best encode the irregular structure presented by sparse NNs aside [16]. A notable exception is [5] which explicitly compares the performance of sparse NN while also considering the overhead required for encoding the irregular weights. The authors conclude that, despite the additional overhead, sparse models outperform dense counterparts that consume similar amounts of memory.

To enhance both memory and computational efficiency on resource-constrained edge compute systems, NN weights are typically quantized to integer formats of low bitwidth for inference [17]. When incorporating sparsity, storing indexing information in a plain CSR representation would create an overhead several times larger than the memory footprint of the matrix’s small, quantized non-zero values. The main contributor to this overhead is the `col_idx` array: while the `row_ptr` array only grows with the number of rows, `col_idx` contains an entry for each non-zero element. The numerical range of the values in the `col_idx` array is determined by the number of columns in the matrix. For most applications, this means that it has to be encoded using a larger number of bits, compared to the actual values.

A common solution to this issue is the use of ‘Relative Indexing’ [18], [19]. This method transforms the `col_idx` array from the CSR representation so that it does not contain the *absolute* column index of a non-zero element, but instead the index *relative* to its predecessor. This reduces the numerical range occupied by the values in the `col_idx` array so that it can easily be encoded using a smaller number of bits. Padding elements are inserted where needed to avoid overflow of the chosen low bitwidth integer base type. It is worth noting that this sacrifices the inherent parallelism for accessing several elements within a row that is present in the CSR format. Since each index depends on its predecessor, calculating the index of an element is inherently recursive and cannot be carried out for multiple elements in a row at once. This explains why Relative Indexing and similar schemes [6] see most use in custom hardware accelerators, where index computation can easily be parallelized across multiple rows using dedicated hardware. Due to the specific quantization requirements of NNs, embedded SIMD parallelizes across several elements of a single row. In this setting, sequential encoding would interfere with parallelism.

An alternative approach is proposed by [20]. The sparse matrix is partitioned into smaller sub-matrices, each of which allows for the column indices to be encoded in a fixed and small number of bits. This preserves the parallel properties of the original CSR within the submatrices, but not across entire rows of elements in the original matrix.

### C. Embedded SIMD

Deploying deep NNs on resource-constrained platforms involves handling computationally complex operations that can trivially be parallelized. Classical COTS MCUs lack the necessary low bitwidth, highly parallel datapaths to accelerate these applications. This limits the field of tinyML to the study of very small network architectures and problems [21]. One solution for the lack of parallel compute in embedded systems is the use of a dedicated co-processor for the processing of NNs. These neural processing units (NPUs) allow for full customization of dataflow, quantization and processing elements [22], resulting in the highest efficiency. This efficiency, however, comes at the price of added design complexity and a lack of flexibility once the custom co-processor is fixed in silicon.

A compromise is the integration of SIMD instructions into the processor core. This results in a more lightweight, flexible implementation. Several works have opted to implement a subset of the RISC-V Vector Extension [23], [24]. In the domain of proprietary MCUs, ARM has specified the M-Profile Vector Extension (MVE) as part of the ARMv8.1 architecture of microcontrollers [25]. This work focuses on it, as it is the most mature at the time of writing.

## III. PROPOSED METHODOLOGY

We present a scheme that encodes the deviation from a predicted uniform spread of elements in a sparse row. A decomposition of the resulting *delta* values is used to encode non-aligned integer ranges in a SIMD-friendly fashion. From the observed properties of this encoding scheme, we further develop an additional format that identifies element groups and uses a shared property within each group to reduce encoding overhead further. Both schemes are combined into a hybrid scheme that combines the strengths of each individual format to achieve a minimal compression overhead.

### A. Delta-Linear Encoding & Dynamic Bitwidths

Delta-Linear Encoding is based on the idea that elements are roughly uniformly distributed in an NN weight matrix after pruning.

1) *Delta-Linear Encoding (DLE)*: While the row index array in CSR only grows linearly with  $\mathcal{O}(n)$  of the matrix dimensions, the column indices array grows quadratically. In practice, this implies that optimizing for low memory consumption should primarily involve reducing the size of the column indices array. To preserve parallelism of execution, there must not be a dependency between adjacent elements in the index calculation. To achieve this, we first *predict* an element's index using a linear mapping of its position in the column indices array and store only the deviation (the *delta* value) from that prediction.

Let us assume a non-zero element has column index  $c_i$ , located at position  $i$  in the column indices array. We can decompose  $c_i$  into a mapping function  $f(i)$  and the deviation from it:  $\Delta c_i = c_i - f(i)$ . The specific function  $f(i)$  is determined by the user and should depend on the distribution of elements in the matrix. We examine two Convolutional Neural Network (CNN) architectures, following the application of unstructured low-magnitude pruning. Inspection of the sparse weights shows that non-zero values do not appear to be concentrated in any particular areas of the weight matrix, and that the density of non-zero elements is instead similar throughout all regions of the matrix. This makes a linear mapping  $f(i) = i \cdot m + n$  a sensible default. The performance of this default on two CNNs is explored in more detail in Appendix B. We can derive the slope of this linear function as the average distance between two adjacent elements from the number of non-zero elements in the row of the sparse matrix  $k_s$  and the row length of the dense matrix  $k_d$ . This results in a slope of  $m = \lfloor k_d/k_s \rfloor$  that yields small deviations of values for  $\Delta c_i$  within a single row. The slope is rounded to the nearest integer to enable integer-only runtime calculations.

For practical implementations, we group several consecutive delta elements into SIMD runs  $S_j$  of length  $g$ . This gives us the group-wise linear mapping function

$$c_i = \left\lfloor \frac{k_d}{k_s} \right\rfloor \cdot (i \bmod g) + \Delta c_i \quad (1)$$

In this representation, only the  $\Delta c_i$  part requires explicit storage, as the first part of the equation can be inferred at runtime. For each SIMD group  $S_j = \{\Delta c_i, \Delta c_{i+1}, \dots, \Delta c_{i+g-1}\}$  will be retained. We assign a group-wise y-axis intercept  $n_j$  to make sure that each element within the SIMD unit is unsigned and minimize the numerical value of delta values within a group at the same time:

$$n_j = \min S_j \quad (2)$$

which we can remove from the per-element values

$$S'_j = \{\Delta c_i - n_j, \Delta c_{i+1} - n_j, \dots, \Delta c_{i+g-1} - n_j\} \quad (3)$$

At runtime,  $n_j$  now describes the base pointer for the group. The offset and base pointer of an element  $c_{i,j}$  in lane  $i$  of SIMD group  $j$  can be reconstructed as

$$c_{i,j} = \underbrace{i \cdot \lfloor k_d/k_s \rfloor + \Delta c_{i,j}}_{\text{scatter/gather offset}} + \underbrace{n_j}_{\text{Base Pointer}} \quad (4)$$

By subtracting the minimum from  $S_j$ , we ensure that all delta elements within  $S'_j$  fall into the range  $[0, \max S_j - \min S_j]$ . A numerical range starting from zero means that the minimal number of bits is required to encode the delta elements. To further reduce the memory footprint for each group, we apply the same linear decomposition the base pointer delta values  $n_j$  that we applied above to the elements in groups. Our goal is to avoid encoding the average distance between two *groups* explicitly, since it is usually large and redundant. The average distance between two groups is computed by multiplying the number of elements in the group by the average distance between them  $g \cdot \lfloor k_d/k_s \rfloor$ . We can omit the y-axis intercept as long as the distribution of elements in the matrix is not

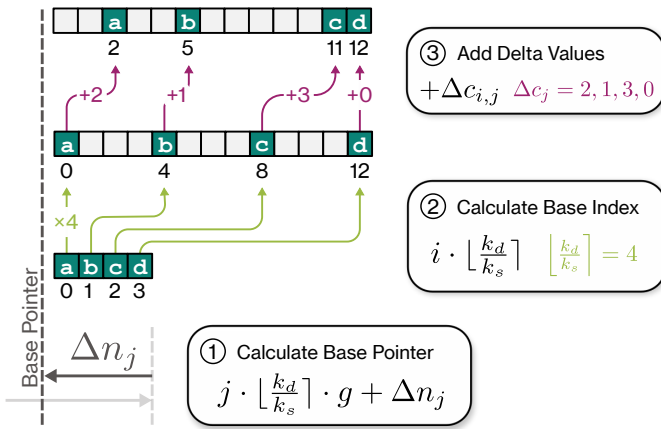


Figure 2. Runtime calculation of base pointer and scatter/gather offsets from Delta-Linear encoding for one SIMD group

highly irregular. To reconstruct  $n_j$  with zero-based indexing, the calculation of  $n_j$  can be expressed as

$$n_j = j \cdot \left\lfloor \frac{k_d}{k_s} \right\rfloor \cdot g + \Delta n_j \quad (5)$$

Alternatively,  $n_j$  can be stored incrementally, similar to the encoding of column indices in Relative Indexing.

$$n_j = \begin{cases} \Delta n_j & \text{for } j = 0 \\ n_{j-1} + \lfloor k_d/k_s \rfloor \cdot g + \Delta n_j & \text{otherwise} \end{cases} \quad (6)$$

In both scenarios,  $\Delta n_j$  is the part of the equation that must be stored in memory. We find that the latter encoding produces a more even distribution of values with lower magnitude in our evaluation, albeit at the cost of introducing an interdependence between groups. The process of deriving a base pointer and per-element offsets from the delta representation is shown in Fig. 2.

