Eureka: Efficient Tensor Cores for One-sided Unstructured Sparsity in DNN Inference

Ashish Gondimalla*
agondima@google.com
Google
Sunnyvale, CA, USA

Mithuna Thottethodi
mithuna@purdue.edu
Purdue University
West Lafayette, IN, USA

T. N. Vijaykumar
vijay@purdue.edu
Purdue University
West Lafayette, IN, USA

ABSTRACT

Deep neural networks (DNNs), while enormously popular, continue to place ever higher compute demand for which GPUs provide specialized matrix multipliers called *tensor cores*. To reduce the compute demand via sparsity, Nvidia Ampere's tensor cores support 2:4 structured sparsity in the filters (i.e., two non-zeros out of four values) which provides uniform 50% sparsity without any load imbalance issues. Consequently, the sparse tensor cores maintain (input or output) operand stationarity, which is fundamental for avoiding high-overhead hardware, requiring only one extra 4-1 multiplexer per multiply-accumulate unit (MAC). However, 2:4 sparsity is limited to 2x improvements in performance and energy without loss of accuracy, whereas unstructured sparsity provides 5-6x opportunity albeit while causing load imbalance. Previous papers on unstructured sparsity incur high hardware overhead (e.g., buffering, crossbars, scatter-gather networks, and address calculators) mainly due to sacrificing operand stationarity in favor of load balance. To avoid adding high overheads to the highly-efficient tensor cores, we propose *Eureka*, an efficient tensor core for unstructured sparsity. *Eureka* addresses load imbalance via three contributions: (1) Our key insight is that a slight weakening of output stationarity achieves load balance most of the time while incurring only a modest hardware overhead. Accordingly, we propose *single-step uni-directional displacement (SUDS)*, where a filter element's multiplication can either occur in its original position or be displaced to a vacant MAC in the adjacent row below while the accumulation occurs in the original row to restore output stationarity. SUDS is an offline technique for inference. (2) We provide an optimal algorithm for work assignment for SUDS. (3) To achieve fewer bubbles in the tensor core's systolic pipeline due to the irregularity of unstructured sparsity, we propose offline *systolic scheduling* to group together the sparse filters with similar, statically-known execution times (based on the number of non-zeros). Our evaluation shows that *Eureka* achieves 4.8*x* and 2.4*x* speedups, and 3.1*x* and 1.8*x* energy reductions over dense and 2:4 sparse (Ampere) implementations, respectively, and incurs area and power overheads of 6% and 11.5%, respectively, over Ampere.

*This work was done as a graduate student at Purdue University

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614312

CCS CONCEPTS

• **Computer systems organization** → **Special purpose systems**; **Neural networks**.

KEYWORDS

Tensor cores, deep neural network (DNN) inference. one-sided sparsity, unstructured sparsity

ACM Reference Format:
Ashish Gondimalla, Mithuna Thottethodi, and T. N. Vijaykumar. 2023. Eureka: Efficient Tensor Cores for One-sided Unstructured Sparsity in DNN Inference. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614312

1 INTRODUCTION

Deep neural networks are deployed widely for many commercial applications including image classification, recognition, natural language processing, and recommendation systems. To meet the high compute demand of many of the models, commercial DNN accelerators (e.g., the GPU and TPU [12]) provide energy- and area-efficient hardware with vast compute and memory bandwidth resources. TPUs use large two-dimensional systolic arrays whereas GPUs use specialized compute units, called *tensor cores*, optimized for dense matrix multiplication [19].

Despite such specialized units, the demand for compute continues to outpace the hardware as the models grow in size and complexity. Sparsity, or zeros in the input matrices, can help meet this demand. Sparsity occurs in the filters due to pruning coupled with retraining for accuracy [9]. Sparsity in the filters [33–35] and activations [16] could be structured with hardware-friendly patterns [35] or unstructured without constraining the location of zeros [8, 9]. For instance, Ampere's structured 2:4 sparsity (i.e., two out of four filter values are non-zeros and fewer than two non-zeros treated as two non-zeros for regularity), leads to 2x better performance *and* nearly 2x better energy over dense (capturing sparsity incurs some modest energy and area overhead). By reducing work, sparsity improves both performance and energy. Due to its uniform 50% sparsity, 2:4 sparsity does not incur load imbalance or compute under-utilization so that the tensor cores maintain (input or output) operand stationarity [3], similar to dense operation, which is fundamental for avoiding high-overhead hardware. Consequently, the tensor cores add *only one* 4-1 multiplexor per multiply-accumulate unit (MAC) for 2:4 sparsity. This minimal extra hardware results in the tensor cores being highly efficient for both 2:4 sparse and dense operations.

Unstructured sparsity, on the other hand, is much higher (e.g., 83%) without loss of accuracy [8]. Capturing unstructured filter

sparsity alone (i.e., *one-sided*) would achieve 5-6x speedup and possibly 4-5x lower energy after accounting for sparsity overheads over dense (at modest area overhead). Increasing 2:4 sparsity from its 2x limit to this level would result in loss of accuracy [33]. Further, sparsity can also occur in the feature maps for some DNNs due to the presence of non-linear layers such as ReLU which zeroes out negative values. Such feature map sparsity, which is necessarily unstructured, can range up to 50%. Exploiting both filter and feature map (i.e., *two-sided*) sparsity can achieve further improvements. However, unstructured sparsity incurs load imbalance due to its non-uniformity.

To achieve load balance and high compute utilization, previous proposals targeting one- or two-sided unstructured sparsity [2, 6, 7, 10, 21, 22, 26, 29, 31, 32] incur considerable hardware overhead due to the added costs of buffering, routing, and sparse computation. As such, it is hard to scale up these proposals to the high compute throughputs of commercial accelerators like GPUs. In addition, these custom sparse accelerators cannot process dense matrices as efficiently as the GPU and TPU which is important for unpruned, dense models that continue to be used. Any unstructured sparse operation must satisfy the tensor cores' challenging constraint of minimal extra hardware (and hence energy and area). Unfortunately, previous tensor core-specific proposals violate this key constraint by adding crossbars, accumulation buffers, and scatter-gather logic, mostly due to sacrificing operand stationarity in favor of load balance [29]. Our goal is to exceed the $2x$ improvement limit of 2:4 sparsity and efficiently capture the opportunity of 5-6x work reduction in unstructured filter sparsity. In doing so, departing significantly from the current, highly-efficient tensor core organization or imposing considerable energy or area overhead on sparse or dense computation is not desirable.

Separately, larger tensor core arrays provide higher efficiency for larger models by exploiting higher reuse. However, broadcast of operands within the arrays is often pipelined for higher speed and lower energy than a brute-force, flat broadcast, resulting in a systolic-style computation (e.g., an 8x8 array may be implemented as four 4x4 arrays arranged in a systolic pipeline). Also, such systolic arrays can be scaled more easily to larger sizes. Any sparse operation, structured or unstructured, should not impede such systolic pipelines.

To achieve efficient unstructured filter (one-sided) sparsity, we propose *Eureka* starting with the basic observation that the 4-1 multiplexers for 2:4 sparsity suffice to capture unstructured sparsity while maintaining operand stationarity (i.e., no extra hardware). A further extension of our observation is that a simply larger multiplexer (e.g., 16-1) suffices to allow the offline *compaction* of a larger sparse filter matrix into a smaller matrix to improve compute utilization without increasing output buffering. While such compaction is not new [6, 14, 29, 32], the sufficiency of the 2:4 sparsity hardware has not been observed before. However, the resultant load imbalance among the non-zero and vacant matrix cells, which cannot be packed arbitrarily while maintaining operand stationarity, considerably hurts performance. Further, while 2:4 sparsity's uniformity allows for smooth systolic operation without any pipeline bubbles, unstructured sparsity's load imbalance-induced timing non-uniformity induces bubbles. To address these issues we make the following contributions:

- To achieve better load balance, our key insight is that a slight weakening of output stationarity achieves load balance most of the time while incurring only a modest hardware overhead. Accordingly, we propose *single-step uni-directional displacement*[1] *(SUDS)*, in which a filter element's multiplication can either occur in its original position or be displaced to a vacant MAC in the adjacent row below while the accumulation occurs in the original row to restore output stationarity. Because the filters do not change during inference, we compact the filters and apply SUDS offline before inference. SUDS adds only two 2-1 multiplexers and a carry-save adder with floating-point (FP) mantissa alignment per MAC.
- We design an optimal polynomial-time algorithm for SUDS work assignment. Our correctness proof also enables SUDS hardware overhead reduction.
- To achieve fewer bubbles in the tensor core's systolic pipeline due to the non-uniformity of unstructured sparsity, we propose offline *systolic scheduling* to group together the sparse filters with similar execution times (known statically based on the number of non-zeros). Systolic scheduling works even better when coupled with SUDS which shrinks the sparse computations' critical paths, lowering the unevenness in the pipeline's stages.

