2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)

Memory-Side Acceleration and Sparse Compression for Quantized Packed Convolutions

Alex Weaver
University of North Texas
Denton, Texas, USA

Krishna Kavi
University of North Texas
Denton, Texas, USA

Pranathi Vasireddy
University of North Texas
Denton, Texas, USA

Gayatri Mehta
University of North Texas
Denton, Texas, USA

***Abstract*—Neural Network compression techniques, such as parameter quantization and weight pruning have made deep neural network (DNN) inference more efficient for low-power devices such as MCUs and edge devices by reducing the memory and computation overhead required with minimal impact on model accuracy. To avoid storing and computing zeros, these techniques necessitate the use of sparse data representations, which introduces execution overhead to locate values required by a computation. Sparse matrix formats like Compressed Sparse Row (CSR) and other more recent designs are computationally inefficient when applied to the convolution algorithm as well as inefficient for storing quantized values. In this paper, we outline an intuitive extension of CSR called Partitioned Sparse Representation (*PSR*) in conjunction with a convolution algorithm that hides the cost of indexing overhead via a simple memory-side RISC-like core. *PSR* divides the entire weight array for a convolution layer into partitions that allow for smaller (e.g., 8-bit) indexes to reduce storage overhead. We also rely on a memory-side accelerator called *HHT*, a programmable, near-memory RISC-like co-processor that enables efficient processing of sparse data (including *PSR*). We show that *HHT* together with *PSR* allows the CPU to maximize the advantage of RISC-V packed instructions on sparse quantized data. We show as much as 10x speedup for sparse CONV with *HHT* over a baseline of the CPU performing all computations on dense data. *HHT* performs 2.7x faster on end-to-end image classification inference over the baseline and achieves 70% energy savings over sparse CONV with CPU performing all computations.**

***Index Terms*—CNN, sparsity, compression, RISC-V, programmable, quantization**

I. Introduction

With neural network (NN) inference moving increasingly to edge devices, much work has been done on model compression techniques to improve performance and efficiency. Because edge devices are constrained by limited on-chip memory and slower processor speeds, optimizations that reduce the size of the parameter space for a given network without significantly reducing the network's accuracy are particularly useful. Parameter quantization and pruning are standard ways to reduce the bit width and increase the sparsity of weights and features for NN layers. As pruning techniques increase sparsity without sacrificing accuracy [1, 2], quantization reduces model size by reducing the bit width of network parameters. This reduction has the double benefit of shrinking the storage requirements of a model as well as reducing the computation cost, substituting integer arithmetic for floating-point arithmetic. The combination of these two techniques presents a unique problem when compressing sparse parameters, which we discuss in detail in Section II. Because the values themselves are compact (usually 8 bits), any metadata used to locate the nonzero values in the sparse format must be similarly compact to avoid overcoming the storage gains of the sparse representation.

Common formats for irregular sparse data like Compressed Sparse Row (CSR [3]) or Compressed Sparse Column (CSC [4]) as well as variants like Block Compressed Sparse Row (BCSR [5]) are not well suited for quantized data because the storage cost of the indexing overhead quickly overcomes the benefits of storing only the nonzero values even at low sparsity. We propose instead an intuitive extension of CSR along with an efficient sparse convolution algorithm specifically tailored to quantized data computation and a simple memory-side accelerator[1] to hide the cost of sparse metadata overhead.

In this work we make the following contributions.

- *Memory-Side Accelerator* Many recent works propose new data compression formats and algorithms for DNN's along with specialized hardware to accelerate the entire computation. We propose to aid the primary processing cores only with indexing and other memory-side operations for common DNNs using sparse data. Our memory-side accelerator (*HHT*) supplies the primary cores *only the required data*, and does not fetch out-of-bounds data.
- *New Sparse Representation.* There are many ways to represent sparse data but they require additional meta-data to specify the location of a nonzero value in a structure (for example, the column location in a row of a sparse matrix). However, when the data is quantized, as is common in TinyML environments, the meta-data may be several times larger than the size of nonzero values. For example, when using 8-bit quantized values, the indexes needed to specify the location of a nonzero may be 32-bits. We propose to partition the data such that the location of a nonzero value within a partition can be represented with 8-bits. Our partitioning is specifically designed to aid convolution algorithms and SIMD like packed arithmetic that operate on four 8-bit values.
- *Specialized RISC-V Instructions.* Since our memory-side accelerator (*HHT*) is used only to aid in fetching the data needed by the primary core, the accelerator core can be

[1] The accelerator is implemented as a RISC-like core with a reduced instruction set.

2643-3001/22/$31.00 ©2022 IEEE
DOI 10.1109/SBAC-PAD55451.2022.00019

very simple RISC-V-like core with minimal instruction set including integer only arithmetic, fewer registers and new address formats.

The rest of the paper is organized as follows. Section II provides the background to motivate our work. Section III describes of our new sparse data representation and Section IV describes our convolution algorithm that takes advantage of the compressed quantized data. Section V includes details of the memory-side accelerator *HHT*. Section VI describes our approach to evaluating our *HHT*. This section also includes results from our experiments and provides an analysis of factors contributing the performance of *HHT* for sparse convolutions as well as a complete network from TinyML benchmarks. A review of related works is presented in Section VII. Section VIII summarizes the contribution.

II. Background and Motivation

A. Convolution Layer

The convolution (CONV) layer is the basis for the Convolutional Neural Network (CNN). A typical CNN model contains a mix of CONV and pooling (POOL) layers with a fully connected (FC) layer before the final classification step, although the number of these layers differ from network to network. We focus our work on improving the performance of the CONV layer in particular because it represents the bulk of computations in most CNNs.