Certain boundary conditions must be adhered to. The calculation of per-element offset values must not overflow the bounds given by the width of the SIMD lane. In addition,  $\Delta n_j$  also must not overflow its base type (an 8-bit signed integer in our implementation). We insert padding to ensure that all of these conditions hold. To ensure this, we employ a greedy algorithm that, in every iteration, inserts a zero padding value into the center of the largest gap in the row until no overflow remains.

2) *Dynamic Bitwidth Decomposition (DBD)*: Although Delta-Linear Encoding (DLE) reduces the size of column index elements within a group, it can only guarantee the upper bound for these values that was achieved during the padding insertion. It would be too wasteful for many memory-constrained applications to encode every column index's value using the size of this upper bound. On the other hand, lower bitwidths that are not an even divisor of the machine word size (such as five, six or seven bits) cannot be directly represented in memory in a way that makes it easy to load them in parallel at runtime. Having the largest element of a row determine the bitwidth for all other elements in a row also means that a single outlier value might lead to a lot of wasted memory when other elements would be representable with fewer bits.

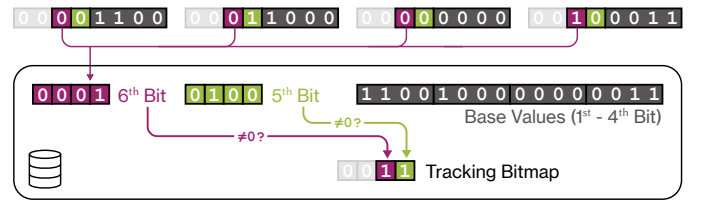


Figure 3. Memory-aligned representation of a SIMD run with a 6 bit value range through decomposition into base values, bit masks and a tracking bitmap

To resolve these issues, each group of delta elements is decomposed into a base value with a fixed number of bits, and several groups of *extension bitmasks* as illustrated in Fig. 3. Every bitmask contains one bit per SIMD lane that marks whether or not the bit is set in the encoded value. Masks that would entirely consist of zeroes are not created. The extension bits can dynamically be added to the base value at runtime if more bits than available in the base value are required. The decomposition is done separately for each SIMD run. Through this, each SIMD run can be encoded in the number of bits required for the largest element in this run, reducing the impact of rare outlier values to a single SIMD group rather than an entire row. The size of the mask is determined by the number of SIMD lanes which can be expected to be a power of two; the extension bitmasks do not cause issues with memory alignment because of this. Since the number of extension bitmasks can vary for each group, it is necessary to explicitly encode how many of them need to be consumed per group at runtime. Each group maintains a *tracking bitmap* for this purpose. This tracking bitmap has one bit for every bit position that might be extended. If the bit is set to one, there exists a bitmask for this position that needs to be consumed at runtime; if it is set to zero, there is no bitmask for this position.

In our implementation, we opt for a base value size of four bits, leaving bits five, six and seven to be extensible. A fixed-size base value is not strictly necessary for the algorithm, as any integer could also fully be decomposed into only a set of extension bitmasks. We introduce the base values because the re-composition at runtime incurs some overhead. Hence, it is more effective to dynamically add bit positions that have a high probability of being absent in a significant number of groups; we've found that this is generally not the case for the lowest four bits, which is why they are always stored as-is. To ensure easy access to the 4-bit base values in memory, we interleave two adjacent groups so that the base values of the first group occupy the upper nibble, while the base values of the second group occupy the lower nibble. The advantage of this memory representation is that when executing a parallel load, each base value is loaded into the correct SIMD lane. The values of either group can be accessed through a single parallel shift or bitwise-or operation. The bitmasks occupy a multiple of a byte and are stored contiguously without interleaving.

During the recomposition of the original value at runtime, all extensible bit positions will be iterated over. If the corresponding bit is not set in the tracking bitmap, the iteration is complete and we continue with the next position. If the bit is set we load the next bitmask from memory and advance

the mask pointer. We then do a parallel bitwise-or operation with the current bit position. The mask is used to apply this operation only to the lanes that need to be extended in this group. The binary lane masking feature that is essential for this process is present in ARM MVE as well as the RISC-V Vector Extension.

The combination of DLE and Dynamic Bitwidth Decomposition (DBD) is required to adapt the CSR format to meet the specific requirements posed by sparse NNs for inference on embedded SIMD processors: parallelizable computation and a memory representation that ensures that each individual component is properly aligned in memory while also providing minimal memory overhead. Together, these techniques form the dCSR encoding format for sparse matrices.

### B. Groupwise Encoding

In DLE, we observe that almost all of the overhead—both in terms of memory and algorithmic complexity—originates from the Delta component  $\Delta c_{i,j}$ . The delta component encodes the deviation of an element's position from an equidistant spacing and must be retained for each element. If the delta values within a group span a large numerical range, they need to be decomposed into a DBD representation and recomposed at runtime. Out of this arises the question: Can both algorithmic and encoding overhead be reduced further if we managed to eliminate the delta component altogether? This would require all elements within a SIMD run to have an equal spacing from each other. Furthermore, existing research suggests that most non-trivial NN models only retain near-baseline performance for medium sparsities, typically in the range below 60% to 80% sparsity. At these relatively low sparsities, the occurrence of equidistant groups of elements is highly likely<sup>1</sup>. The analysis of throughput in [7] suggests that zero-skipping during computation is only worthwhile at very high sparsities. At low sparsities, higher throughput is achieved by extracting sparse tensors into the system's SRAM and performing a dense computation. The cause of this is an additional performance penalty of using gather memory accesses in sparse compute kernels. This allows us to relax another requirement: If an  $n$ -dimensional tensor's data is backed by a contiguous 1-D array in memory (as is typically the case) then it is sufficient to recreate this 1-D array at runtime, rather than extract each row individually. For a high-dimensional tensor, we commit only the sparse structure of its flattened 1-D memory representation to memory. A single, large and contiguous array allows groups to span several rows of the original tensor. Without flattening the tensor first, a jump from one row to the next would otherwise be a hard limit. This simplification of the underlying data structure to a large 1-D array increases the search space and thus the chance of finding equidistant element groups further.

1) *Grouping of Elements*: For the relaxed problem instance of encoding a one-dimensional array, we can describe a sparse array  $\mathbf{A}$  of length  $c_{\max}$  as a set of tuples. These tuples are

comprised of a non-zero value  $v$  and its corresponding position  $c$  in the array, where  $c$  is a unique integer column index:

$$\mathbf{A} := \{(v, c) \mid \forall (v, c) : c \in \mathbb{N} \wedge c < c_{\max} \wedge (\forall (v', c') \neq (v, c) : c' \neq c)\} \quad (7)$$

Within  $\mathbf{A}$ , there might be any number of portions  $\mathbf{B}' := \{b_1, b_2, \dots, b_n\}$ ,  $\mathbf{B}' \subseteq \mathbf{A}$  for which all elements are evenly spaced by some constant distance  $d$ , starting from an offset  $n$ :

$$b_j := \{(v, c) \in \mathbf{A} \mid \exists I_j := (k \cdot d_j + n_j, (k+1) \cdot d_j + n_j, \dots, (k+s) \cdot d_j + n_j), \\ k \in \mathbb{Z}, d_j \in \mathbb{N}^+, n_j \in \mathbb{N} : (\forall (v', c') \in b_j : c' \in I_j)\} \quad (8)$$

from which we are interested only in the subset that minimizes the number of groups and thus provides the partitioning with the lowest overhead while still containing all non-zero elements that are contained in  $\mathbf{B}$ :

$$\mathbf{B} := \min_{\mathbf{B} \subseteq \mathbf{B}'} |\mathbf{B}| \quad (9)$$

$$\text{s.t. } \forall (v, c) \in \mathbf{B}' : (\exists b \in \mathbf{B} : (v, c) \in b) \quad (10)$$

For each group  $b_j$ , this makes it possible to reconstruct the group's index  $I_j$  from only three parameters: the group's size  $s_j$ , the position of the first element  $n_j$  and the distance between elements  $d_j$ —regardless of the number of elements in the group. Analogous to Equation (4), we can calculate the position of  $c_{i,j}$  as:

$$c_{i,j} = \underbrace{i \cdot d_j}_{\text{scatter/gather offset}} + \underbrace{n_j}_{\text{Base Pointer}} \quad \text{s.t. } i < s_j \quad (11)$$

This representation removes the need for storing a *per-element* offset  $\Delta c_{i,j}$ . Instead, the encoding overhead of the group is distributed among all its members and remains constant. Particularly for large groups, this means that the overhead per non-zero element is spread over many group members. For an array that can be broken down into large groups, the overhead will scale favorably.