Eureka improves performance by $4.8x$ and $2.4x$, and reduces energy by $3.1x$ and $1.8x$ over dense and 2:4 sparse implementations, respectively. To achieve these improvements, at each MAC we (1) replace Ampere's 4-1 multiplexer with a 16-1 multiplexer and (2) add two 2-1 multiplexers and a carry-save adder with FP mantissa alignment for area and power overheads of 6% and 11.5%, respectively, over Ampere.

We considered extending our ideas to two-sided sparsity which, however, is significant only in convolutional neural networks (CNNs). Recurrent neural networks (RNNs), long short-term memory models (LSTMs), and recently, transformers, which are *more prevalent* than CNNs [11], have only a few to no ReLU layers. As such, we do not pursue two-sided sparse tensor cores.

2 BACKGROUND

2.1 Deep neural networks

A deep neural network has many layers, where each layer's output is generated using operations such as convolutions, matrix multiplication, and non-linear operations (e.g., ReLU). Convolutions and matrix multiplications are compute-intensive and consume time and energy. Convolutions can be transformed into matrix multiplication using implicit GEMM kernels without IM2Col memory bloat [20].

Matrix multiplication can be implemented using two fundamentally different methods. *Inner product* method computes the *dot product* of every pair of row and column vectors, from the first and second input matrices, respectively. *Outer product* computes a partial output matrix as a *cross product* of only the matching i^{th} column and the i^{th} row vectors, from the first and second matrices, respectively. Accumulating the partial matrices of different

[1] Like the water being displaced up in Archimedes's "Eureka" moment.

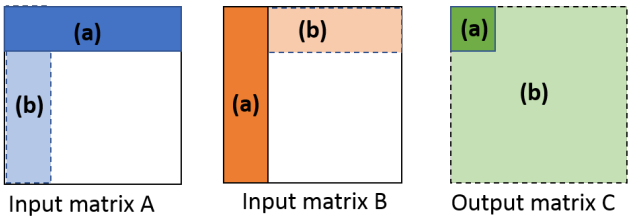

Figure 1: (a) Inner product; (b) Outer product methods

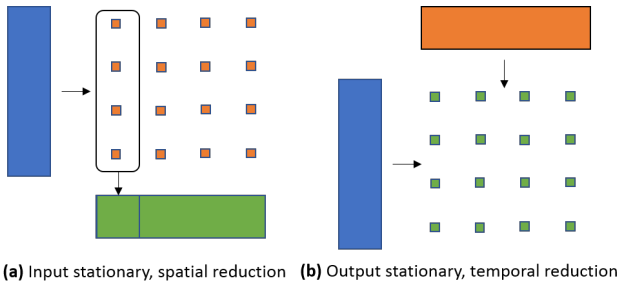

Figure 2: (a) Input- (b) Output-stationary dataflows

column-row pairs gives the final output matrix. Figure 1 shows the methods.

A key aspect is effectively utilizing the hardware multiply-accumulate units (MACs) with low buffering and data movement, as determined by the hardware dataflow. A well-known approach is to hold stationary one of the matrices' elements in the MACs and minimally move the remaining matrices [3]. In an *input-stationary* dataflow, each MAC holds an input element (or a group of input elements) of one matrix (Figure 2(a)). The other input matrix is broadcast to the MAC to generate the partial output element. The partial output elements are transferred from one MAC to the next (spatially) for accumulation to compute the final output element. Alternatively, the four marked MACs are interconnected using a reduction tree to enable spatial reduction. Thus, for the input-stationary dataflow, an inner product method allows for minimal data transfers. In contrast, in an *output-stationary* dataflow (Figure 2(b)), each output element (or a group of output elements) is generated in a MAC in place by broadcasting both of the corresponding input vectors. Each partial output element is accumulated in place over time to produce the final output element. Thus, the output-stationary dataflow supports the outer product method with minimal data transfers. Although there is a reduction tree in the input-stationary approach and two-directional broadcast in the output-stationary design, the approaches are similar in terms of overall cost.

Matrix multiplication is often tiled and parallelized on many MACs. The tiles can be of various shapes (e.g., row, square, or column). In addition, a tile of one input matrix can be multiplied with multiple tiles of the other input enabling enormous reuse. The tiles can be nested with either method at each nesting level, allowing for many hybrid methods.

2.2 Sparsity in DNNs

Exploiting two-sided sparsity gives high performance benefits but adds expensive hardware to tackle irregularity via buffering, index-matching or routing [6, 7, 22, 29]. Two-sided sparse architectures like SCNN [22], SparTen [6], and DSTC [29], which closely resembles SCNN except for bit masks like SparTen to reduce SCNN's address calculation overhead, achieve high performance while incurring high energy overhead. For example, SCNN employs expensive crossbars for partial sum accumulation while SparTen adds prefix-sum and priority encoder logic, and buffers. Although SparTen's overheads drop significantly for smaller tensor cores (size of 4 or 8), performance also falls without a large look-ahead window (e.g., at least 32). Additionally, input feature map sparsity is limited to layers containing ReLU, which is common in most CNNs. However, as discussed in Section 1, other *more prevalent* DNNs, such as LSTMs and recently, transformers [11], have only a few to no ReLU layers limiting two-sided architectures' opportunity. On the other hand, filter sparsity is present in almost all DNNs. Accordingly, Ampere's tensor core exploits one-sided filter sparsity [2, 35] but is limited due to its structured 2:4 sparsity. Thus, we need a low-cost approach to capture unstructured one-sided sparsity in filters.

2.3 Tensor Core

Nvidia's tensor core architecture details are not public. As such, we first describe a sensible, well-motivated tensor core design that efficiently performs dense as well as 2:4 sparse matrix multiplications. Because the design matches observed latencies and operates efficiently, any other equivalent design would not change our techniques or results.

Each tensor core in Volta and Turing has 64 MACs [19]. Custom instructions, such as HMMA.1688 and HMMA.844 [30], take input matrices of sizes 16x8, 8x8 and 8x4, 4x4 respectively. Without loss of generality, we consider the building block of two 4x4 matrices multiplied on a 4x4 MAC array taking four cycles (and a 16x8 or 8x8 matrix multiplication taking eight cycles). This computation can be realized with many architectures, each with its own well-known trade-offs. For example, a systolic design with a 4x4 array has low-cost, near-neighbor communication but high latency. In contrast, a full broadcast-tree based 4x4 array would have lower latency with higher communication costs (broadcast costs grow quadratically with size). Further, as discussed in Section 2.1, efficient execution can be realized with the appropriate method and hardware support. For example, inner product is efficient with reduction trees whereas outer product is efficient with two-directional broadcast support.

The 16-MAC array can be extended to 64 MACs, organized as a flat 8x8 array or as four 4x4 sub-arrays connected systolically [15] to operate on 8x8 matrices or 16x8 matrices partitioned into 2 8x8 sub-matrices. Figure 3 shows a 2D systolic compute array.

Given these possibilities, we assume a simple and efficient canonical building block of 4x4 array with a flat broadcast within this small array and a larger array composed of these building blocks (e.g., four 4x4 sub-arrays for an 8x8 array) with a systolic broadcast from one sub-array to the next.

2.3.1 Tensor core for structured sparsity. We extend the above dense matrix tensor core design to Ampere's 2:4 structured sparse operation. Sparse operation in tensor cores is easier with outer product

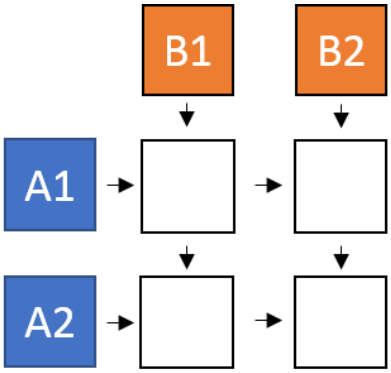

Figure 3: Systolic array producing 4 matrices in 4 cycles

(a)

A00			A03
	A11	A12	
		A22	A23
A30	A31		

(b)