The CONV layer is composed of 4 tensors: inputs, outputs, weights, and biases. The inputs and outputs are structured as a set of 2d *feature maps*, each of which is called a *channel*. The weights are similarly structured as a stack of 2d *filters*. Each 2d input channel corresponds to a distinct 2d weight filter, and the filters are typically grouped into a single 3d filter or *kernel*, matching the total number of input channels. During the convolution, the weight kernel slides across successive windows of the input features at a set stride, and for each window, all values are element-wise multiplied and summed across all channels producing a single output result. This entire computation produces one 2d output feature map, or output channel. Each output value can be adjusted by adding a bias, which is the same for every element of the resulting output channel. Additional 3d kernels can be applied to the same input features to produce additional output channels. [6]

B. Quantization

Quantization is a common compression technique that reduces the precision of model parameters from floating-point to smaller integers, often 8 bits or less. The model accuracy is not significantly reduced by this process, but quantization allows for a more compact network with reduced storage and computational complexities. Typically quantization is performed after training a network with full precision, usually converting parameters from 32-bit floating point to 8-bit integer.

C. RISC-V P Extension

The P extension of the RISC-V ISA defines a set of packed single instruction multiple data (SIMD) digital signal processing (DSP) instructions that operate on XLEN-bit integer registers for embedded RISC-V processors [7]. These instructions are designed to increase the processing capabilities of RISC-V CPUs on DSP algorithms but they can also be utilized by quantized CONV layers because they operate on multiple 8-bit integers that are ”packed” into XLEN-bit registers. In particular, the Signed Multiply Four Bytes with 32-bit Adds (SMAQA) instruction operates on two 32-bit source registers, multiplying the four 8-bit signed integers packed into each of the registers. The results of the 4 multiplications are then added together to a signed 32-bit integer destination register (accomplishing Multiply-Accumulate of four pairs of 8-bit integers). However, packing CONV input features requires additional computational overhead because they are not always accessed in the same order they are stored in memory. Indeed when we compared the performance of dense CONV with and without packed instructions, the scalar version outperformed the SIMD version. This is further complicated when the network uses sparse data (either sparse weights, inputs or both).

D. Weight Pruning

Increasing the sparsity of weights in a CNN is achieved mainly through pruning, where redundant weight values are set to 0. As more values become 0, the sparsity of the weights increases. Numerous pruning methods have been developed, employing different strategies to determine which parameters are defined as redundant and can be pruned. In [1] and [2], for example, pruning is done through an iterative process of setting a subset of weights below a set threshold to zero and retraining the network to regain accuracy. Increasing sparsity through pruning allows for the weights to be compressed into a sparse matrix representation that reduces storage by storing only non-zero values as well as metadata required to locate those values within the original dense matrix.

E. Indexing Overhead

Although prior works explore both sparse data representations and accelerating sparse convolutions with specialized hardware, the use of 8-bit quantized parameters and specialized 8-bit SIMD (or packed) instructions presents unique challenges. The reduction in bit-width that comes from quantization is itself a significant reduction in the memory footprint of a network. If the sparse parameters are compressed naively after quantization, the indexing data needed to identify the position of a non-zero weight value can quickly overcome any storage savings from storing just the 8-bit non-zero values.

Any compression scheme involves at the very least storing the non-zero values along with at least one index to compute it’s original position within the dense matrix. CSR, for example, stores the column index of a value in a row as well as the cumulative number of non-zero values in each row. As others [8] have noted, minimizing the size of the index array (or column numbers in the case of CSR) is the primary way to minimize the total overhead of the compressed data. Extra care must be taken when dealing with quantized non-zero values that are limited to a specific bit-width. Even if index values are the same width (e.g., 8 bits for 8-bit quantized data values) the sparse representation will only save storage space (and consequently data access latency) if the sparsity of the original matrix is at least 50%.

TABLE I: Size of indexes and non-zero values over size of dense matrix for range of sparsities and index widths

Index	**Sparsity**						
bit-width	30%	40%	50%	60%	70%	80%	90%
32-bit	3.5	3	2.5	2	1.5	1	0.5
16-bit	2.1	1.8	1.5	1.2	0.9	0.6	0.3
8-bit	1.4	1.2	1	0.8	0.6	0.4	0.2
4-bit	1.05	0.9	0.75	0.6	0.45	0.3	0.15

Table I shows the total storage size of all 8-bit non-zero values and their index over the total size of the corresponding uncompressed matrix for various index bit-widths and a range of sparsities (number of zeros). This overhead applies to any matrix size. These numbers give a rough picture of the complication posed by quantized data. Even discounting any additional overhead needed to index higher-dimensional matrices (using just one index array implies the initial matrix was a 1d vector), it is more efficient to store weights in their uncompressed form if the sparsity is low. Using 32-bit indexes requires at least 80% sparsity to see any storage savings from compressed representation. Similarly, 16-bit indexes require 70% sparsity to achieve storage efficiency. On the other hand, using 8-bit indexes sparse representation is efficient at 50% sparsity and above and 4-bit indexes are efficient at even lower sparsities. Naturally, the range of sparsity in neural network layers varies from model to model, but limiting indexes to 8 bits allows for quantized sparse data to have greater storage efficiency than the uncompressed form at sufficiently low sparsities for most real-world examples.

Although prior works have approached the problem of reducing the size of sparse indexes, few have addressed the problem specifically in terms of quantized data. Our goal is to design efficient inference capabilities with micro-controllers that have very limited storage and computing capabilities, as well as stringent power limitations. To achieve this goal we decided to use quantized data and defined a new compression format where indexes are limited to 8 bits. We also designed a memory-side accelerator that provides only the needed data to the primary core, eliminating the indexing computations from the primary core. The next few sections provide details of our contributions.

III. *PSR* Compression Overview

The Partitioned Sparse Representation is similar to widely used Compressed Sparse Row (CSR) format, with modifications that make it better-suited to convolution with quantized weights. First, *PSR* is designed so that all indexing overhead values are restricted to a specific bit-width. This width is adjustable depending on the needs of the application, but for our experiments we set the bit-width at 8, because this is the minimum width with which the RISC-V instruction set currently can operate. In the next two sections, we describe the layout of the sparse representation for convolution weights as well as its use within the convolution kernel.