2) *Group Sizes and Occupancy*: In theory a group can have any number of elements. In practice, this is limited to a handful of useful group sizes. First, groups with too few elements can not profit from the parallel unpacking and reduced overhead (as the constant overhead is spread over only a few elements). Second, a group's elements should be word-aligned in memory to facilitate fast access at runtime. This limits the practical group sizes to multiples of four. The number of lanes in the ARM MVE is 16, which upper-bounds the group size. We pick a group size of four as the lower limit for the reasons given above, leaving 4, 8, 12 and 16 as practically useful group sizes. This might lead to situations where not every element from  $\mathbf{B}'$  is included in  $\mathbf{B}$ , violating the optimization constraint given in Equation (10). The implications of allowing elements of  $\mathbf{B}'$  to be omitted from  $\mathbf{B}$  are addressed in more detail in Section III-C. Having a small set of group sizes also eliminates the need to encode  $s_j$  for each group individually. Instead, we assign groups to separate sub-arrays, depending on their size so that groups of equal size are stored contiguously in memory.

<sup>1</sup>Refer to Appendix A for a detailed analysis of the portion of a sparse array that can be assigned to groups.



It might also be advantageous to allow for gaps in groups. Handling large groups of elements that have some gaps as if those gaps were not present contributes to lower total overhead. If we encode a group of 15 non-zero elements and a gap as if it contained 16 elements by simply including a padding zero in the position of the gap then the constant overhead will still be spread out over 15 active elements. Figure 4 exemplifies this with the first group: allowing for a padding element in a small number of positions will help in finding more groups that can be represented using the groupwise scheme. We can formalize this intuition and calculate how high the group occupancy has to be. Assume that we want to encode a group of  $n_e$  non-zero elements, each with a size of  $c_e$ . For a group size of  $s$  (where  $s \in \{4, 8, 12, 16\}$  in our case) and an additional constant overhead  $c_g$  which encodes each group's starting position and element distance, the amount of memory  $m_g$  required to encode the group becomes

$$m_g = \left\lceil \frac{n_e}{s} \right\rceil \cdot (c_g + s \cdot c_e)$$

with a compression overhead per non-zero element of

$$m_c = 1 - \frac{m_g}{n_e \cdot c_e}$$

For each group size, this helps determine the number of padding elements that can be inserted in order to maximize the overall compression ratio. Due to the proportionally smaller jumps between the larger group sizes 8, 12 and 16, full occupancy is required for  $s = 12$  and  $s = 16$ , while a group of size 8 can have up to one padding element and still have a favorable compression ratio, compared to a fully occupied group of size 4. When comparing each group's size to an assumed 50% overhead for the remainder, this number increases to three padding elements for  $s = 16$  and  $s = 12$  and one for  $s = 8$ . This relaxed configuration trades off a small amount of compression performance in favor of finding fewer and larger groups. Larger group sizes are beneficial because they also contribute to faster extraction, since they make better use of the available SIMD lanes. A case by case distinction based on these values for different group sizes would complicate the search algorithm. Instead, we capture the trade-off between compression rate and preference for larger group sizes in an occupancy parameter  $\theta$  that expresses the desired occupancy as a fraction of the group's size. We propose a fixed value of  $\theta = 0.8$  for a good balance of larger group sizes with compression performance. We validate this parameter in more detail in Appendix C.

3) *Discovering element groups in a sparse array:* An important question is how to find  $\mathbf{B}$ , given  $\mathbf{A}$ . We opt for a greedy approach that tries to find the largest groups with the highest occupancy first. Once a group is found, it is removed from the search space. When all groups above the threshold for a certain group size are exhausted, we move to the next smaller group size and continue the search. We use a 1D convolution over a binary mask of  $\mathbf{A}$  to identify groups and their occupancy. Given a group size  $s$  and element distance  $d$ , we can create a binary mask  $\mathbf{G}(s, d)$  that has ones in the element positions and zeros in the space between elements. Refer to Appendix D for more details on layout and generation of the group mask.

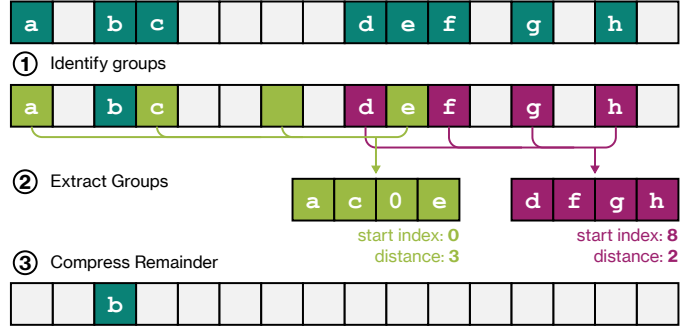


Figure 4. Decomposition of a sparse array of 8 elements into two groups  $b_1, b_2$  of  $s = 4$  with 75% and 100% occupancy and a remainder  $\mathbf{R}$  of one element.

---

### Algorithm 1: Extraction of element groups from a sparse array

---

**Input:** sparse array  $\mathbf{A}$

**Parameters:**

$S$  list of group sizes, sorted in descending order

$D$  set of element distances

$\theta$  occupancy threshold w.  $\theta \in [0, 1]$

**Data:**

$\mathbf{M}_A$  binary mask of  $\mathbf{A}$

$\mathbf{G}$  binary group mask

$\mathbf{C}$  discrete convolution of  $\mathbf{G}$  over  $\mathbf{M}$

**Output:**

$\mathbf{B}$  list of element groups, defined by  $b_j = (s_j, d_j, n_j)$

$\mathbf{R}$  sparse remainder

$$\mathbf{M}_A[i] \leftarrow \begin{cases} 1 & \text{where } \mathbf{A}[i] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

**foreach**  $s \in S$  **do**

**repeat**

**foreach**  $d \in D$  **do**

$\mathbf{G} \leftarrow \text{GroupMask}(s, d)$

$\mathbf{C} \leftarrow \mathbf{G} * \mathbf{M}_A$

$(v_{\max}, n_{\max}) \leftarrow \max \mathbf{C}, \text{argmax } \mathbf{C}$

**if**  $v_{\max} \geq s \cdot \theta$  **then**

$\mathbf{B} \leftarrow \mathbf{B} + (s, d, n_{\max})$

**foreach**  $s_i \in \{0, 1, \dots, s-1\}$  **do**

$\mathbf{M}_A[n_{\max} + s_i \cdot d] \leftarrow 0$

**until**  $\forall d \in D : v_{\max} < s \cdot \theta$

$$\mathbf{R}[i] \leftarrow \begin{cases} \mathbf{A}[i] & \text{where } \mathbf{M}_A[i] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

**return**  $\mathbf{B}, \mathbf{R}$

---

Instead of convolving over  $\mathbf{A}$ , we use a binary mask  $\mathbf{M}_A$  to indicate the positions of non-zero elements, regardless of their numerical values. We can then use the result of the convolution  $\mathbf{C} = \mathbf{G}(s, d) * \mathbf{M}_A$  to identify group starting positions with high occupancy: the convolution response in  $\mathbf{C}$  indicates positions with high overlap of ones in the group mask and ones in the mask of  $\mathbf{A}$ . The convolution result is equivalent to the