A00	A03
A11	A12
A22	A23
A30	A31

Figure 4: A 2:4 sparse matrix: (a) original; (b) left-aligned

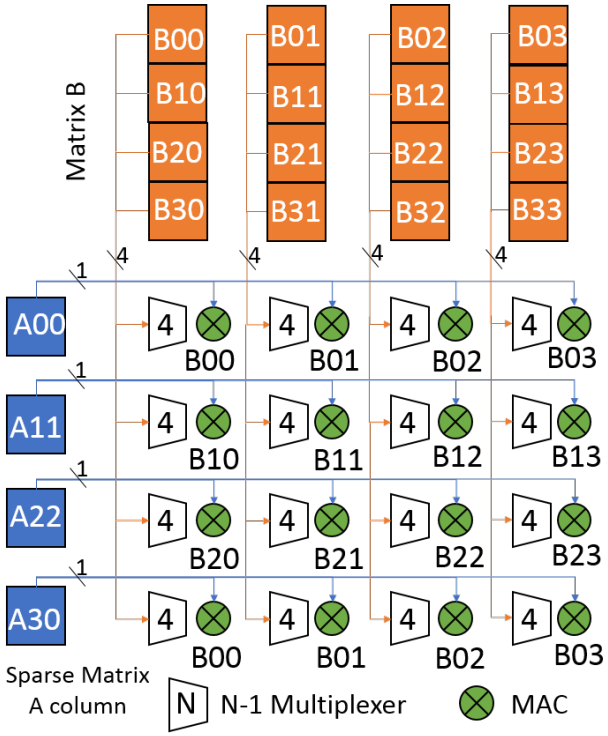

Figure 5: Multiplexers for structured sparsity

and output stationary approach, as also observed by DSTC [29]. Inner product with one-sided sparse input requires fetching the matching element in the second input and the stationary input faces uncertainty over the non-zero positions [23]. Inner product with stationary output would also face similar uncertainty [6]. In contrast, outer product with stationary output requires only modest hardware addition, especially for one-sided 2:4 sparsity where the four columns of a 4x4 matrix are left-aligned into two columns (i.e., exactly two non-zero values in every row), as shown in Figure 4.

Recall from Section 2.1 that in outer product with stationary outputs for *dense* matrices (i.e., cross product of column i in the first input matrix and row i in the second input matrix), the MAC for output element (n,m) multiplies the n^{th} row in column i and m^{th} column in row i. Thus, in the 4x4 MAC array in Figure 2(b), each column-row pair computes the full 4x4 cross product by broadcasting the column i left to right and the row i top to bottom, and accumulates the output in place each cycle. In 2:4 sparsity, however, each sparse row contains two non-zero elements where the first element can be from any of the first three columns within the same row of the original matrix and the second element from the last three columns (Figure 4(b)). Accordingly, the (n,m)th MAC's first input is the n^{th} row of one of the original first three columns (e.g., the first column in Figure 4(b) holds elements from one of the first three columns in the same row in Figure 4(a)). Hence, the second input has a 3-1 choice among the *first* three rows of the m^{th} column. However, the same MAC is used next cycle for the next column-row pair where the second input has a 3-1 choice among the *last* three rows of the m^{th} column. Hence, we need a 4-1 multiplexer to select the appropriate row for the second input. Figure 5 shows the operation for the first column of Figure 4(b). Thus, with only one extra 4-1 multiplexer per MAC, outer product produces the output for 2:4 sparsity in exactly two cycles without any uncertainty (dense matrices take 4 cycles). Because a filter column's elements are broadcast to their respective rows in the MAC array as in the dense case (Figure 2(b)), the MACs in a row share the multiplexer control.

2:4 sparsity adds 2 bits of metadata per value to identify the value's original column. This increase is more than offset by the 50% reduction in the matrix size. For such one-sided sparsity, both the matrix data size and the number of MAC operations reduce by the same factor (50%), keeping the memory bandwidth demand similar to that of dense operation. The metadata incurs a small increase in the bandwidth demand. Fortunately, the abundant reuse exploited by the output stationary dataflow ensures that the sparse tensor core's net bandwidth demand remains reasonable.

3 *EUREKA*

Recall from Section 1 our basic observation that the structured 2:4 sparse tensor core can also be used for unstructured sparsity. The flexibility provided by the 4-1 multiplexers is enough for unstructured sparsity where the second input for the (n,m)th MAC has a 4-1 choice among all four rows of the second input matrix's n^{th} column. Unfortunately, the unpredictability of the non-zero counts, as opposed to 2:4 sparsity's exactly two non-zeros in every four dense values, causes considerable load imbalance and compute underutilization. For instance, with 87.5% observed at moderate pruning in ResNets, each 4x4 matrix has around two non-zero elements on

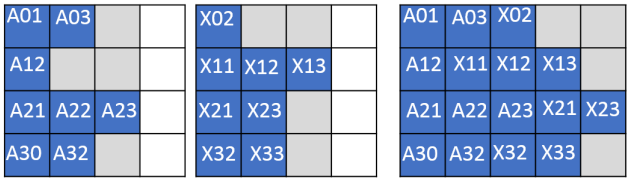

Figure 6: Compacting left-aligned unstructured-sparse matrices

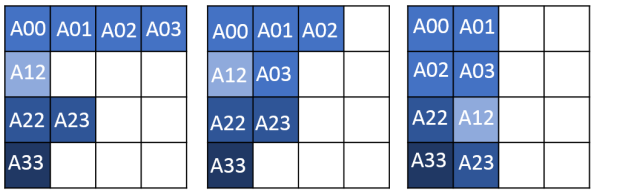

Figure 7: Displacement: Greedy versus Optimal

average. If the two non-zeros are in the same column (best case), the multiplication can finish in just one cycle but achieve only 50% utilization. In the worst case, where the non-zeros are in the same row, utilization drops to just 25%.

Fortunately, as filters are available offline we can pre-process them to achieve better compute utilization. We start off with well-known, offline matrix compaction where multiple sparse input matrices are compacted into a single matrix [6, 14, 29, 32]. In Figure 6, two left-aligned, unstructured-sparse matrices are compacted along the rows, which improves utilization. The residual under-utilization correlates with the number of vacant cells in the shaded area (4+4 = 8 in Figure 6(a) versus 4 in Figure 6(b)).

We extend our basic observation with the point that replacing Ampere's 4-1 multiplexer with a larger multiplexer suffices for this compaction without increasing output buffering (e.g., 8-1 or 16-1 multiplexer for compacting 4x8 or 4x16 matrices into a 4x4 matrix). To see this extension, observe that each column of the compacted matrix is broadcast to the compute array, similar to Figure 5. However, given that 8 (or 16) columns are compacted instead of 4 columns as done in 2:4 sparsity, we need 8-1 (or 16-1) multiplexer instead of 2:4 sparsity's 4-1 multiplexer.

Being offline, the compaction does not add any run-time overhead. However, the metadata to identify a non-zero value's original column increases (e.g., from 2 bits to 4). Further, increasing the *compaction factor* (ratio of the larger to smaller matrix sizes) improves the sparse tensor core's utilization and reuse but also increases buffering. Fortunately, these costs are modest for small compaction factors (e.g., 2-4).

Despite matrix compaction, load imbalance exists due to irregularity of unstructured sparsity. The non-zero and vacant cells of the compacted matrix cannot be packed arbitrarily while maintaining output stationarity, leading to MAC idling. To address this issue, we propose: (1) *single-step uni-directional displacement (SUDS)* for moving the multiplication work to a vacant MAC in the adjacent row below, (2) an optimal work assignment algorithm for SUDS, and (3) systolic scheduling to reduce bubbles due to uneven sparsity in the scaled-up systolic compute arrays. The optimality proof also helps in reducing the SUDS hardware overhead. The following subsections describe each contribution in detail.

3.1 Single-step uni-directional displacement

While matrix compaction improves utilization by compacting the filters along the rows as discussed above, there is no compaction along the columns. Thus, compaction is a hit or miss approach: a compacted matrix where one row is much longer than the others would incur severe under-utilization. While simply distributing the values among the rows would achieve better utilization, such arbitrary distribution would disrupt output stationarity leading to high hardware, area, and energy overheads. Our key insight is that a slight relaxation of output stationarity achieves load balance in most cases while incurring low overheads. Accordingly, we propose single-step uni-directional displacement (SUDS), done offline in software, in which a filter element's multiplication can either occur in its original position or be displaced to a vacant MAC in the adjacent row below while the accumulation occurs in the original row to restore output stationarity.

Figure 7 shows an example of SUDS. The first row of the sparse matrix A initially (Figure 7(a)) has four values while the other rows have two or fewer. In Figure 7(b), a value from the first row is displaced to an adjacent MAC below, reducing the cycle count from four to three. The partial product generated in the bottom MAC is accumulated at the original MAC to restore output stationarity.

A key correctness point is that in the outer product method without any displacement, the partial products for all the elements in a row i of a filter matrix are accumulated at the same row i of the output matrix. Consequently, a displaced value's partial products can be accumulated at the partial products of the row above, irrespective of the column to which the value is displaced. Though restricted to displacing work only to the row below the original position, SUDS is powerful because multiple elements can be displaced achieving good load balance. Figure 7(c) shows a perfectly load-balanced result.