With *PSR*, the entire (typically 3d) kernel for each weight channel of a convolution layer is flattened and then partitioned into equal sub-matrices. The size of the partition is chosen based on the desired indexing bit width. For 8-bit indexes, the partitions can be no larger than 256 values. The partition size is further limited by the total size of a single 3d weight kernel for a given CONV layer. To minimize computational overhead, partitions should be aligned to the beginning of each kernel and must divide the kernel into equal sub-arrays. The compression scheme consists of 3 arrays: a *vals* array, *offset* array, and *nnz* array. The *vals* array stores only the nonzero values of the matrix. The *offset* array store the relative offset into a partition of each corresponding non-zero value. And finally, the *nnz* array stores the number of values in each partition. Therefore, while the lengths of both the *vals* and *offset* arrays correspond to the total number of non-zero values in the dense matrix, the length of the *nnz* array is equal to the number of partitions.

The steps to compress a dense matrix of weights into *PSR* format are as follows:

- Starting with the uncompressed 3d weight kernels (one per output channel as shown in Figure 1), each kernel is flattened into a 1d vector.
- Next, each flattened kernel is divided into one or more partitions, depending on the size of the kernel and the desired bit-width of the indexes. For example, if 8-bit indexes are required, the partition size is limited to 256 so that the position of any given element can be identified with an 8-bit index.
- For each partition, the number of non-zero values are stored along with their relative offset from the start of the partition in the *vals* and *offset* arrays respectively.
- Finally, as values are stored in the *vals* array, the total count is accumulated for each partition and stored in the *nnz* array in the position corresponding to the partition number.

Figure 2 shows steps 2-4 above for a small example matrix. The diagram on the left shows the flattened dense vector for each weight kernel while the diagram on the right shows the resulting compressed sparse storage representation.

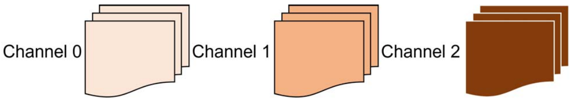


Fig. 1: Dense weight kernels (one for each output channel) to be flattened and separately partitioned

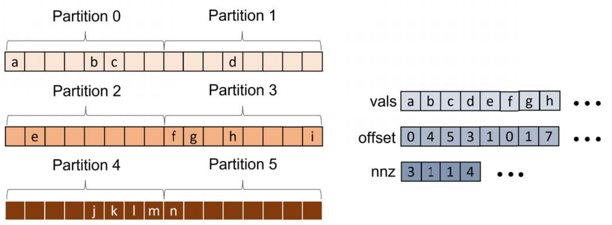


Fig. 2: *PSR* compression of dense weights with 3 channels and partition size of 8 elements

IV. Sparse Convolution Algorithm with *PSR*

In our design, input data is not compressed as we feel that the overhead of dynamically compressing input data and then use indexing metadata to match inputs to weight values may negate any storage savings. It is now necessary to match an input value (from dense input matrices) with a non-zero weight value in our PSR format.

The steps required to compute a single CONV output value are shown in Algorithm 1. The outer loop structure is identical to the dense algorithm and each output is computed one at a time to maximize weight reuse. For each partition in an output channel, the non-zero weights are processed in storage order. Algorithm 2 shows how to compute the position of the input element corresponding to each non-zero weight in the convolution algorithm with *PSR* and pack 4 matching values into either a buffer in the case of *HHT* or into a 32-bit value that can then be loaded into a CPU register. For each non-zero weight, the position of the value within the dense matrix is identified on lines 3-5. That position is then converted in lines 6-7 into the actual input position by adding the start index of the current convolution window as well as adjusting for factors such as convolution stride and dilation. Finally once the corresponding input element is identified, the 8-bit value can be loaded from memory and packed into 32-bit SIMD value that is fed to the packed instruction.

Algorithm 1 *PSR* CONV

```
for out_index ← 0 to total_outputs−1 do
    acc ← 0
    for p ← 0 to parts_per_channel−1  do
        v_end ← nnz[u×parts_per_channel+p]
        for v ← 0 to v_end do
            PACK_INPUT( )
            SMAQA
            packed_weight_addr ← packed_weight_addr + 4
            v ← v + 4
        end for
    end for
    output_tensor[out_index] ← acc
end for
```

Algorithm 2 PACK_INPUT( )

```
1: for v ← start_index to start_index+3 do
2:     index ← offset[v]
3:     filter_i ← index / stride_i % filter_height
4:     filter_j ← index / stride_j % filter_width
5:     filter_k ← index / stride_k % input_depth
6:     in_i ← out_i + filter_i
7:     in_j ← out_j + filter_j
8:     packed_val[v] ← input_tensor[i][j][filter_k]
9: end for
```

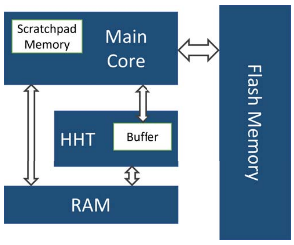


Fig. 3: *HHT* in an embedded micro-controller

In our architecture, this algorithm is offloaded to a memory-side accelerator called *HHT*. The accelerator supplies a packed set of input values as described in the algorithm to the CPU via a shared buffer. The main processor retrieves these values along with non-zero weights (which are in consecutive bytes of the *vals* array of our *PSR* format) and performs multiply-accumulate to calculate output values.

V. Design of *HHT*

The *HHT* accelerator is modeled as simple near-memory RISC-V core. This core is designed to be smaller than the main CPU. When in use for the algorithm outlined in this paper, the *HHT* accesses the cache or SRAM available on chip to load inputs, while the weights are stored in a separate, faster scratchpad memory for sequential use by the main CPU. Additionally, because the CPU is not using any of the indexing overhead for the weights, that metadata is stored in the main memory for use by the *HHT*. This design helps to limit the needed size for the reserved scratchpad memory.

Because the *HHT* core is used only for memory indexing computations, it can be much simpler than the main core. All it needs is the baseline RISC-V integer instruction set without any additional extensions. The core can be paired with another simple main core (as we focus on MCU environments) or one that is more complex as the system requires. In our tests, neither core needs floating-point instructions because the quantization scheme allows integer-only inference.

Figure 3 shows the position of the *HHT* core within an embedded micro-controller situated between the main core and memory. *HHT* can be embedded in memory or placed very closed to memory (either SRAM or cache memory if available). The *HHT* core and the main CPU core communicate via a set of control flags and memory-mapped registers that contain the metadata needed by *HHT*. These registers will contain values, such as matrix dimensions and base addresses for arrays depending on the specifics of the algorithm and *HHT* kernel. For the sparse CONV, this metadata includes the weight matrix dimensions, addresses for the dense input array and sparse index and nnz arrays, as well as array stride values needed to compute filter offsets. The CPU is responsible for loading the necessary data into the registers before initiating *HHT* with the *start* flag.