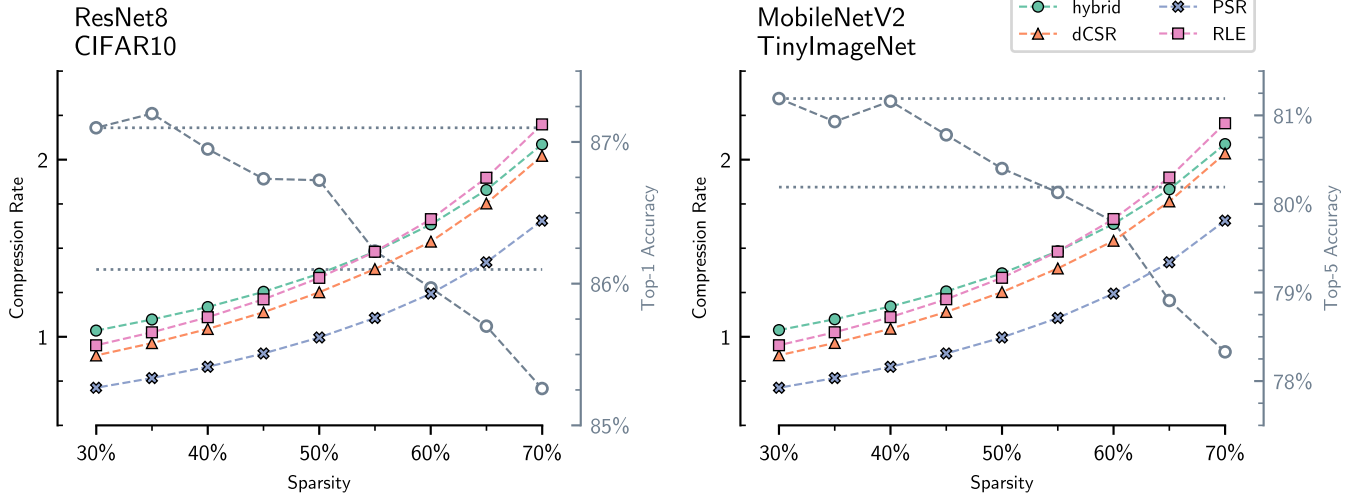


Figure 5. Model accuracy and comparison of compression rates for different sparse encoding schemes across multiple sparsities. Dotted lines mark baseline accuracy and 1 p.p. accuracy drop.

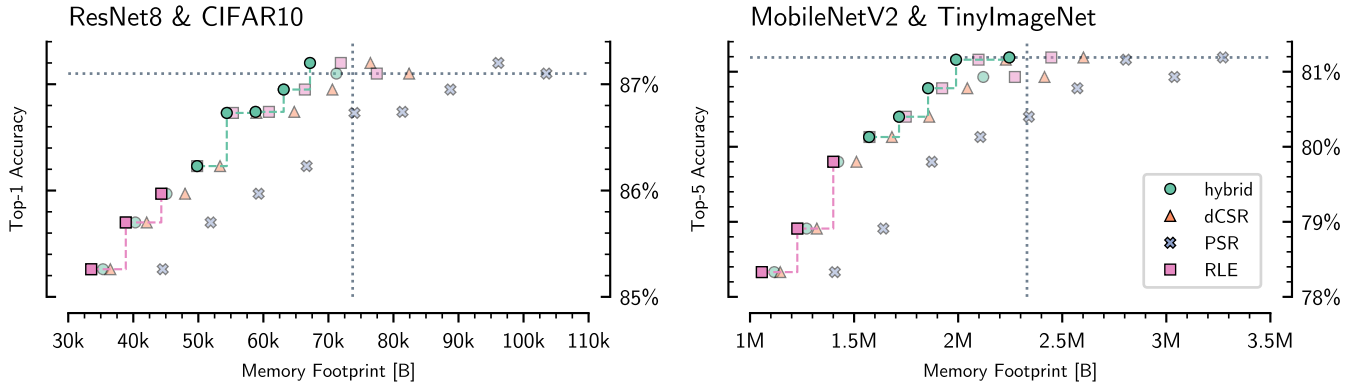


Figure 6. Pareto front of sparse model size and accuracy. Dotted lines mark dense model's accuracy and memory footprint.

group's occupancy. To avoid an overlap of groups (e.g. a run of 17 consecutive values would be decomposed into two fully-occupied groups of 16 elements if we allowed groups to overlap), we only identify one group at a time, remove it from  $\mathbf{M}_A$  and repeat the procedure. If several positions in  $\mathbf{C}$  are above the threshold, we pick the one with the highest occupancy as the newly found group's starting position. If several positions share the highest occupancy, we pick the first one. Starting with the largest group size  $s$ , we test all group distances  $d$ . If we can find no group for any  $d$ , this means that we have exhausted all groups for the current group size and we move to the next-smaller size. The search terminates when we can't find any groups above the occupancy threshold for the smallest group size. The procedure is given in pseudocode in Algorithm 1. This algorithm only provides an approximation of  $\mathbf{B}$  as given in Equation (9). An optimal solution to the group partitioning problem, rather than a simple heuristic, would likely increase the runtime of the algorithm in a way that is not scalable to meaningful applications.

### C. Hybrid Compression Scheme

If we allow group sizes of  $s \geq 1$ , we can ensure that  $\mathbf{B} = \mathbf{A}$  for any sparse matrix  $\mathbf{A}$ . As discussed in Section III-B2, this is not practical since the benefits of groupwise encoding decrease with smaller group sizes. When limiting the search to practical group sizes, there might remain elements that can't be assigned to any element group. The question then becomes how the remaining elements  $\mathbf{R} := \mathbf{A} \setminus \mathbf{B}$  should be handled.

It is important to note that  $\mathbf{R}$  as a subset of  $\mathbf{A}$  is simply another sparse matrix. This means that we can encode it using any scheme that supports arbitrary sparsity patterns. However, the dCSR encoding scheme we described in Section III-A lends itself naturally to this task since it optimizes for the same parameters as groupwise encoding: parallelism and high compression rate. To extend groupwise encoding to real world scenarios, we therefore propose a two-stage encoding algorithm. First, an exhaustive search for increasingly smaller groups is carried out. All values assigned to a group are removed from the sparse matrix. The remainder matrix is then encoded using dCSR.

#### IV. RESULTS AND DISCUSSION

Both dCSR and the hybrid encoding scheme are evaluated in terms of the compression rate that they achieve and the performance of the underlying sparse model. Additionally, we measure the cycles necessary to extract the compressed sparse tensors into SRAM on a prototype system emulating the ARM Cortex-M55 MCU.

The complexity of the greedy search algorithm that partitions the sparse weight tensors into a set of groups  $\mathbf{B}$  grows with the number of non-zero elements (i.e. it is proportional to the size of the flattened array and inversely proportional to the array's sparsity). While the partitioning happens within milliseconds for the smallest weight tensors, partitioning the largest tensors in the ResNet34/TinyImageNet configuration (with up to 2.36 million non-zero elements) might take up to more than an hour. However, this partitioning is only run once during model preparation. Also, since a fast group partitioning algorithm is not the purpose of this study, we only implemented it in a sequential, single-threaded program. Given that the partitioning of a weight tensor is independent of the rest of the network's parameters, a multi-threaded implementation that parallelizes the partitioning of different weight tensors would be a straightforward improvement.

##### A. Model Accuracy and Compression Rate

Sparse models must maintain a delicate balance: increased sparsity leads to smaller models, but can also negatively impact model accuracy. How much a model can be shrunk for a given sparsity depends on the overhead produced by the encoding of the sparse tensors. We choose three image classification tasks and CNNs to investigate this trade-off.

1) *ResNet8 and CIFAR10*: Using a ResNet8 [26] model to classify images from the CIFAR10 [27] data set is a test case taken from the TinyMLPerf library [28] of embedded machine learning challenges. The ResNet8 is trained using the architecture and optimization parameters given in the original work. The baseline model is pruned iteratively over 30 epochs to the desired target sparsity, followed by 10 epochs of retraining. Network parameters are optimized using stochastic gradient descent (SGD) with a learning rate of  $5 \times 10^{-2}$ . The learning rate is decayed by 90% after epochs 2 and 7 of the retraining. Small weight tensors do not contribute much to the network's memory consumption, but often disproportionately influence the final accuracy. For that reason, we only prune weight tensors with more than 2048 parameters. These prunable tensors still contain 95.2% of all weight parameters.