Though the multiplication for a displaced value occurs at the MAC below, the accumulation of the partial product occurs at the displaced value's original position to restore output stationarity. Accordingly, we add extra wires for the displaced partial product to return to the original MAC above (Figure 8). In the figure, there are four possibilities for *two vertically-adjacent* MACs: (1) neither of the values they are multiplying was displaced (e.g., *A22* and *A33* in Figure 7(c)), (2) the top value was not displaced but the bottom value was (from above) (e.g., *A00* and *A02*, respectively), (3) the top value was displaced but the bottom value was not (e.g., *A02* and *A22*, respectively), and (4) both values were displaced (e.g., *A03* and *A12*). The first case requires no change. In the second case, the two partial products (from the top and bottom MACs) belong to the same output element and are accumulated at the top MAC, performing a three-input addition. In the third case, the top MAC's product goes to the MAC above leaving the top MAC's adder unused (the bottom value stays at the bottom MAC). In the final case, the top

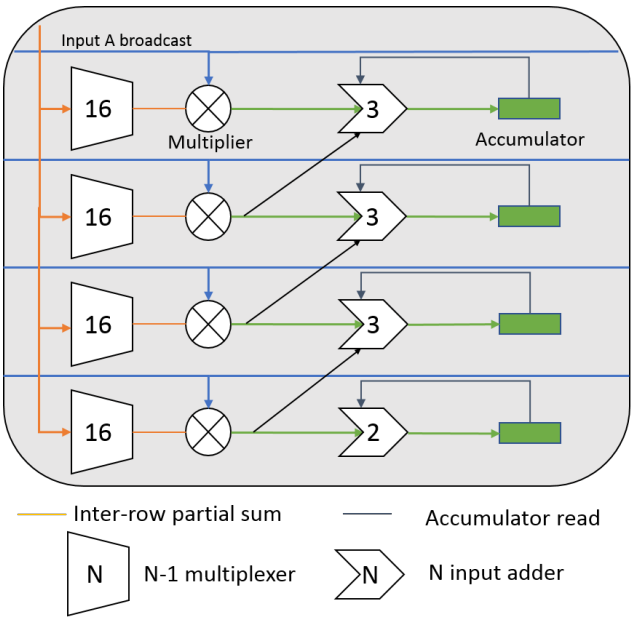

Figure 8: SUDS hardware (one MAC column shown)

MAC's product goes up one hop and the bottom MAC's product is accumulated at the top MAC. Hence, we need a three-input adder in each MAC.

The three-input adder's inputs are the local product, local accumulator, and the product from below (Figure 8). In the first case above, the third input is set to zero. In the second case, all three inputs are valid. In the third case, no addition is done. And in the final case, the first input is set to zero. Therefore, the first and third inputs need a 2-1 multiplexer each. As in the case of 2:4 sparsity (Figure 5), the relevant multiplexers of a row share their control. The three-input addition can be implemented as a carry-save adder to reduce the three values to two (the sum and carry) followed by a full adder. For FP16 values, the three exponents are compared against each other before the three mantissas are aligned and added together.

To indicate to the hardware whether a value is displaced requires only one bit per value, in addition to *Eureka*'s 4-bit metadata for compaction (Section 3).

SUDS can cut the critical path, the longest row, by 50% even for the worst case. For example, in a worst-case sparse matrix that has a single row with four values, utilization improves by $2x$ by displacing two values from the original row to the row below. In general, we need to find the assignment of the values across the rows that achieves the best utilization. As mentioned before, because the filters do not change for inference, this work assignment is done offline. We describe our optimal work assignment algorithm in the next section.

3.2 Optimal SUDS work assignment

A naive, greedy algorithm may not be able to generate the optimal work assignment. For example, a greedy algorithm that simply finds an empty anti-diagonal slot in the compacted matrix can fail to achieve the minimum critical path. As shown in Figure 7(b), while considering the first row the algorithm checks only the second row and finds one empty slot. While considering the second row checks only the third row and finds no slots. Thus, the algorithm produces a three-column matrix after displacement. However, the optimum is a two-column matrix, as shown in Figure 7(c). Thus, we need an algorithm to achieve the optimal work assignment.

Algorithm 1 Algorithm for the decision problem

Input: Sparse Matrix M, number K
Output: Sparse Matrix O with max row length ≤ K

```
slack_rows ← {}
for each row in M do
    if (length(M[row]) ≤ K then
        slack_rows.append(row)
    end if
end for
for each row in slack_rows do
    O ← M
    i ← p − 1
    while i ≥ 0 do
        row_above ← row − 1 (mod p)
        C ← length(O[row])
        C_above ← length(O[row_above])
        slack ← K − C
        n_disp ← min(C_above, slack)
        //Displace n_disp elements from row_above to row
        O[row] ← displace(O[row_above], n_disp)
        row ← row_above
        if length(O[row]) > K then
            Break to next slack row
        end if
        i ← i − 1
    end while
    return O // found an optimal solution
    // no need to try more slack rows
end for
return No solution
```

A key challenge in finding the optimum is that while a row may have some room, the row above may need to displace more than the room available so that the row with the room itself may have to displace to the row below to make more room for the row above. And so on, across all the rows. Thus, it is not easy to decide how much to displace to the next row because the next row's length may change based on how much the next row itself displaces later. Therefore, we employ a two-step process where we first find feasible work assignments that can fit all the rows within a given length and then search through the feasible solutions to find the optimum.

For the first step, we define the following decision problem.

Definition 1. *Given a sparse matrix M of dimension pxq and a number K, is there an assignment such that each value of row i either stays in the i^{th} row or is displaced to the $i+1^{th}$ (mod p) row (i.e., wrap around) and the final matrix's longest row (i.e., the critical path) is less than K.*

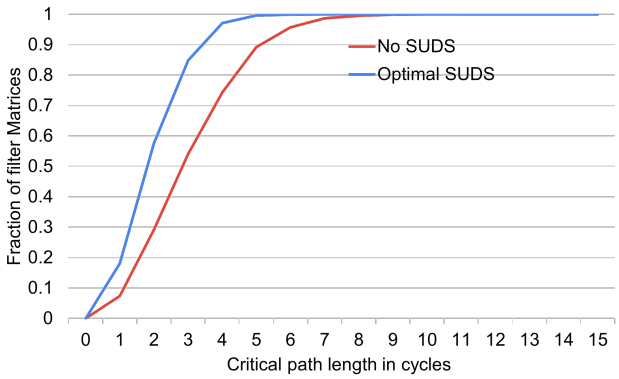

Figure 9: Critical path distribution of four filter sub-matrix groups in *ResNet50*

Additionally, we define the following two terms:
Slack rows: A row in M with a length C such that $K \geq C$.
Base rows: A row that does not displace any elements to the adjacent row below.

With these definitions, we see that a slack rows can accept displaced values from the previous row. However, a slack row may itself displace some of its values to the adjacent row to satisfy the decision problem. We also define that a satisfying assignment is *minimal* if there are no redundant displacements. An example redundant displacement is one that displaces exactly one value in each row without changing the longest row length.

If the decision problem has a minimal solution, then there is at least one slack row that is also a base row (i.e., no value displaced to the adjacent row). We prove this statement by contradiction. Let us assume that a minimal solution exists with no base rows so that each row has displaced at least one value. However, this solution is not minimal as there is a redundant movement. Hence a minimal solution must contain at least one base row. In addition, the base row is always a slack row; otherwise, the row length would exceed K so that the solution would not satisfy the decision problem.

Algorithm 1 finds a minimal solution for the decision problem 1 if the solution exists. Because we do not know which slack row is a base row, the algorithm tries each slack row as a base row. The algorithm starts at a slack row and at each row greedily fills up the available slack by displacing from the adjacent row above before moving to the row above. If this process results in a negative slack for any row (i.e., row length > K), then the algorithm moves on to try the next slack row. Only a truly base row would lead to a solution. The time complexity of this offline algorithm is $O(p^2)$.

Given this algorithm to find a minimal solution for the decision problem, we extend the algorithm to determine the optimal longest row length K_{opt}. To that end, the extension tries every K between the lower bound of $ceil(count(M)/p)$ where count is the number of values in M (the longest row cannot be shorter than this bound) and the upper bound of q. For larger matrices, the trials can be binary-searched between these bounds leading to a complexity of $O(p^2 \log q)$. Using the row lengths of the minimal solution with K_{opt}, we displace the values of M to produce the displaced filter matrix O.

The SUDS assignment compresses the sparse sub-matrices leading to two key changes in the sub-matrix critical path distribution: (1) more sub-matrices have shorter critical paths and (2) the critical path distribution has shorter range and lower variation. Figure 9 shows the distribution before and after the optimal assignment while scheduling four sparse filter sub-matrices for an intermediate layer in *ResNet50*.