The programming model for the *HHT* accelerator requires a separate kernel that is compiled and loaded into the instruction cache of the *HHT*. This kernel defines the metadata computations assigned to *HHT* as required by a given algorithm. In our sparse CONV algorithm using *PSR* compression outlined in Section IV, for example, the *HHT* kernel consists of all the instructions in the inner-most loop body of the convolution required to compute address of each input feature and load it into the shared buffer for processing by the main core.

Additionally, we envision ISA optimizations for the *HHT* core to further reduce its complexity. Since *HHT* core deals with memory accesses, it only needs integer arithmetic and may be restricted to fewer registers (say 16). Because the RISC-V 32-bit ISA allows for up to three register operands, there are many examples of certain blocks of instructions that can be combined. For example, if a new load instruction of the type *lw R-destination,R-index, R-offset* were implemented in RISC-V, we can replace *add a5,a3, a5* and *lw a5, 0(a5)* with a single instruction *lw a5, a3, a5*. Or a shift operation on an index (to obtain byte offset of a value) can be included in a memory access, for example, *load-with-shift-left a5, a3, a5, shift-amount* where the address is obtained by adding contents of *a3* with left shifted value of *a5*. If we permit very small shift amounts like 1, 2, 3 to obtain byte offsets for 2, 4 or 8 byte data sizes, such instructions can be easily implemented in conventional RISC-V pipelines. More complex instructions may also be worth considering for memory-side accelerators such as our *HHT*, including combining multiple instructions that use the same functional unit (e.g., division and remainder operations like those seen in the index computation kernel for our sparse CONV algorithm), or indirect memory addressing. Customizing the instruction set in such a way has the potential for significant energy savings if they are repeated throughout the course of a program's execution. Though these latter types of instructions are not as trivial to implement, they are not infeasible, especially with the aid of tools like Google's Custom Functional Unit [9].

VI. Experimental Evaluation

We evaluated the performance of our sparse CONV algorithm using the RISC-V spike simulator [10] to model an embedded micro-controller environment. Spike is configured to incorporate the programmable *HHT* accelerator. Because *HHT* is modeled as a low-power core that is similar in function and design to the main CPU core, the cycle delays for the *HHT* core to load a single value into the shared buffer match those of the main core. Both cores require the same number of instructions to compute the index and a single input to its corresponding weight value. We further extended Spike to provide a cycle-accurate timing simulation that enabled us to collect total execution cycles, total *HHT* cycles, and total cycles (if any) the CPU spends waiting for the accelerator to supply data.

The system configuration we used in our evaluation consists of a 1MB SRAM along with a 1.1 GHz main core. The main core uses 32-bit RISC-V base integer architecture as well as compressed, atomic, multiply, and packed-SIMD extensions. Floating-point instructions are not required for the quantized CONV as all scaling multiplication can be accomplished with fixed-point integer arithmetic as outlined in [11]. The primary core uses an in-order 3-stage pipeline implementation. The *HHT* core utilizes only the base integer architecture and runs at a similar frequency to the main core.

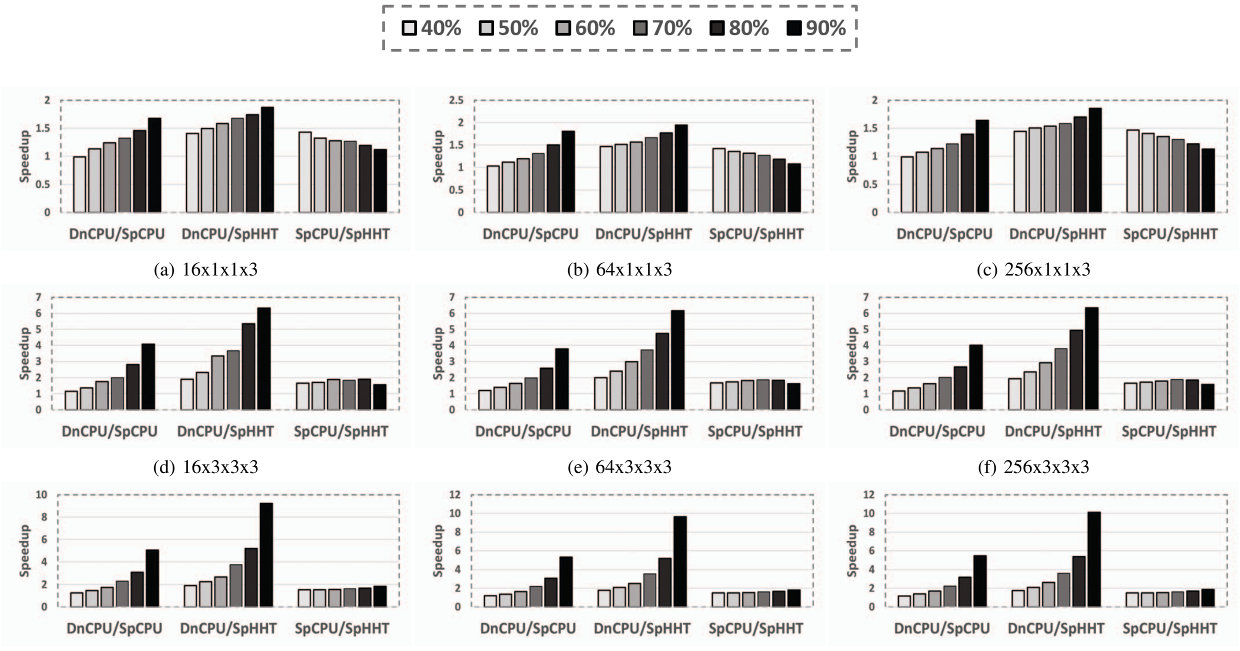
A. Workloads

We evaluated our sparse CONV algorithm both with and without the aid of *HHT* using synthetic CONV weights of varying sizes and sparsities. We also ran a full end-to-end network from the MLPerf Tiny Deep Learning Benchmarks for Embedded Devices [12]. Specifically, we performed image classification with *resnet* on the CIFAR-10 dataset [13]. We evaluated both pruned and unpruned versions of the network.