2) *MobileNetV2*: We choose MobileNetV2 [29], since it is a lightweight architecture that is already well-optimized for use on edge devices. First, a dense baseline model is trained on the TinyImageNet dataset, a reduced version of the full ImageNet [30] challenge. In TinyImageNet, the number of classes is reduced from 1000 to 200, with 500 training and 50 validation samples per class. Images are reduced to  $64 \times 64$  pixels in size from an original  $224 \times 224$  pixels for ImageNet. The augmentation procedure is the same as for the original ImageNet dataset. The model is pruned over 18 epochs, which is followed by 8 epochs of retraining. Similar to the

ResNet experiment, the SGD optimizer uses a learning rate of  $5 \times 10^{-2}$  for the pruning and the first two retraining epochs and is lowered by 90% after retraining epochs 2 and 6. The cutoff for deciding which tensors are pruned is set to 8192. The prunable tensors then account for 95.7% of the model's weights.

3) *ResNet34*: ResNet34 [26] is included as an example of a larger model to test performance on CNNs with high parameter counts. To account for the smaller input size compared to the full ImageNet dataset, we make a small alteration to the architecture by changing the stride of the first MaxPooling layer from two to one. This does not affect the parameter count but reduces the downsampling in the initial portion of the network. From a pre-trained model, we prune for 10 epochs, followed by 6 epochs of retraining. The model is optimized using SGD with an initial learning rate of  $5 \times 10^{-2}$  and a decay of 90% after the 10th and 14th epoch. In this architecture, even the smallest weight tensors have 36864 elements, which means that a cutoff is not necessary and all weight tensors are being pruned.

4) *MobileViT*: While the initial success of the Transformer architecture was limited to the field of Natural Language Processing (NLP) [31], it has been adopted in other fields, including computer vision [32]. The Transformer architecture now competes with the more established CNNs. Due to the growing importance of this architecture across domains, we validate whether our method continues to work when applied to Transformer-style architectures. While large-scale Transformers like Large Language Models (LLMs) or diffusion models with billions of parameters will not be deployable on embedded systems in the foreseeable future, architectures that combine convolutions with attention layers for image recognition might be small enough to fit even the tight constraints of microcontrollers. We include the XXS variant of the MobileVisionTransformer (MobileViT) [33], one such architecture, in the comparison. The model is trained for 20 epochs on the TinyImageNet dataset with a learning rate of  $1 \cdot 10^{-3}$  that is decayed to  $1 \cdot 10^{-5}$  using a cosine schedule. Iterative pruning is carried out using the same hyperparameters, but with the initial learning rate reduced to  $1 \cdot 10^{-4}$ . For the same reasons as above, the cutoff value is set to 8192, meaning that pruning targets 83% of the model's parameters.

We carry out a detailed analysis of the trade-off between model accuracy and memory footprint for the two smaller models. Each baseline model is pruned to different sparsities while keeping the same hyperparameters. The resulting sparse models are compressed using different encoding schemes, as described in Table II.

Model performance and compression rates across a range of sparsities are compared in Figure 5. For both networks, we observe that a steep and mostly linear degradation of accuracy starts at 50% sparsity for ResNet8 and at around 45% for MobileNetV2. At these levels of sparsity, the weight matrices still contain a relatively high number of non-zero elements. This increases the likelihood of encountering element groups, which allows for a large portion of the weights to be encoded using the groupwise scheme with low overhead. The compression rates achieved by the various encoding formats appears to be



Table I  
COMPARISON OF MEMORY FOOTPRINT AND EXTRACTION CYCLE COUNT FOR SEVERAL SPARSE COMPRESSION FORMATS ACROSS ONE TRANSFORMER AND THREE CNN ARCHITECTURES FOR DIFFERENT LEVELS OF SPARSITY

	Params. (1)	Sparsity	$\Delta$ Acc. Top-1 $\uparrow$ (p.p.)	Memory Footprint $\downarrow$ (total / relative)				Cycle Count $\downarrow$ (total / relative)				
				hybrid	RLE	dCSR	PSR	Inference	hybrid	RLE	dCSR	PSR
ResNet8 CIFAR-10	77.77k / 73.73k	30%	-0.22	<b>71.23k</b> / <b>0.97</b>	77.44k / 1.05	82.41k / 1.12	103.53k / 1.40	<u>29.72M</u>	451.11k / 0.02	392.46k / 0.01	369.79k / 0.01	<b>306.04k</b> / <b>0.01</b>
		50%	-0.54	<b>54.38k</b> / <b>0.74</b>	55.32k / 0.75	58.92k / 0.80	74.03k / 1.00	<u>29.72M</u>	354.44k / 0.01	283.32k / 0.01	265.55k / 0.01	<b>222.13k</b> / <b>0.01</b>
		70%	-1.96	35.34k / 0.48	<b>33.53k</b> / <b>0.45</b>	36.47k / 0.49	44.55k / 0.60	<u>29.72M</u>	241.34k / 0.01	174.28k / 0.01	169.73k / 0.01	<b>138.05k</b> / <b>0.00</b>
MobileViT_XXS TinyImageNet	1.08M / 899.07k	30%	0.12	<b>844.80k</b> / <b>0.94</b>	922.72k / 1.03	986.86k / 1.10	1.23M / 1.37	<u>136.56M</u>	5.11M / 0.04	4.67M / 0.03	4.43M / 0.03	<b>3.60M</b> / <b>0.03</b>
		50%	-0.17	<b>645.97k</b> / <b>0.72</b>	663.62k / 0.74	711.25k / 0.79	884.66k / 0.98	<u>136.43M</u>	4.11M / 0.03	3.40M / 0.02	3.22M / 0.02	<b>2.62M</b> / <b>0.02</b>
		70%	-1.64	425.91k / 0.47	<b>409.28k</b> / <b>0.46</b>	450.81k / 0.50	535.54k / 0.60	<u>136.25M</u>	2.83M / 0.02	2.12M / 0.02	2.11M / 0.02	<b>1.65M</b> / <b>0.01</b>
MobileNetV2 TinyImageNet	2.46M / 2.33M	30%	0.00	<b>2.25M</b> / <b>0.96</b>	2.45M / 1.05	2.60M / 1.12	3.27M / 1.40	<u>49.74M</u>	13.93M / 0.28	12.71M / 0.26	11.70M / 0.24	<b>9.66M</b> / <b>0.19</b>
		50%	-0.79	<b>1.72M</b> / <b>0.74</b>	1.75M / 0.75	1.86M / 0.80	2.34M / 1.00	<u>49.73M</u>	11.26M / 0.23	8.99M / 0.18	8.42M / 0.17	<b>7.12M</b> / <b>0.14</b>
		70%	-2.86	1.12M / 0.48	<b>1.06M</b> / <b>0.45</b>	1.15M / 0.49	1.41M / 0.60	<u>49.73M</u>	7.63M / 0.15	5.50M / 0.11	5.31M / 0.11	<b>4.42M</b> / <b>0.09</b>
ResNet34 TinyImageNet	21.38M / 21.27M	30%	0.06	<b>20.49M</b> / <b>0.96</b>	22.33M / 1.05	23.74M / 1.12	29.86M / 1.40	<u>2.88G</u>	125.39M / 0.04	113.57M / 0.04	103.11M / 0.04	<b>89.83M</b> / <b>0.03</b>
		50%	-0.17	<b>15.65M</b> / <b>0.74</b>	15.95M / 0.75	16.98M / 0.80	21.35M / 1.00	<u>2.88G</u>	101.92M / 0.04	82.05M / 0.03	73.96M / 0.03	<b>66.45M</b> / <b>0.02</b>
		70%	-2.40	10.19M / 0.48	<b>9.64M</b> / <b>0.45</b>	10.46M / 0.49	12.84M / 0.60	<u>2.88G</u>	69.48M / 0.02	50.11M / 0.02	46.76M / 0.02	<b>41.04M</b> / <b>0.01</b>

(1) total / in sparse tensors

Table II  
REFERENCE ENCODING FORMATS

Format	Description
hybrid	The proposed hybrid encoding scheme in which the array is first decomposed into groups with $s = \{4, 8, 12, 16\}$ , $d = [1, 16]$ , $\theta = 0.8$ . The irregular remainder is encoded using the dCSR format.
dCSR	All weight arrays are encoded using the plain dCSR format with 4 bit base indices
RLE	CSR encoding with relative column indices [18], [19] where each relative index is encoded in 4 bit and padding elements are inserted in case of an overflow
PSR	Weight arrays are encoded using the PSR format with 8 bit indices as presented in [20]

highly consistent across the different networks in our evaluation for the same level of sparsity. The additional evaluation of ResNet34 in Table I supports this finding, suggesting that there is a strong correlation between sparsity and compression rate.