Finally, although the cases for small $4x4$, $4x8$ matrices can be enumerated exhaustively, especially if offline, the above algorithm is scalable to larger sizes due to it polynomial time complexity. A key observation from our proof is that the number of displacements needed is just $p-1$ for a pxq matrix because the base row does not displace. Consequently, the hardware can avoid SUDS support in one of the MACs. Accordingly, we offline rotate the matrix so that the base row is placed always on the last $((p-1)^{th})$ MAC row avoiding expensive wraparound wires from the last MAC row to the first (Figure 8). Also, the last row of MACs can use two-input, instead of three-input, adders. For this rotation, we add a two-bit field, indicating the rotation amount, to each filter matrix for adjusting the software index while loading the filter matrix into the GPU and storing the output matrix into memory. The rest of the computation stays oblivious of this rotation.

3.3 Systolic scheduling

Recall from Section 2.3 that the tensor core is organized as a large systolic array of smaller MAC sub-arrays (Figure 3). If the sub-arrays take different number of cycles to complete due to uneven sparsity, there would be bubbles in the systolic array. While SUDS optimizes the smaller sub-arrays' utilization, the bubbles considerably degrade utilization as each bubble keeps an entire MAC sub-array idle. In contrast to *Eureka*'s unstructured sparsity, 2:4 sparsity incurs no bubbles because the sub-arrays are guaranteed to have the same latency. To reduce the number of bubbles, we propose offline systolic scheduling to feed sparse sub-matrices with the same or that add up to the same critical path length along each row of the $2D$ systolic array. The critical paths are known from our optimal assignment.

Recall systolic movement from Figure 3. In Figure 10(a), the sub-matrices $A1$ in input batch 1 and $A3$ in input batch 2 take two cycles each in *each* of their systolic stages in the top systolic row. The sub-matrices $A2$ and $A4$ take one cycle each in each of their stages in the bottom systolic row. Every stage takes the *same* number of cycles, which may be one or more and may vary across input matrices. Consequently, $A2$ waits in place in cycle 4, incurring a bubble as $A1$ and $A3$ continue in their respective stages for their second cycle. Similarly, $A4$ waits in place in cycle 6 as $A3$ continues in its stage for its second cycle. With systolic scheduling (Figure 10(b)), we can feed either two sub-matrices each taking one cycle adding up to two cycles ($A2$ and $A4$) or a sub-matrix which takes two cycles in place of $A2$ (not shown), matching $A3$'s latency in each systolic stage. $A2$ cannot move in cycle 4 as $A3$ continues in its stage for its second cycle. Instead, $A4$, scheduled with $A2$ to match $A3$'s two cycles, replaces $A2$ in cycle 4.

Boosted by the SUDS work assignment providing many sub-matrices with short critical paths (Section 3.2), the systolic scheduling achieves high utilization.

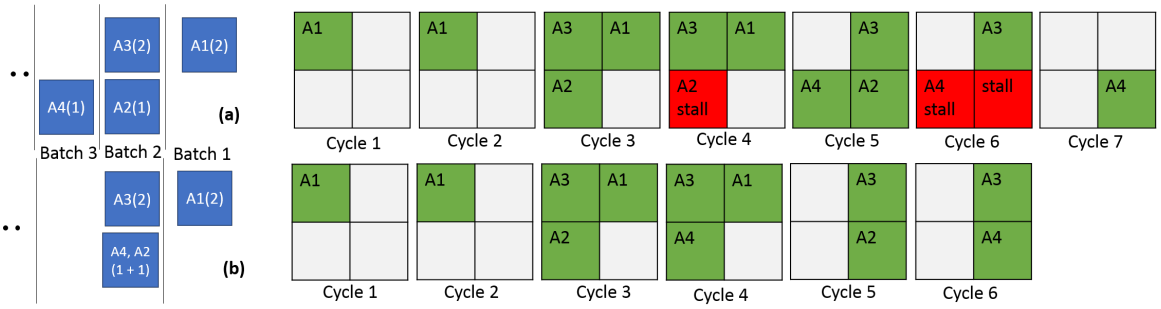

Figure 10: Example for systolic scheduling: (a) original (b) scheduled

Finally, our systolic scheduling is offline. To avoid high bandwidth pressure for loading the filter sub-matrices from the D cache into registers, we limit scheduling only a small number of filter sub-matrices in one row of the systolic array (e.g., 2). The small number suffices due to being coupled with the optimal assignment. Systolic scheduling does not incur much addressing complexity because the scheduler packs entire sparse filter sub-matrices, not individual elements. The input filter sub-matrix's filter identifier (in a GPU register) is used to, store the corresponding output sub-matrix (via GPU instructions). The scheduling requires some minor instruction changes. A register specified in the multiply instruction indicates the cycle in which an output should be extracted based on the critical path known to the optimal assignment.

3.4 Other issues

Two other issues are: (1) prevalence of sparsity and unstructured sparsity's accuracy, and (2) scalability of *Eureka*'s multiple tensor cores.

For the first issue, all sparse accelerators, including Ampere, need sparsity which exists for most models. Without sparsity (rare), even Ampere will not work. In fact, unstructured sparsity is more robust for accuracy than structured sparsity. Still, Eureka incurs low energy overhead over Ampere for dense models.

For the second issues, one large array (e.g., TPU) and multiple tensor cores have similar number of compute units (MACs) though larger arrays have more data reuse (key difference). With a shared L2, tensor cores achieve similar on-chip reuse as the TPU, making memory bandwidth sharing moot (if anything, unstructured sparsity requires lower bandwidth). That said, multiple tensor cores is GPU's choice (which seems to work well), not *Eureka*'s.

4 METHODOLOGY

To evaluate *Eureka*, we build a cycle-level simulator modeling various sparse architectures and estimate the designs' area and power from their Verilog implementations.

Benchmarks: We obtain the sparse models from *SparseZoo* [18] with conservative and moderate degree of pruning, ordered by increasing moderate-pruning sparsity in Table 1. We use batches of 32 224*x*224*x*3 inputs from ImageNet [5] except for *Inceptionv3* which has an input size of 299*x*299*x*3 [28]. For BERT, we use the SQUAD model [24]. While transformer-based models do have significant two-sided sparsity and thus favor *Eureka*, SparseZoo has only one transformer-based model – BERT. Generating our own sparse models is both time-consuming and expensive, and less credible than public models.

Table 1: Benchmarks

Benchmark	#layers	Unst. fil. dens. & acc. conservat.		Unst. fil. dens. & acc. moderate		S2TA dens. act.	S2TA dens. fil.
		%	%	%	%	%	%
MobileNetv1	27	27	70.9	22	70.1	39	38
Inception-v3	94	18	77.4	16	76.6	-	-
ResNet50	53	20	76.1	13	75.3	44	38
BERT-squad	72	20	88.6	10	88.07	50	50

Simulated systems: We simulate a dense tensor core architecture (*Dense*), 2:4 structured-sparse Ampere (which covers STC as well), one-sided, unstructured-sparse Cnvlutin, two-sided unstructured-sparse DSTC [29] and SparTen [6], two-sided structured-sparse S2TA [16], and *Eureka* with two offline matrix compaction factors of 2 and 4 (indicated by *P*). We simulate 432 tensor cores (similar to Ampere) each comprising a systolic array of four 4x4 sub-arrays of MACs, and cache and memory (DDR5). Our compute-bound workloads' maximum demand is 251 GB/s memory bandwidth (compared to Ampere's 1.5 TB/s available bandwidth).

We carefully model DSTC's outer product operation, which closely resembles that of SCNN, with 1x8 and 1x16 vectors, crossbar for scatter-gather to and from the accumulation buffers, and buffering. We also faithfully model SparTen's inner product operation with prefix sum and priority encoder logic for finding matching non-zero positions in the inputs, and significant buffering and hardware greedy balancing (GB-H) to ensure good compute utilization (two double-buffered input *chunks* of 32 FP16 values each with accompanying bitmasks and two double-buffered FP!6 output values, totaling 280 bytes per MAC). S2TA seems to clock-gate the MAC when there is no matching non-zero filter value for a non-zero activation value and performs the multiplication otherwise, similar to EIE [7]. While S2TA's text is unclear about the no-match condition, Figure 6d's clock-gated MAC and Figure 6e's bitmask before the multiplexer suggest such idling. Thus, contrary to its name, S2TA exploits only *one-sided* structured (activation) sparsity for performance and two-sided structured sparsity for energy. By forgoing two-sided sparsity's performance, S2TA achieves energy efficiency,