B. Convolution with Synthetic Weights

To understand the relative effect of PSR compression, the optimized CONV algorithm described in Algorithm 1, as well as the benefits of off-loading indexing to our memory-side accelerator *HHT*, we measured the performance of both CPU-only and *HHT* (i.e, CPU + *HHT*) versions of our packed SIMD sparse CONV algorithm with *PSR* compression of quantized values relative to a dense baseline implementation (not using *PSR* compression). We used synthetic CONV inputs for a range of weight dimensions and sparsity levels for this assessment. Additionally, we measured the benefit of adding custom RISC-V instructions to the ISA of the accelerator by modifying the cycle delay for *HHT* to supply a buffer value. The reduced delay corresponds to a reduction in instruction count after using the instructions outlined in Section V. We tested both the optimized and non-optimized versions of *HHT* on the same synthetic benchmarks, and present here the results of the optimized experiments. Figure 4 shows the performance gains for 9 different weight matrix sizes as overall weight matrix sparsity (or percent of zero values) is increased from 40% to 90%. The *four* values associated with each sub-graph designate the shape of each weight matrix and correspond to the number of output channels (same as the number of 3d weights kernels), the 2d weight filter height and width, and the number of input channels respectively. For example, 64x3x3x3 indicates 64 output channels, a 3x3 filter and 3 input channels, as shown in Figure 4(e).

We evaluated *three* different weight filter sizes along with *three* distinct total output channels for each filter size. In each sub-graph of Figure 4, we compare (1) the performance of CPU-only execution using our quantized compressed PSR Algorithm 1 against a baseline of CPU performing all computations on dense weights (the first set of bars of each sub-figure); (2) the performance of *HHT* version (CPU+*HHT*) using PSR sparse Algorithm 1 against the dense baseline (the second set of bars in each sub-figure); and (3) the performance



(a) 16x1x1x3

(b) 64x1x1x3

(c) 256x1x1x3

(d) 16x3x3x3

(e) 64x3x3x3

(f) 256x3x3x3

(g) 16x5x5x3

(h) 64x5x5x3

(i) 256x5x5x3

Fig. 4: Speedup of sparse *PSR* CONV with (SpHHT) and without (SpCPU) *HHT* over dense packed baseline (DnCPU) for selected weight kernel sizes and sparsities (40%-90%).

of optimized *HHT* version against CPU-only using *PSR* sparse Algorithm 1 (the last set of bars). Intuitively, as the sparsity increases, so does the performance of both sparse versions over the dense baseline in all cases (the first two sets of bars in each sub-figure of Figure 4). This data shows the benefits of our sparse format as it reduces computations performed to only when non-zero weights are processed by the CPU. Without any indexing overhead from sparse compression, the ideal speedup of the sparse algorithm over dense would be directly proportional to the sparsity. As an example, at 50% sparsity this would be 2x because the CPU is performing half the amount of multiply operations. With the aid of the *HHT* accelerator hiding the cost of indexing overhead arising from *PSR* representation of weights, and by optimizing the instructions used in the accelerator, the performance gains shown in the third set of bars match the ideal case for larger kernel sizes. The number of output channels had minimal impact on the relative performance of each version of the algorithm. As the filter size increases, however, sparse CONV with *PSR* performs increasingly better than the dense version, and the relative speedup for the *HHT* version over CPU-only (the last set of bars in each sub-figure) increases as well. This trend is clearly seen in Figure 4 as the bars are taller in each successive row. For kernel sizes of 5x5x3, the largest we tested, the sparse CONV algorithm achieves 5x speedup over the dense version (baseline) for high sparsities. Overall, CONV with optimized *HHT* acceleration saw 10x speedup over the dense baseline and 1.8x speedup over the sparse CPU-only version for the highest matrix sizes and sparsity levels. These results are only slightly better than what we saw with the un-optimized *HHT*, which achieved 9x and 1.7x speedups for the same measures. It should be noted that even if we used two cores for CPU-only implementation, the maximum performance gain will be 2x, which is rarely the case for most two-threaded implementation of applications.

As explained in Section I, the partition size of the *PSR* matrix compression depends on the 3d filter size. For sub-graphs (a) - (c) that number is 3, or 1x1x3. It increases to 27 (3x3x3) for sub-graphs (d) - (f) and to 75 (5x5x3) in sub-graphs (g) - (i). For both the CPU-only and *HHT* versions, this size determines how many input values can be processed in a single partition loop. If this size is too small, as is the case with the 1x1x3 filter, the added computational overhead of processing each compressed partition overwhelms the benefits of sparsity arising from a reduction in the number of multiplications. The use of SIMD instructions further limits the performance of small partition sizes. Because the packed instructions used in the sparse CONV algorithm operate on *four* (8-bit) input and weight values at a time, any partition size that is not a multiple of 4 requires zeros to be packed into the packed registers. Partition sizes *less than four* are the worst case from a performance perspective because the cost of these extra zeros is amortized as the size increases. Therefore performance gains will not be as high for network layers with small weight kernels as shown in the first row of Figure 4. Fortunately, though it is common for networks to utilize CONV layers with a 2d weight filter size of 1x1, the number of filters in a 3d kernel is often large enough to necessitate an efficient *PSR* partition size. As an example, half of the layers in the MobileNet visual wake words [14] benchmark from MLPerf Tiny, are CONV layers with 1x1 filter size. The channel dimension in all cases, however, is at least 8 and as high as 256 in the last layers.

The partition size also determines the maximum number of values that *HHT* can load into the shared buffer at one time (supplying input values to the CPU). Therefore a smaller size also limits any potential additional performance gains from offloading index processing to the memory-side accelerator. Likewise, as the sparsity increases, the average number of non-zero values in each partition decreases, further reducing the work available for *HHT*. This is why the relative performance gains of the *HHT* over the CPU-only version shown in the right-most set of bars of each sub-figure decreases for 1x1x3 kernels as the sparsity increases and also decreases for 3x3x3 kernels at 90% sparsity. The larger window sizes of (g) - (i) ensure that the relative performance increases along with sparsity even for the highest percentages.