Within a 1 p.p. drop in accuracy, the ResNet8 model achieves a compression rate of 1.48 at 55% sparsity when compressed using the hybrid encoding scheme. In the same performance envelope, MobileNetV2 reaches a sparsity of 50% and a compression rate of 1.36. Between the start of an actual reduction in size and a drop in accuracy exceeding 1 p.p., hybrid encoding achieves, on average, a size reduction of 4.3% and 5.4% respectively over *RLE*, the next-best reference format. Note that the hybrid encoding scheme already compresses

models to below the size of the dense baseline at sparsities that are as low as 30%. Across the most interesting sparsity region between 30% and 60% where model performance is still close to the baseline the hybrid scheme achieves the highest compression rates of all schemes in the comparison. The trade-off between accuracy and memory consumption is further explored in Figure 6, where we compare each sparse model's accuracy directly to its size under different encoding schemes. This confirms that the hybrid scheme performs particularly well when compressing low to medium sparsities where the likelihood of encountering element groups is high. These low to medium sparsities coincide with performance at or near that of the dense baseline model. The additional evaluation of the larger ResNet34 model in Table I confirms the observations from the two smaller models: While the compression performance of the hybrid scheme degrades for higher sparsities, it can leverage even very low sparsities to reduce the size of models within one percentage point of the baseline accuracy. Evaluation of the MobileViT model also underlines another finding from the three CNNs, namely that the same encoding format tends to produce similar compression rates for the same sparsity, regardless of the model's architecture.

### B. Decompression Throughput

We estimate the throughput achievable by each sparse encoding format on the ARM MPS3-AN547 platform. MPS3-AN547 is an FPGA prototyping system that emulates the ARMv8.1 architecture and its MVE instructions. For NN

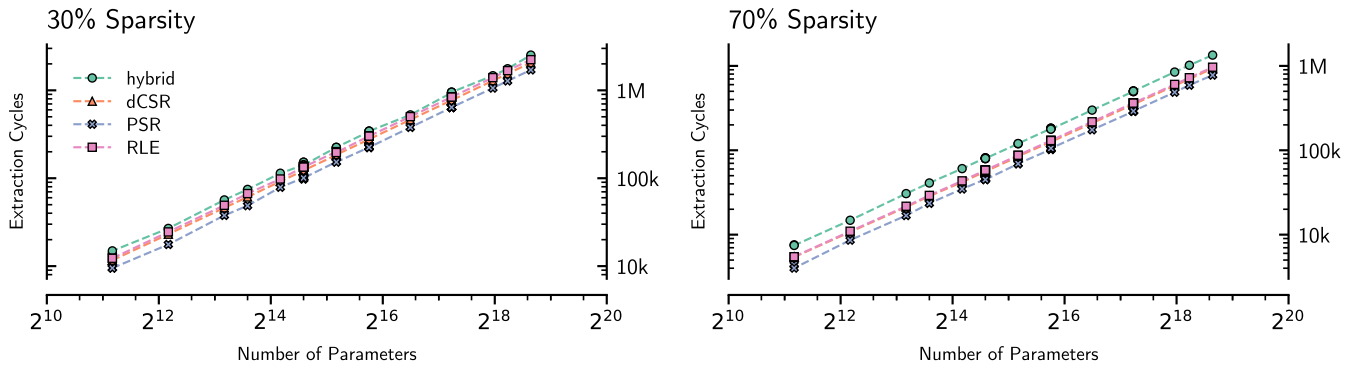


Figure 7. Cycles for sparse buffer extraction on ARM-MPS3 AN547 for weight tensors from ResNet8 and MobileNetV2. Comparison between 30% and 70% sparsity.

applications, MVE improves in several ways over previous generations of Cortex-M devices by providing support for 8-bit Multiply-Accumulate (MAC) operations and a 128-bit vector word width. All weights are loaded out of the system’s DDR4 RAM at runtime. From their sparse representation, weights are extracted into a dense buffer in SRAM. We measure the execution cycles for each extraction, as reported by the ARM Cortex-M55 core’s Performance Monitoring Unit (PMU). For context, we also provide the amount cycles required for a single inference run with a batch size of one in Table I. Inference is run using a model that was converted for inference using the CMSIS-NN kernel library with the microTVM NN compiler [34] and compiled using the same settings as other benchmarking code.

All sparse weight tensors from the two reference architectures in Section IV-A are merged into a benchmark set. We obtain a broad range of tensor sizes with parameter counts between 2,304 and 409,600 elements. The impact of sparsity on throughput is assessed by evaluating two different sparsities at the upper and lower ends of the range of most interesting sparsities, i.e. the sparsities between the onset of compression benefits and the point where the degradation of accuracy becomes too large. The benchmarking results across our benchmark set are presented in Figure 7. The PSR format achieves the highest throughput of all reference formats since its 8-bit index values have a larger memory footprint but require practically no further processing at runtime. The hybrid scheme is algorithmically less complex than plain dCSR since it omits the DBD of indices. Despite this, throughput is lower by a geometric average of 19.8% for the benchmark with 30% sparsity and 42.8% at 70% sparsity. This is because a large portion of the groups neither reach the maximum size nor are they fully occupied. Both a small group size and reduced occupancy contribute to the underutilization of SIMD hardware, since only one group can be extracted at a time. For a group size of  $s = 4$ , this results in 75% of the parallel compute capacity remaining unused during extraction. This also explains why throughput is closer to dCSR at lower sparsities, with a larger gap for increased sparsities; at lower sparsities, the chance of encountering groups of larger size is increased due to a higher density of elements. This makes the hybrid scheme more

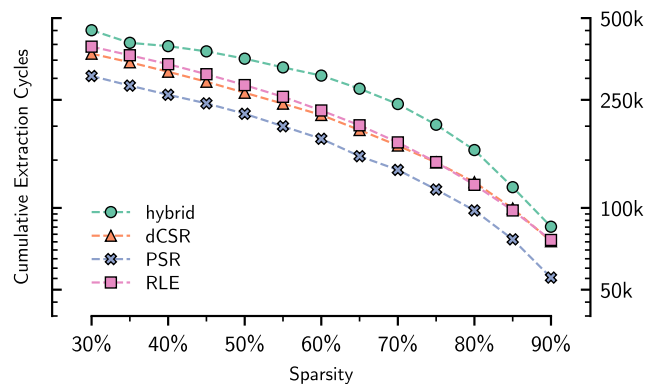


Figure 8. Cumulative cycles for buffer extraction on ARM-MPS3 AN547 for sparse weight tensors from ResNet8 for different sparsities.

efficient at low sparsities, while extraction overhead rises with higher sparsity. The underutilization of SIMD units is also the main contributor to the 10% and 38% overhead of the hybrid scheme over the use of Relative Indices. A hyperparameter that controls the trade-off between throughput and compression rate is the set of available groups sizes  $S$ . Allowing for small group sizes, e.g.  $4 \in S$ , increases the likelihood of encountering element groups, which improves the compression rate, but leads to poorer SIMD utilization. Determining which choice of group sizes yields the best results is dependent on the requirements of the target application, the NN hardware architecture used, hardware platform and resource availability.

The results from Figure 7 suggest that throughput is directly proportional to the number of parameters in the extracted buffer. The results also indicate that the throughput of different methods is influenced in different ways by the underlying weight tensor’s sparsity. We further investigate this by comparing the cumulated extraction for all sparse weight tensors from ResNet8 across a broader range of sparsities in Figure 8.

When comparing the decompression throughput with the cycles required for inference in Table I, it is clear that the total overhead of sparse compression and decompression is highly dependent on the model architecture. Both ResNet

variants rely on 2D Convolutions which have a higher arithmetic intensity. Due to their higher reuse of parameters, their inference dominates the overall cycle count, while no combination of sparsity and decompression algorithm adds more than 4% runtime overhead. The overhead is more pronounced for MobileNetV2, which uses fewer computations per parameter due to its reliance on Depthwise-Separable Convolutions. Whether the reduced throughput justifies the savings in memory is dependent on the base model's architecture as well as the constraints on memory on the target systems. On embedded systems, where memory is contained on the device itself and can't easily be extended after production, even small reductions in an application's memory requirements can be crucial.