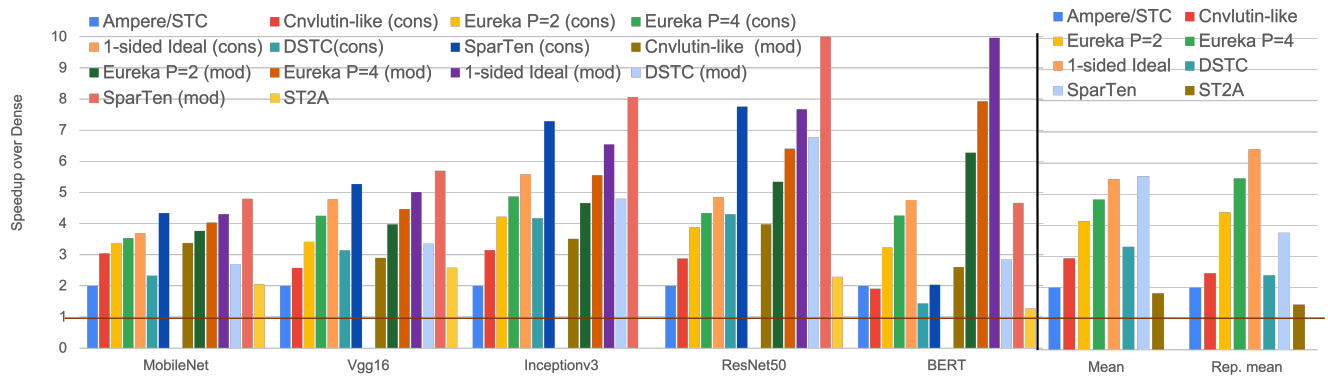

Figure 11: Performance

as confirmed by S2TA's 2.1x speedup and 2x energy over *SA-ZVCG* which power-gates the MAC upon a zero and is essentially *dense* for performance and *one-sided*, unstructured sparse for energy. S2TA compares against SparTen and SCNN only for energy but not performance! While S2TA's activation sparsity is different for each layer, we use the mean sparsity for each benchmark (Table 1), the only data listed in the paper. S2TA's filter sparsity is 2:4. While Ampere/STC, DSTC, S2TA, and *Eureka* are all tensor core-based, we scale non-tensor-core-based Convlutin and SparTen to match *Eureka* (same number of MACs, and cache and memory, but more hardware than *Eureka*). In fact, SparTen is *more* energy-efficient at smaller scales than the original, and performs *better* than *Eureka* for two-sided benchmarks, as shown in Figure 11.

Area/Power estimates from ASIC synthesis: To estimate area and power, we synthesize individual components such as FP16 MACs, multiplexers and crossbars implemented in Verilog using the Synopsys Design Compiler, FreePDK-15 nm [17] along with NanGate-15 nm open cell library. For power, the default estimates from FreePDK-15 nm deviate from expected values by orders of magnitude. Instead, we synthesize our RTL implementation using FreePDK-45 nm and then scale the power estimates from 45 nm to 15 nm [27].

5 RESULTS

We present the following results:(1) performance comparison of *Eureka* against 2:4 sparsity and Cnvlutin [2], both of which are one-sided sparsity schemes, (2) performance isolation of *Eureka*'s techniques, (3) energy comparison of the schemes, (4) performance sensitivity to the MAC array size, and (5) area, power, and delay estimates from an ASIC synthesis of an RTL implementation of *Eureka*.

5.1 Performance

We compare *Eureka* to dense tensor core (*Dense*), 2:4 structured sparse architecture (*Ampere/STC*), *Cnvlutin-like*, *DSTC*, *SparTen* and *S2TA*. While Cnvlutin is designed specifically for convolutions with one-sided unstructured, sparse feature maps, *Cnvlutin-like* is an adaptation with sparse filters. *Cnvlutin-like* inherits not only Cnvlutin's equivalent of compaction factor of 4 but also its lack of targeted load balancing. For *Eureka*, we vary the compaction factor P as 2 and 4 (a $4x4P$ matrix is compacted into a small matrix). We also show an ideal version of *Eureka* where the speedup is limited only by one-sided sparsity (*1-sided Ideal*). The X-axis shows conservatively-pruned (*cons*) and moderately-pruned (*mod*) models. S2TA uses only one version of structured sparsity (not *cons* or *mod*); S2TA is shown in the *mod* clusters. Further, S2TA does not run *Inceptionv3* whose structured activation sparsity is hence unavailable. We show the mean to the right (sans *Inceptionv3* for S2TA). However, because modern DNN workloads are dominated by DNNs other than CNNs such as transformers [11], we show a *representative mean (rep mean)* with BERT's weight at 75% and that of the rest (CNNs) at 25%. This 75-25 split is shown by TPUv4i [11].

Figure 11 shows performance normalized to that of *Dense* on the Y-axis and the X-axis shows the benchmarks in the order of increasing moderately-pruned sparsities (Table 1) to capture the performance trends. The bars for conservative and moderate pruning for each benchmark are separated for clarity. Across the benchmarks, *Ampere/STC* achieves $2x$ speedup over *Dense* as expected from 2:4 sparsity. Exploiting unstructured sparsity provides higher opportunity for both *Cnvlutin-like* and *Eureka*. However, *Eureka* performs better than *Cnvlutin-like* which lacks load balancing. Increasing *Eureka*'s compaction factor from 2 to 4 improves the compute utilization, pushing the speedups closer to *1-sided Ideal*. In addition to compaction, *Eureka*'s load balancing (SUDS and systolic scheduling) efficiently capture most of the opportunity via only local displacement.

Despite being two-sided, *DSTC* does not achieve higher speedups than *Eureka* for the CNNs, which have reasonable two-sided sparsity, because of (1) being power- and area-limited to four 4x4 crossbars which can route upto only 16 partial products out of a maximum of 64 in DSTC's 8x8 array (16 accumulation buffer values per cycle, as stated in [29]), and (2) the lack of load balancing. DSTC discusses these limitations, which result in around 5x lower speedups than the opportunity exposed by two-sided sparsity (e.g., DSTC shows around 30-200x opportunity for BERT's individual layers but only 7-9x speedups). In our results, *DSTC* captures similar fractions of the opportunity. Unlike CNNs, BERT does not include ReLU resulting in nearly-dense activations and lower opportunity

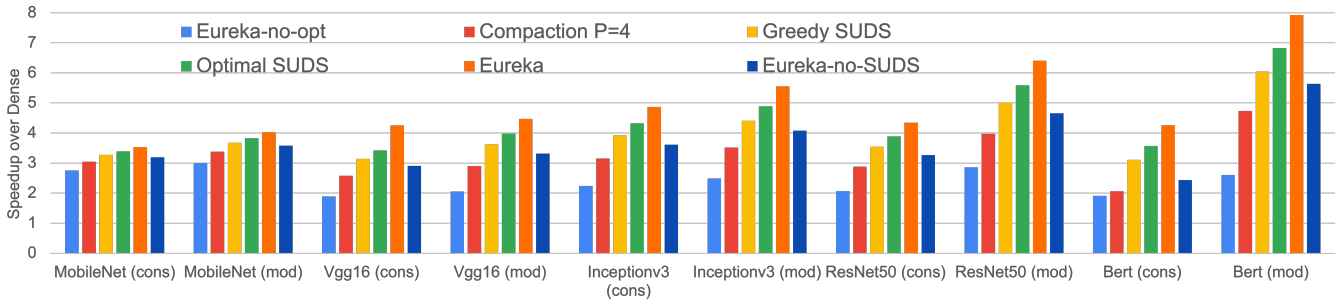

Figure 12: Isolating the impact of *Eureka*'s techniques

for *DSTC*. Further, *DSTC* incurs heavy load imbalance in *BERT* because of large chunks of high sparsity in the filters. In contrast, the two-sided *SparTen* achieves higher speedups than *Eureka* for the CNNs though at the cost of energy, as seen next. However, *SparTen* has worse speedups than *Eureka* for BERT because BERT's coarse filter sparsity pattern implies that large parts of the nearly-dense activations are fetched and skipped over wasting time and energy, despite SparTen's GB-H. In contrast, *Eureka*'s SUDS fills the sparse chunks with non-zero weights from elsewhere to achieve good utilization. Being essentially one-sided for performance with around 50% density (Section 4), S2TA achieves speedups similar to those of *Ampere/STC*, except for *BERT* whose activations are nearly dense. Because our workloads are highly compute-intensive, the memory system accounts for only 9-13% of the execution time in all the architectures (not shown to avoid clutter).

On average, *Eureka* $P = 4$ is $2.4x$ faster than A100 (mean). The more power-efficient variant, *Eureka* $P = 2$, performs $2.0x$ better than A100. Due to its limitations, *DSTC*'s mean is only slightly better than that of *Cnvlutin-like*. The two-sided *SparTen* achieves higher mean speedups than *Eureka* of $5.5x$ over A100. Being one-sided for performance, *S2TA*'s mean is similar to A100's. Because *rep mean* weighs *BERT* more than the CNNs, the *rep mean* speedups for all the schemes are close to those of *BERT* where *Eureka* significantly outperforms *SparTen*.