C. CPU Wait Cycles

The benefits of a memory-side accelerator such as *HHT* depends on the amount of work offloaded to the accelerator and the amount of execution that can be overlapped with CPU computations. As already discussed in the previous section, if insufficient amount of work is offloaded to *HHT*, we see minimal performance gains. On the other hand, if too much work is assigned to *HHT*, CPU may be idling. The goal is to overlap the execution of the two cores as much as possible. Perfect overlap would result in the indexing computations associated with sparse compression being completely hidden from the total execution time. In reality, such a perfect balance is difficult to achieve. This is in part due to CPU waiting for *HHT* to provide packed input values in shared buffer. Because there is only a single buffer shared between the two cores, the *HHT* must finish writing to the buffer before the CPU can read any values. If the CPU is ready to process input values before the buffer has been completely filled, the CPU must wait for *HHT*. The amount of time CPU is idling can be measured from the number of cycles CPU is waiting for *HHT* as shown in Figure 5, which shows the fraction of total execution time the main core idles. These wait cycles were measured with the optimized *HHT*, as the primary benefit of the optimization is a reduction in the number of cycles the accelerator needs to locate and load each input values into the shared buffer. Because *HHT* can fill the buffer more quickly, the CPU spends fewer cycles waiting over the course of program execution. For all weight kernel sizes tested, the CPU spends more time waiting at lower sparsities because more work is being offloaded to the accelerator[2]. As the sparsity increases to 80% and 90%, the wait times drop almost to 0, which is reflected in the improved performance data. If the CPU never waits for buffer values, the speedup of the *HHT* version of CONV should approach 2x over the CPU-only version, which is what we see. Similarly 50% wait cycles (i.e., the CPU spends half of the total execution time waiting) should result in a speedup of about 1.5x, which is what we see for sparsities at 40% and 50%.

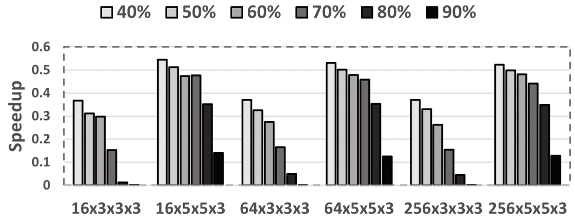


Fig. 5: Portion of total execution cycles the CPU is waiting for HHT to supply needed input values

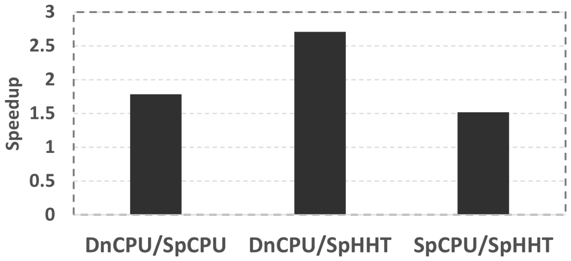


Fig. 6: Performance Comparison of CPU-only and *HHT* Versions of Resnet

D. End-to-End Image Classification Inference

The performance benefits of *PSR* CONV with *HHT* acceleration are best understood in real-world context. We therefore compared execution time of a pruned Resnet image classification benchmark both with and without *HHT*. Table II shows the size of weight kernels for each CONV layer in the network. Using the Tensorflow pruning API [15], we fine-tuned the pre-trained Resnet model with polynomial decay pruning schedule to a final sparsity of 80% for all layers. This sparsity target, while high, only results in a slight reduction of model accuracy on the test set from 87% to 85%. Likewise the AUC-ROC Curve (an important metric that measures the quality of a classification model's predictions) dropped less than 0.5%. Detailed pruning strategy is beyond the scope of this work. We report the accuracy merely to show that it is possible to aggressively prune weights to high sparsity without a significant drop in model performance.

Figure 6 compares the average total execution cycles for both CPU-only and *HHT* to complete an end-to-end inference pass for the pruned Resnet model. Both versions utilize the sparse *PSR* algorithm for all CONV layers in the network. The CPU-only version achieves close to 1.8x speedup over the dense baseline, while the *HHT* versions achieves over 2.7x speedup. As the right-most column of the graph shows, the relative speedup for *HHT* over CPU-only is about 1.5x. These results match our performance measurements with synthetic

[2] We did not show wait cycles for 1x1x3 weight kernels since as previously discussed, the amount of work offloaded to *HHT* is very small and *HHT* can supply the single value needed for each kernel without delays.

CONV weights for similar matrix sizes and sparsities (shown in Figure 4).

TABLE II: Resnet CONV Weight Dimensions

CONV Layer	Weight Dimensions
Layer 1	16x3x3x3
Layer 2	16x3x3x16
Layer 3	16x3x3x16
Layer 4	32x3x3x16
Layer 5	32x3x3x32
Layer 6	32x1x1x16
Layer 7	64x3x3x32
Layer 8	64x3x3x64
Layer 9	64x1x1x32

E. Energy Estimation

We estimated the relative energy of a CPU-only system with our programmable *HHT* design using energy estimates for Riscy and Micro-Riscy cores [16]. If we assume that the main core is a Riscy core and *HHT* is similar in footprint to the Micro-Riscy core, we can compute an estimate of total energy for a single end-to-end Resnet inference pass with the total cycles executed by each core. Assuming *HHT* is 2x more energy efficient than the main core [16], we can estimate the energy for the *HHT* system as:

$$\text{HHT } \textit{Cycles} \times 0.5 + \textit{Total Execution Cycles} - \textit{CPU Wait Cycles} \quad (1)$$

The relative energy efficiency of *HHT* over its counterpart is then computed as:

$$\textit{Total CPU-only Cycles } / \textit{ Normalized CPU}+\text{HHT } \textit{Cycles} \quad (2)$$

Using these calculations, we estimate that using *HHT* to offload indexing computations increases the energy efficiency by 70%.