## V. CONCLUSION AND OUTLOOK

While unstructured Pruning is effective at reducing the active parameters in an NN model, it imposes the challenge of highly irregular patterns in the remaining weights. This often means that fewer active parameters do not end up reducing the network's memory consumption. We proposed two ways of encoding these irregular patterns for the latest generation of embedded MCUs. Small amounts of available memory, low-bitwidth quantization of weights and the growing availability of SIMD instructions set embedded applications apart from their desktop and HPC counterparts. These properties inform the need for a high compression rate, combined with the capability of recovering the indices of elements within a row in parallel. From this, we first develop dCSR. dCSR predicts an even spacing of elements in a sparse tensor and only encodes the deviation of each element from that even spacing in order to reduce the memory footprint. This DLE produces values of a small numerical range, but does not guarantee an upper bound of this numerical range. To solve the resulting issues with memory alignment, DBD is used to make sure that all data structures can efficiently be accessed by parallel instructions through proper memory alignment. While dCSR is most beneficial at high sparsities, we find that many NN applications can only tolerate medium sparsities before performance is degraded to unacceptable levels. By augmenting dCSR with an additional groupwise encoding scheme, we improve the compression rate for low and medium sparsities. This scheme finds groups of equidistant elements which can be encoded with even lower overhead. Both are combined into a hybrid scheme in which elements that can be assigned to groups are encoded using the low-overhead groupwise encoding, while the sparse remainder is encoded using dCSR

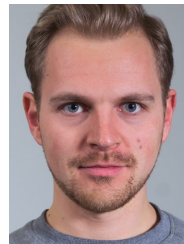
Our evaluation shows an average reduction of 4.3% to 5.4% in size, compared to the closest encoding format for two CNN models trained on different datasets. This reduction is achieved in the range of low to medium sparsities, where the model's performance remains close to that of the dense model. We find that the throughput of the hybrid encoding scheme is below that of other encoding formats, despite being able to parallelize extraction operations. We attribute this shortcoming to the underutilization of parallel hardware by small groups.

Embedded microcontrollers and NNs pose a unique set of challenges when compared to edge, desktop and HPC systems.

This is due to a different balance between memory and compute capabilities. Enabling the use of sparsity in this domain is a vital tool for bringing NN performance closer to the user's devices and data, without the need for cloud computing and permanent connectivity.

## ACKNOWLEDGMENTS

The project on which this report is based was funded by the German Ministry of Education and Research (BMBF) under the project number 16ME0542K. The responsibility for the content of this publication lies with the author.



**Elias Trommer** received an M.Sc. in Computer Engineering from Technische Universität Berlin in 2020. He is currently pursuing a Ph.D. degree at Technische Universität Dresden, in cooperation with Infineon Technologies Dresden. His research interests include the application of gradient-based optimization techniques and the inference of neural networks on resource-constrained devices like microcontrollers.



**Bernd Waschneck** received an M.Sc. in semiconductor physics from Ludwig-Maximilians-Universität Munich in 2013 and a Ph.D. in manufacturing engineering from the University of Stuttgart in 2020. Since 2014, he works at the semiconductor company Infineon Technologies in the fields of data science and artificial intelligence. He currently works as Director of System Innovation & Software in the Infineon Development Center Dresden, where he leads the System Innovation team. His research interests include hardware and software AI acceleration and embedded AI on microcontrollers for smart sensors.



**Akash Kumar** received the joint PhD degree in electrical engineering and embedded systems from the Eindhoven University of Technology and the National University of Singapore (NUS) in 2009. From 2009 to 2015, he was with NUS. He is currently a Professor with Technische Universität Dresden, where he is directing the Chair for Processor Design. His current research interests include the Design, Analysis, and Resource Management of Low-Power and Fault-Tolerant Embedded Multiprocessor Systems.

## REFERENCES

- [1] K. Hornik, M. B. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [2] M. S. Mahdavejad, M. Rezvan, M. Barekatain, P. Adibi, P. M. Barnaghi, and A. P. Sheth, "Machine learning for internet of things data analysis: A survey," vol. 4, no. 3, pp. 161–175, 2018.
- [3] N. N. Alajlan and D. M. Ibrahim, "TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications," *Micromachines*, vol. 13, no. 6, p. 851, May 2022.
- [4] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015, pp. 1135–1143.
- [5] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018*, 2018.
- [6] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G.-Y. Wei, "Structured compression by weight encryption for unstructured pruning and quantization," in *CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, 2020, pp. 1906–1915.
- [7] E. Trommer, B. Waschneck, and A. Kumar, "dCSR: A memory-efficient sparse matrix representation for parallel neural network inference," in *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*. IEEE, 2021, pp. 1–9.
- [8] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
- [9] M. Kreutzer, "Performance engineering for exascale-enabled sparse linear algebra building blocks," Ph.D. dissertation, University of Erlangen-Nuremberg, Germany, 2018.
- [10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 1994.
- [11] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*. ACM, 2015, pp. 339–350.
- [12] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi, "TaiChi: A Hybrid Compression Format for Binary Sparse Matrix-Vector Multiplication on GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3732–3745, Dec. 2022.
- [13] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS Conference, Denver, Colorado, USA, November 27-30, 1989*. Morgan Kaufmann, 1989, pp. 598–605.
- [14] Y. He and L. Xiao, "Structured Pruning for Deep Convolutional Neural Networks: A survey," 2023.
- [15] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *CoRR*, vol. abs/1705.08922, 2017.
- [16] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag, "What is the state of neural network pruning?" in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, 2020.
- [17] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," *ArXiv*, Jun. 2021.
- [18] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016*, 2016.
- [19] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953–1965, Sep. 2020.
- [20] A. Weaver, K. Kavi, P. Vasireddy, and G. Mehta, "Memory-Side Acceleration and Sparse Compression for Quantized Packed Convolutions," *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 81–90, Nov. 2022.
- [21] H. Han and J. Siebert, "TinyML: A Systematic Review and Synthesis of Existing Research," in *ICAIIC 2022*, Feb. 2022, pp. 269–274.
- [22] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead," *IEEE Access*, vol. 8, pp. 225 134–225 180, 2020.
- [23] M. Johns and T. J. Kazmierski, "A Minimal RISC-V Vector Processor for Embedded Systems," in *2020 Forum for Specification and Design Languages (FDL)*, Sep. 2020, pp. 1–4.
- [24] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, "Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters," *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, Oct. 2022.
- [25] A. Skillman and T. Edsö, "A Technical Overview of Cortex-M55 and Ethos-U55: Arm's Most Capable Processors for Endpoint AI," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, Aug. 2020, pp. 1–20.
- [26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Jun. 2016.
- [27] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.
- [28] C. R. Banbury, V. J. Reddi, P. Torelli, N. Jeffries, C. Király, J. Holleman, P. Montino, D. Kanter, P. Warden, D. Pau, U. Thakker, A. Torrini, J. Cordaro, G. D. Guglielmo, J. M. Duarte, H. Tran, N. Tran, W. Niu, and X. Xu, "MLPerf tiny benchmark," in *NeurIPS Datasets and Benchmarks 2021, December 2021, Virtual*, 2021.
- [29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, Jun. 2018.
- [30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
- [31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. ukasz Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017.
- [32] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," Jun. 2021.
- [33] S. Mehta and M. Rastegari, "MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.
- [34] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

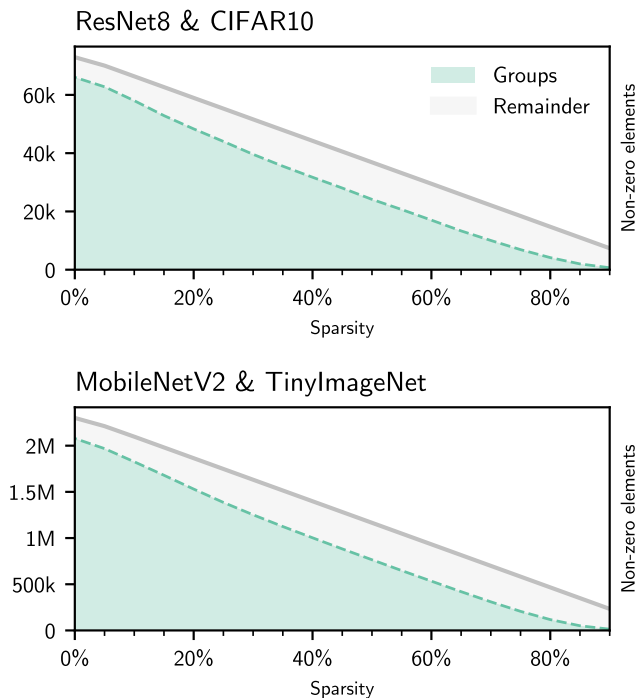


Figure 9. Number of non-zero weights in groupwise submatrix  $\mathbf{B}$  and remainder  $\mathbf{R}$  for different sparsities

## APPENDIX

### A. Likelihood of element groups

To substantiate our theory that a significant portion of a sparse matrix can be represented using a groupwise encoding, we perform an experimental evaluation of the number of non-zero elements assigned to groups compared to the number of elements in the remainder  $\mathbf{R}$ . The evaluation is performed using the parameters described in Section IV-A. The distribution of non-zero elements into groups and remainder, depending on sparsity is shown in Figure 9.