Our key insight that a slight relaxation of output stationarity suffices to achieve load balance most of the time (Section 1) is borne out by the modest gap between *Ideal* and *Eureka*. This gap exists because SUDS's local displacement cannot achieve perfect utilization for some inputs in contrast to unconstrained displacement. However, capturing this gap requires foregoing output stationarity with the accompanying costs of buffering and routing, which may not be justified.

5.2 Isolation of *Eureka*'s techniques

To isolate the impact of compaction, SUDS, and systolic scheduling, we add each of these techniques in a progressive manner. We start with unoptimized *Eureka* (*Eureka-unopt*) and progressively add (1) compaction with a factor of 4, (2) SUDS with a greedy work assignment (*Greedy SUDS*), (3) SUDS with optimal work assignment (*Optimal SUDS*), and (4) systolic scheduling (*Eureka*). Lastly, we also show full *Eureka* including systolic scheduling but without SUDS (*Eureka-no-SUDS*). In Figure 12, the Y-axis shows speedups normalized to *Dense*. From the left, *Eureka-unopt* does not exploit much of the opportunity because of uneven sparsity, incurring as many cycles as the longest row length. Next, compaction improves utilization by packing more non-zeros into the columns. The greedy and optimal versions of SUDS shorten the critical path to boost performance. Adding systolic scheduling (i.e., full *Eureka*) further improves performance by ensuring fewer pipeline bubbles. *Eureka-no-SUDS* is better than *Compaction P=4* showing systolic scheduling's benefit without SUDS. However, the larger difference between *Eureka* and *Eureka-no-SUDS* than that between *Optimal SUDS* and *Eureka* shows that systolic scheduling is more effective when the critical paths are shortened by SUDS (Section 3.3). We see similar trends for all the benchmarks and pruning levels.

5.3 Energy

Figure 13 presents the compute energy *including on-chip buffers* estimated using our ASIC synthesis and off-chip memory energy (Y axis) for *Dense*, *Ampere/STC*, *Cnvlutin-like*, *Eureka* $P = 2$, *Eureka* $P = 4$, *DSTC*, *SparTen* and *S2TA* normalized to that of *Dense*. In addition to our benchmarks, we also include an unpruned, dense model, *Dense Bench*, which shows the sparse schemes' energy overhead for such models. Accel-sim [13] shows that for dense GEMM kernels, the workhorse of many ML workloads, running on a GPU, the off-chip memory's share of the total energy is 10-20% which is not surprising given the heavily compute-bound nature of the kernels. We choose the upper end of the range favoring two-sided schemes which save memory energy for both activations and filters unlike *Eureka* which saves only filter memory energy. We set *Dense Bench*'s compute-memory energy split to be 80-20 by calibrating the relative energy cost of a memory access with respect to that of a MAC operation in the *Dense* architecture. We then apply this relative cost to the other benchmarks whose compute-memory split may be different from 80-20 depending on each benchmark's operations per byte (e.g, *mobilenet* has fewer operations per byte and hence a higher share of memory energy than *ResNet50*). As before, We show at the far right the mean and the representative mean, which do *not* include *Dense Bench*.

In Figure 13, as expected, *Ampere/STC* dissipates more than 50% energy across the sparse benchmarks due to its 2:4 sparsity after accounting for the sparsity overheads. *Cnvlutin-like* and *Eureka* dissipate lower energy than *Ampere/STC* due to their higher sparsity despite their modestly higher overheads. With a compaction factor

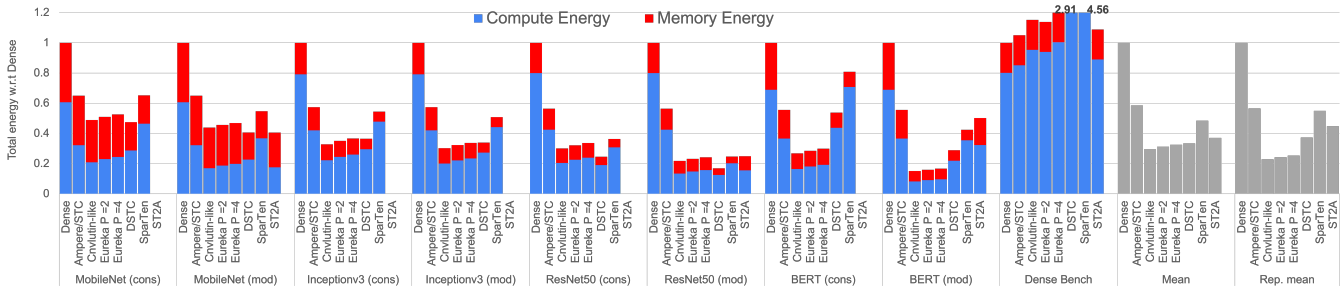

Figure 13: Total energy

of 4 but no SUDS overheads, *Cnvlutin-like* dissipates slightly lower energy than *Eureka* $P = 2$ and *Eureka* $P = 4$ which employs larger multiplexers and buffering, and hence dissipates slightly higher energy, than *Eureka* $P = 2$. By exploiting two-sided unstructured sparsity, *DSTC* dissipates less energy (compute and memory) than *Eureka* for the CNNs but not for *BERT* whose nearly-dense activations incur high compute energy overhead *which outstrips DSTC's memory energy advantage over Eureka. SparTen* incurs significant energy overhead due to its prefix sum and priority encoder logic and large buffering. *S2TA* exploits two-sided structured sparsity, which is lower than *DSTC*'s sparsity, to achieve lower energy than the one-sided schemes and *SparTen* but equal to or higher than *DSTC*. The exception is *BERT* where *S2TA*'s structured sparsity is much lower than that of the other schemes' unstructured sparsity (Table 1). *Ampere/STC* incurs around 6% average compute energy overhead for *Dense Bench*, whereas *Eureka* $P = 4$ increases this overhead to around 20%. The higher energy is due to the sparsity hardware which should be minimal as the same tensor core hardware is used for dense operations as well. As expected, the two-sided schemes *DSTC* and *SparTen* incur high overheads whereas *S2TA* remains efficient by foregoing two-sided sparsity for performance.

Overall, *Eureka* $P = 4$ saves 3.1x mean energy over *Dense* and 1.8x over *Ampere/STC*, while *Eureka* $P = 2$ saves 3.4x and 1.8x energy over *Dense* and *Ampere/STC*, respectively. Two-sided *DSTC* and *SparTen* achieve, respectively, slightly worse and worse mean energy reductions than *Eureka* whereas *S2TA*'s mean is overwhelmed by *BERT*. As before, *rep mean* tilts the mean towards *BERT* where *Eureka* achieves significantly lower energy than *DSTC*, *SparTen* and *S2TA*.

Combining performance (Figure 11) and energy (Figure 13), while two-sided *SparTen* achieves better mean performance but higher mean energy than *Eureka*, *rep mean* shows *Eureka* to be significantly better in *both* performance and energy.

5.4 Area, power and delay analysis

Table 2 shows per-MAC estimates for the area and power of the key components of Ampere and *Eureka* $P = 4$, as well as DSCT' cross bars and SparTen's prefix sum and priority encoder logic and buffers. Ampere's total includes a MAC and a 4-1 multiplexer. *Eureka*'s total includes a MAC, a carry-save adder, a 16-1 multiplexer, and two 2-1 multiplexers. *Eureka* incurs area and power overheads of 6% and 11.5%, respectively, over Ampere. In comparison, only DSTC's cross bars, ignoring any other overheads, and SparTen's logic and buffers contribute, respectively, 89% and 72% area and 38% and 6.5% power over Ampere. Adding other components of these schemes will only favor *Eureka*. Finally, our Ampere design achieves 1.66 ns latency which increases to 1.84 ns (11%) for *Eureka*. Both latencies will likely reduce with better commercial tools than our public-domain tools and some pipelining, enabling 1-2 GHz clocks for both designs.