F. Summary and Discussion

When evaluated against a dense baseline, our sparse CONV algorithm with *PSR* compression achieves significant speedup for higher sparsities and weight matrix sizes, even without offloading the sparse representation indexing overhead. The addition of the *HHT* accelerator further improves the performance gains, approaching the maximum possible speedup for a two core system of 2x. We highlighted the additional advantage of *HHT* as a minimal-functionality core in terms of overall energy efficiency. Most importantly, we have shown that an end-to-end inference example using a representative image classification model can be pruned to the high sparsities where our design shows the most benefit and performance gains seen in the full example matched what we saw with our synthetic benchmarks.

VII. Related Research

In our previous work, we explored memory side acceleration for non-quantized data sets using both ASIC HHT [17] and programmable RISC core [18].

Sparse accelerators. Much work has been done on compression for DNNs, but the unique challenges posed by quantized data and SIMD processing have not been adequately addressed. dCSR [8] proposes to address the issue of indexing bit-width by altering CSR format to store an index *delta* rather than the column index itself. This *delta* is calculated as the difference between the actual column index of a non-zero value and the average index for a matrix row. This format tackles the same issue of compressing the bit width of sparse column indices, but is much more complicated to implement in a streaming run time. Our solution is simpler and more flexible because 1) the compression algorithm does not require advance knowledge of the matrix layout as in dCSR and 2) our storage format is agnostic to the initial shape of the dense matrix.

Kwon et al. [19] employ a software compression format that is very similar to PSR. They compress input feature maps of convolution layers using a two-step software compression that, like PSR, first flattens the input matrix into a 1D vector and then partitions the vector into 256-element chunks. Only the non-zero elements are stored along with their relative index into their respective chunk and a count table that keeps a cumulative total of non-zero elements. One major limitation of this compression technique as described is that to maximize the benefits of index bit-width compression over other similar formats when using 8-bit or lower quantized data, the size of the entire dense feature map would be limited to 2^{16} elements which can be exceeded by some quantized models. In contrast, storing the non-cumulative non-zero count for each partition has the advantage that there is no maximum size and the totals can be stored in 8-bits, thus further reducing the indexing overhead. Additionally, the choice of compressing inputs rather than weights limits the usefulness of the compression format, especially when using SIMD support. The representation is inherently streaming in that the computational overhead is minimized when the non-zero values are accessed sequentially every time. By compressing the weights rather than the inputs, we can process one output element at a time and eliminate the need for any buffering of partial outputs. [19] also doesn't address quantization. In our view, the use of quantized data is what necessitates the PSR compression scheme to begin with. Because the authors are still dealing with 32-bit data values, they don't take full advantage of the benefits of such a compression format.

Our work is the only one that combines the advantages of quantization and pruning with both SIMD instructions and a compression format that minimizes the bit-width of indexes to match the bit-width of the quantized values. All of this is accomplished with a programmable accelerator model that does not need to be tailored to the specifics of the algorithm or even the compression format.

Prefetchers. There are several studies on data prefetching into on-chip cache to hide the memory latency problem. Most of the early works [20]–[24] are sequential stride-based prefetchers. Some more recent prefetchers can even be programmed or learn to handle irregular accesses [22], and those used by Apple M1 systems. However, in most cases prefetchers are unaware of program specific structure bounds (or array bounds) and this can be exploited by security attacks as reported in [25]. Our *HHT* is programmed not only to understand access patterns but also with memory bounds for different structures, limiting the ability of out of bounds attacks.

Helper Threads. While there have been many prior studies in terms of *decoupling* or *off-loading* memory access operation (consider an early decoupling work reported in [26]), *HHT* is a flexible hardware which can be programmed to process application specific metadata processing. Helper threads (particularly software threads) have been used to aid primary threads with some operations (for example see [27]). Such software techniques may not lead to performance gains if the threads are scheduled on different cores requiring cache coherency related overheads.

New Sparse Representations. In a different vein, there have been proposals on improving compression of sparse matrices and proposed techniques including hierarchical bit vectors [28] or compression on top of CSR [29]. There are proposals for specialized hardware to compress and decompress data for use by CPU (assuming that the CPU uses conventional *SpMV* software) [29]. Others propose hardware for new compression formats (such as hierarchical bit maps) for performing sparse matrix computations [28]. We programmed *HHT* to handle sparse data represented using SMASH [28] format. SMASH format requires complicated indexing to locate the row and column positions of non-zero values of a sparse matrix. This implies that *HHT* for SMASH is performing more work than the CPU, causing CPU to idle. Moreover, we feel that SMASH format may not be suitable for embedded systems.

Accelerators for Machine Learning. Interest in DNN based accelerators have seen a rise in recent years, leading to many specialized hardware accelerators, too many for us to include here. Many of these specialized accelerators based on either dataflow or tensor/systolic arrays that lack flexibility or reconfigurability [30]–[36]. These accelerator either rely on very specialized sparse data representations or implement specific DNN algorithms. Our *HHT* only aids in memory-side operations and does not perform actual computations. In this contribution we focused on quantized data used in TinyML applications. Our *HHT* is programmable and can be used with different compression formats and for different DNN algorithms.

Several works focus on accelerating sparse matrix-dense vector multiplication (*SpMV*) operations [37]–[41], We only proposed to aid in index computations for any sparse data based algorithm instead of accelerating the actual computation. While [42] and [43] are somewhat similar to our work, they expand sparse data into dense data so that the primary cores can rely on simple algorithms; our *HHT* only provided required or matching non-zero values needed for the computation, still keeping the primary core computations simple.

VIII. Conclusions

In this paper, we explored CNN inference in the context of low-power, resource constrained devices. Low memory footprint is vital on such devices, which motivates the use of sparse data stored in a compressed representation. Storing data in this way introduces computational overhead and so we introduce a programmable memory-side accelerator called *HHT*. *HHT* offloads the sparse data overhead, supplying the main core with only the data it needs for computation. We also presented *PSR*, a storage-efficient sparse compression format specifically for low bit-width quantized data, as well as a corresponding CONV algorithm that utilizes weights stored in *PSR* format. Finally, because *HHT* core is selected with the minimal instruction set required, we outlined new address formats for certain RISC-V instructions to further reduce the required instructions and increase system efficiency. We have shown that our design along with the *PSR* CONV algorithm outperforms the baseline in terms of execution time as well as storage and energy efficiency. Our evaluations show as much as 10x speedup for sparse CONV with *HHT* over a baseline of the CPU performing all computations on dense quantized data. *HHT* performs 2.7x faster on Resnet inference over the dense baseline and gives 70% energy savings over sparse CONV with CPU performing all computations. Because the instruction set optimizations we modeled yielded limited performance benefit, further exploration of that design space is left for future work. We conclude that sparse *PSR* CONV with *HHT* acceleration is an intuitive and efficient method for improving the performance and storage efficiency of NN inference on low-power devices.