Notably, even for 0%, there is a sparse remainder. This is peculiar since a dense matrix is trivially decomposable into one or several groups, all of them with  $d = 0$ . This indicates that the greedy approximation of  $\mathbf{B}$  generated by Algorithm 1 fails to produce an optimal solution due to the assignment of elements to groups of non-optimal sizes. Improving the performance of the group search algorithm could therefore be a worthwhile direction of future research.

Despite the non-optimal group search algorithm, the majority of non-zero elements can be assigned to groups for sparsities of 65% to 70% for both networks. At 30% sparsity, a point that typically incurs marginal performance loss, 77% of elements are assigned to groups and can be encoded with reduced overhead.

### B. Spacing of elements

To examine the error between the actual positions and the positions predicted by a linear mapping function  $f(i)$  in the two example CNN architectures from Section IV-A, we construct a histogram of the residual error  $\Delta c_i$  between the ground truth

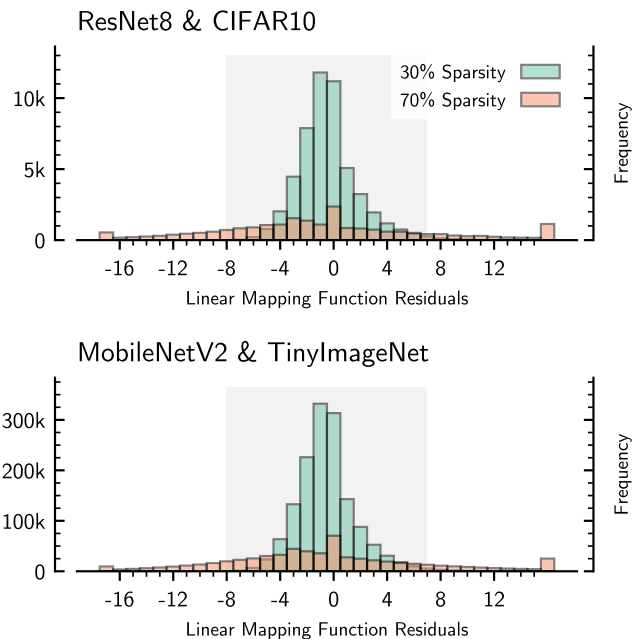


Figure 10. Histogram of residual error between linear prediction and encoded indices. Shaded region marks value range that can be encoded in four bits.

and the mapping function  $f(i)$ . In each SIMD run, we track the difference between the linear spacing predicted by the evenly spaced base index and the actual indices encoded in each lane. If the distances between elements were to follow a normal distribution, the error residuals would show a normal distribution that is centered around zero. From the experimental data shown in Figure 10, we can see that the distribution of residuals is not perfectly normal, but slightly skewed to the right. The reason for this skew is likely that the distance between adjacent elements is lower-bounded by a distance of one, while it is only upper bounded by the size of the array. This causes the value range below the mean error to be slightly more compressed than the range above the mean. Despite this mismatch, the amount of values with an error between  $-8$  and  $7$  (i.e. offsets that can be encoded using 4-bit values) is very high given the linear spacing function based on the mean distance of non-zero elements. This suggests that choosing a different mapping function would likely only lead to marginal improvements in compression performance for the weight tensors studied.

### C. Impact of group occupancy $\theta$ on partitioning

Both the distribution of group sizes and the encoding overhead are affected by the choice of the group occupancy parameter  $\theta$ . We conduct an experiment by varying  $\theta$  when partitioning two ResNet8 instances with 30% and 70% sparsity. For each configuration, we track the amount of non-zero elements assigned to each group size, as well as the additional overhead created by padding elements and other metadata. As partitioning also affects the size of the remainder, we additionally list the non-zero elements in  $\mathbf{R}$  and its additional overhead, as well as the combined overhead in relationship to



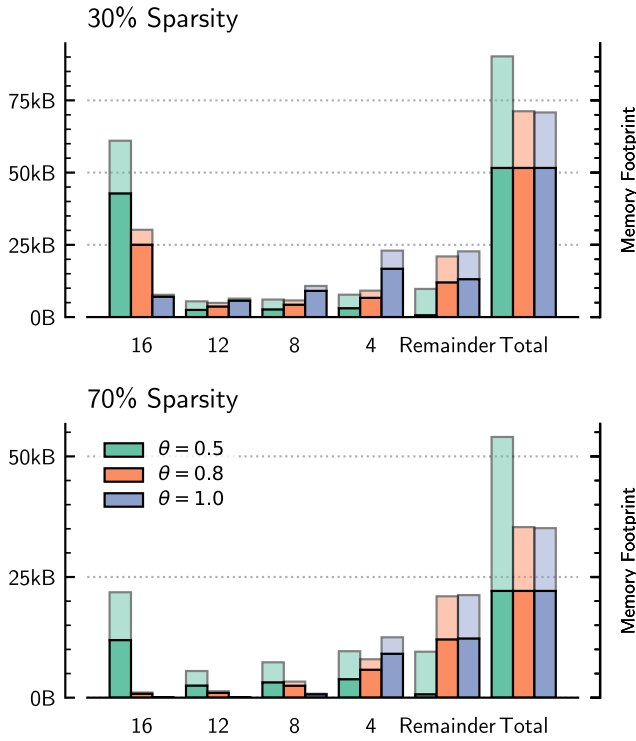


Figure 11. Distribution of memory footprint across group sizes and remainder for different group occupancy values  $\theta$  on ResNet8/CIFAR-10 for 30% and 70% sparsity. Dark region marks non-zero elements, shaded region marks compression overhead.

the compressed non-zero elements. The results in Figure 11 indicate that a low  $\theta$  value increases the likelihood of finding more and larger groups because lower occupancy means that fewer non-zero elements are required to form a group. However, this lower threshold also causes the inclusion of more padding elements into the groupwise partitioned portion **B**. Because of this, the solution with  $\theta = 0.5$ , the lowest occupancy threshold, results in nearly twice the compression overhead of the solutions that employ higher values for  $\theta$ . The solutions for  $\theta = 0.8$  and  $\theta = 1.0$  are very close in terms of compression rate with  $\theta = 1.0$  giving a 0.58% improvement for the lower sparsity case and a 0.05% improvement for the higher sparsity. Particularly for the lower sparsity, a slightly lower value for  $\theta$  helps reduce the number of elements in the remainder, which can be beneficial, depending on the overhead of the encoding scheme used to encode it. Similarly, the lowered occupancy threshold of  $\theta = 0.8$  helps the search algorithm converge towards a solution with fewer and larger groups. This can help in trading compression performance for reduced runtime overhead.

#### D. Binary Group Mask

Groups are detected using a greedy search algorithm. A discrete convolution of a binary group mask over the sparse array representation shows groups of high occupancy. Each combination of group parameters  $s, d$  is represented by a binary mask **G** that is convolved over a binary mask of the array **M**.

---

#### Algorithm 2: Generation of binary convolution Mask from group parameters

---

**Input:**

- $s$  Number of group elements w.  $s \in \mathbb{N}^*$
- $d$  Distance between group members w.  $d \in \mathbb{N}$

**Data:** total length of mask  $l$

**Output:** binary group mask **G**

**Function** GroupMask( $s, d$ ):

```

 $l \leftarrow s + (s - 1) \cdot (d + 1)$ 
G  $\leftarrow$  Array(size= $l$ )
G[ $i$ ]  $\leftarrow$   $\begin{cases} 1 & \text{where } i \bmod d = 0 \\ 0 & \text{otherwise} \end{cases}$ 
return G

```

---

**G** will be an array of length  $s + (s - 1) \cdot d$ , containing  $s$  ones with  $s - 1$  runs of zeros, each with length  $d$  in-between, e.g. GroupMask(4, 1) = 1010101 or GroupMask(6, 2) = 1001001001001001. Note that there are no trailing zeros, as those would interfere with finding groups towards the end of the array. The procedure for generating the group mask is also given in Algorithm 2.