Table 2: ASIC 15 nm Area and power

Component	Area (μm^2)	Power (μW)
MAC	1230	771
FP carry-save adder (per MAC)	43	47
16-1 Multiplexer (per MAC)	32	43
4-1 Multiplexer (per MAC)	16	14
2-1 Multiplexer (per MAC)	8	7
DSTC Crossbar (per MAC)	1105	299
SparTen logic (per MAC)	250	21
SparTen buffers (per MAC)	648	30
Total Ampere (per MAC)	1246	785
Total Eureka (per MAC)	1321	875

5.5 Sensitivity to MAC array size

Figure 14 shows the mean and *rep mean* speedups over *Dense* for moderate and conservative pruning, as we vary the MAC array size as 4x4, 8x8, and 16x16 in two ways: plain scale up (*8x8-plain* and *16x16-plain*) and systolic scale up using 4x4 as building blocks (*8x8-systolic* and *16x16-systolic*), as discussed in Section 2.3. In *both* dense and sparse tensor cores (i.e., not specific to *Eureka*), plainly-scaled-up larger arrays achieve higher reuse (lower memory bandwidth and energy) and higher area efficiency (shared buffers). However, such larger arrays incur slower clocks and higher power (e.g., operand broadcast to 8 MACs instead of 4). Further, such larger arrays both have a higher chance of unbalanced rows despite SUDS and lose more utilization for the same unbalanced row length than smaller arrays, causing significant performance loss as shown in Figure 14. Fortunately, systolically-scaled up larger arrays can achieve the same reuse and area efficiency as plainly-scaled up larger arrays with modest performance loss, nearly obviating this trade-off. Across conservative and moderate sparsities, the moderate sparsity

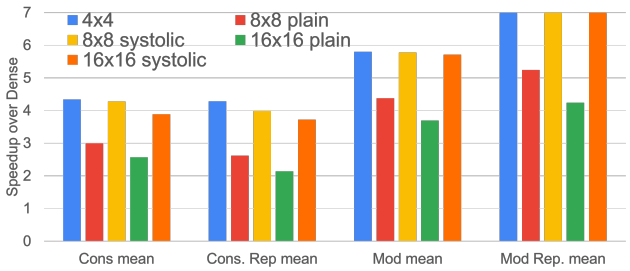

Figure 14: Sensitivity to array size

distributions are better-balanced than the conservative distributions leading to fewer systolic bubbles.

6 RELATED WORK

Two-sided sparse proposals not based on tensor cores [6, 7, 22, 31, 34] improve performance albeit at considerable energy (Section 2.2). The one-sided Sparse Tensor Core (STC) [35] suggests pruning for structured sparsity, preceding Ampere's 2:4 structured sparsity. Such pruning loses accuracy beyond 75% sparsity while unstructured sparsity provides much higher opportunity. The Dual-Sided Sparse Tensor Core (DSTC) [29] closely resembles SCNN except for using a bit-mask representation like SparTen [6] to reduce SCNN's partial sum address calculation overhead [22]. However, by abandoning output stationarity like SCNN, DSTC incurs the merge overhead in outer product, requiring an expensive crossbar for scatter-gather exactly like SCNN. As such, DSTC's performance comes with high energy and area costs for the widely-used transformers which do not have activation sparsity. As discussed in Section 4, S2TA [16] exploits one-sided activation sparsity for performance and two-sided sparsity for energy.

Bit-tactical [4] suggests work stealing from the adjacent bit lanes using a large look-ahead window for finding bit-level work. Such windows, when applied to 8- or 16-bit values, incurs considerable buffering and routing (e.g., [6]). Further, Bit-tactical adopts an online, sub-optimal, greedy approach which does not capture the full opportunity, as shown in Section 5.2. Our emphasis is on minimal extra hardware for constrained displacement with an optimal offline algorithm.

Other work [14] improves systolic utilization by combining sparse columns similar to matrix compaction used in this work (not our contribution). However, unlike matrix compaction, combining of multiple non-zero values eliminates all but one of the values and hence, leads to accuracy loss.

A few proposals [1, 4, 25] explore sparsity at bit level, referred to as *bit sparsity*. Bit sparsity, however, has the following shortcomings. (1) Most bit-sparsity is captured already by Booth encoding. (2) The papers report high bit sparsity (e.g., 99% or higher [4] implying only one '1' bit for 99 or more Booth-encoded '0' bits). Such unexpectedly-high bit sparsity stems from unconventional, *linear* quantization, mentioned briefly in one of the papers [4], where the entire actual value range is quantized linearly to capture a few outlier values. There are only 256 bins in INT8 for the entire real value range whereas most actual values are clustered closer to zero than not. Consequently, most of the linear bins are sparse with only a few outlier values whereas most actual values are crammed into a few bins near zero (e.g., -2, -1, 0, 1, 2, 3). While these near-zero bins have more Booth-encoded '0' bits giving rise to high bit sparsity, cramming the actual values into only a few bins would significantly degrade accuracy. Due to this reason, most real-world DNNs use *information-theoretic*, *saturating* quantization where the outlier values are saturated to the '255' or '-256' bins while spreading most of the actual values over all the 256 bins to retain more information content of the actual values and hence, high accuracy. However, such spreading also implies an even split between '0' bits and '1' bits, and significantly-diminished bit sparsity especially after Booth encoding.

Proposals on hyper-sparse outer product [10, 21, 26, 32] attempt to improve matrix multiplication for extreme sparsity. The techniques target large matrices (e.g., 10^6x10^6) with orders-of-magnitude lower sparsities than 1%. These methods involve merging partial products over large dimensions by storing in memory, along with expensive hardware for dynamic merging and sorting. Such methods are not compatible with the much lower sparsities of DNNs.

7 CONCLUSION

To meet the increasing compute demand for machine learning, GPUs provide tensor cores for dense matrix multiplication and structured one-sided (filter-only) sparse matrix multiplication. High energy efficiency for both sparse and dense operations is a primary constraint for tensor cores. Because increasing structured sparsity for higher speedups and energy savings leads to accuracy loss, exploiting unstructured sparsity is necessary. In contrast to structured sparsity's uniformity, unstructured sparsity causes load imbalance. Previous unstructured sparse custom or tensor core organizations incur high energy overheads caused mainly by sacrificing (input or output) operand stationarity in favor of higher utilization.

To address these issues, we proposed *Eureka* based on the key insight that a slight relaxation of output stationarity can achieve load balance most of the time while incurring only a modest overhead. Accordingly, we proposed *single-step uni-directional displacement (SUDS)*, an offline technique for inference, where a filter element's multiplication can either occur in its original position or be displaced to a vacant MAC in the adjacent row below while the accumulation occurs in the original row to restore output stationarity. We designed an optimal algorithm for work assignment for SUDS. To mitigate bubbles in the tensor cores' systolic pipeline due to the irregularity of unstructured sparsity, we propose offline *systolic scheduling* to group together the sparse filters with similar, statically-known execution times. Our evaluations showed that *Eureka* improves performance by 4.8x and 2.4x, and reduces energy by 3.1x and 1.8x over dense and 2:4 sparse (Ampere) implementations, respectively, and incurs area and power overheads of 6% and 11.5%, respectively, over Ampere. *Eureka*'s efficiency and applicability to a broad array of neural networks make it attractive option for tensor cores.

REFERENCES

[1] Jorge Albericio, Alberto Delmas, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium*

on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017. 382–394. https://doi.org/10.1145/3123939.3123982

[2] Jorge Albericio, Patrick Judd, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. 1–13. https://doi.org/10.1109/ISCA.2016.11

[3] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. 262–263. https://doi.org/10.1109/ISSCC.2016.7418007

[4] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3297858.3304041

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[6] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[7] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[8] Song Han, Huizi Mao, and William J. Dally. 2016. 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, Conference Track Proceedings*. http://arxiv.org/abs/1510.00149

[9] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 1135–1143. http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf

[10] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 84–96. https://doi.org/10.1109/HPCA51647.2021.00017

[11] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons from Three Generations Shaped Google's TPUv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[13] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[14] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 821–834. https://doi.org/10.1145/3297858.3304028

[15] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference. *IEEE Computer Architecture Letters* 19, 1 (2020), 34–37. https://doi.org/10.1109/LCA.2020.2979965

[16] Zhi-Gang Liu, Paul N. Whatmough, Yuhao Zhu, and Matthew Mattina. 2022. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 573–586. https://doi.org/10.1109/HPCA53966.2022.00049

[17] NCSU. [n. d.]. FreePDK45. https://www.eda.ncsu.edu/wiki/FreePDK15/.

[18] Neural Magic. 2021. Sparse Zoo. https://docs.neuralmagic.com/sparsezoo/

[19] Nvidia. [n. d.]. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[20] Nvidia. 2022. Nvidia Deep Learning Performance documentation. https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html. Updated: 2022-May-17.

[21] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[22] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[23] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. 2019. Modeling Deep Learning Accelerator-Enabled GPUs. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 79–92. https://doi.org/10.1109/ISPASS.2019.00016

[24] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. arXiv:1606.05250 [cs.CL]

[25] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. 2019. Laconic Deep Learning Inference Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 304–317. https://doi.org/10.1145/3307650.3322255

[26] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[27] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. https://doi.org/10.1016/j.vlsi.2017.02.002

[28] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[29] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-Side Sparse Tensor Core. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1083–1095. https://doi.org/10.1109/ISCA52012.2021.00088

[30] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071

[31] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[32] Z. Zhang, H. Wang, S. Han, and W. J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 261–274. https://doi.org/10.1109/HPCA47549.2020.00030

[33] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch. In *International Conference on Learning Representations*. https://openreview.net/forum?id=K9bw7vqp_s

[34] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

[35] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3352460.3358269