Acknowledgements This research is supported in part by a SRC grant AIHW Task 2943 and a NSF grant #1828105. The authors acknowledge Nagendra Gulur (TI), Mahesh Mehendale (TI), Chris Tsongas (TI), Jaekyu Lee (ARM) as well as their UNT colleagues Charles Schelor and Shashank Adavally for their valuable suggestions.

References

[1] Song Han, Jeff Pool, John Tran, and William J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[2] Song Han, Huizi Mao, and William J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015.

[3] Nathan Bell and Michael Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. 2009, ACM.

[4] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 2011, pp. 721–733, IEEE.

[5] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, Friedhelm Meyer auf der Heide and Michael A. Bender, Eds. 2009, pp. 233–244, ACM.

[6] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017.

[7] RISC-V Foundation task group, "Risc-v p extension spec proposal," https://github.com/riscv/riscv-p-spec.

[8] Elias Trommer, Bernd Waschneck, and Akash Kumar, "dcsr: A memory-efficient sparse matrix representation for parallel neural network inference," 2021.

[9] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby R. Banbury, Alan V. Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi, "CFU playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas," *CoRR*, vol. abs/2201.01863, 2022.

[10] Abraham Gonzalez, "The RISC-V ISA Simulator," 2019.

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017.

[12] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al., "Mlperf tiny benchmark," *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[13] Alex Krizhevsky and Geoffrey Hinton, "Learning multiple layers of features from tiny images," Tech. Rep. 0, University of Toronto, Toronto, Ontario, 2009.

[14] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes, "Visual wake words dataset," *CoRR*, vol. abs/1906.05721, 2019.

[15] Martín Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, Software available from tensorflow.org.

[16] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[17] Pranathi Vasireddy, Krishna Kavi, and Gayatri Mehta, "Sparse-t: Hardware accelerator thread for unstructured sparse data processing," *International Conference on Computer-Aided Design (ICCAD '22)*, 2022.

[18] S. Adavally, A. Weaver, P. Vasireddy, K. Kavi, G. Mehta, and N. Gulur, "Heterogeneous architecture for sparse data processing," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Los Alamitos, CA, USA, jun 2022, pp. 6–15, IEEE Computer Society.

[19] Jisu Kwon, Joonho Kong, and Arslan Munir, "Sparse convolutional neural network acceleration with lossless input feature map compression for resource-constrained systems," *IET Computers & Digital Techniques*, vol. 16, no. 1, pp. 29–43, 2022.

[20] John W. C. Fu, Janak H. Patel, and Bob L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Los Alamitos, CA, USA, 1992, MICRO 25, pp. 102–110, IEEE Computer Society Press.

[21] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[22] Xiangyao Yu, Christopher Hughes, Nadathur Satish, and Srinivas Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*. IEEE, 2015, pp. 178–190.

[23] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, Feb 1999.

[24] Mahzabeen Islam, Soumik Banerjee, Mitesh Meswani, and Krishna Kavi, "Prefetching as a potentially effective technique for hybrid memory optimization," in *Proceedings of the Second International Symposium on Memory Systems*, New York, NY, USA, 2016, MEMSYS '16, pp. 220–231, ACM.

[25] J. Sanchez Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, Los Alamitos, CA, USA, may 2022, pp. 1518–1518, IEEE Computer Society.

[26] James E Smith, "Decoupled access/execute computer architectures," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.

[27] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin, "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1309–1324, 2008.

[28] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri-Ghiasi, Taha Shahroodi, Juan Gómez-Luna, and Onur Mutlu, "SMASH: co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. 2019, pp. 600–614, ACM.

[29] Arjun Rawal, Yuanwei Fang, and Andrew A. Chien, "Programmable acceleration for sparse matrices in a data-movement limited world," in *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pp. 47–56, IEEE.

[30] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[31] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016.

[32] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2017, ISCA '17, p. 27–40, Association for Computing Machinery.

[33] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 781–795.

[34] Pengcheng Dai, Jianlei Yang, Xucheng Ye, Xingzhou Cheng, Junyu Luo, Linghao Song, Yiran Chen, and Weisheng Zhao, "Sparsetrain: Exploiting dataflow sparsity for efficient convolutional neural networks training," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[35] E. Qin, A. Samajdar, H. Kwon, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," Feb 2020.

[36] T. Moreau, T. chen, L. Vega, J. Roesch, E. yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware-software blueprint for flexible deep learning specialization," *IEEE Micro*, Sept/Oct 2019.

[37] Aydin Buluç and John R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM J. Scientific Computing*, vol. 34, 2012.

[38] Joseph L. Greathouse and Mayank Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Piscataway, NJ, USA, 2014, SC '14, pp. 769–780, IEEE Press.

[39] Ariful Azad and Aydin Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 2017, pp. 688–697, IEEE Computer Society.

[40] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry T. Pileggi, and Franz Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. 2019, pp. 347–358, ACM.

[41] Leonid Yavits and Ran Ginosar, "Sparse matrix multiplication on CAM based accelerator," *CoRR*, vol. abs/1705.09937, 2017.

[42] Adrián Barredo, Jonathan C Beard, and Miquel Moretó, "Poster: Spidre: Accelerating sparse memory access patterns," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 483–484.

[43] Shashank Adavally, Nagendra Gulur, Krishna Kavi, Alex Weaver, Pranoy Dutta, and Benjamin Wang, "Express: Simultaneously achieving storage, execution and energy efficiencies in moderately sparse matrix computations," in *The International Symposium on Memory Systems*, 2020, pp. 46–60.